COSC 3P98 Computer Graphics
B. Ross, Winter 2021

Assignment 2

Due date: 9:00am Friday March 19, 2021. No lates accepted.

Objectives: 2D/3D computational geometry and OpenGL.

Hand in: All files and folders in directory structures on Sakai (as with assignment 1)

Note: You may work alone or in groups of 2 for this assignment. If in a group, all members must contribute equally to the assignment, and be able to answer questions about all submitted code.

Do **two (2)** of the following.

1. 2D Hull [25 + 8 = 33]

(a) Initialization: [4] There are 2 initialization modes:
(i) The default is “r” (random). The user may have entered a number N on the command line with your program, or perhaps a right-click menu option. Generate N unique random vertices (if they forgot N, just generate a default of 100). Alternatively, they may select a menu item, which adds 100 more random points each time selected.
(ii) The user can type “m” and then interactively enter points with the mouse and cursor. “m” quits the input mode. Be sure to check that the vertices given by the user are unique (ignore any that are not). When drawing the points from any input mode, use *glPointSize* so that they are large enough to see.
(b) Convex hull: [15] Implement any one of the convex hull algorithms discussed in class on the vertices in part (a). A command “c” will begin the convex hull computation and rendering. Save the hull edges in a data structure (necessary for question 2).
(c) Hull peel: [6] Repeatedly apply the convex hull routine to the set of points to create a peel, which is as follows. Find the hull for the points and draw it. Remove these hull points from the set, and iterate on the remaining points. Continue until less than 3 points remain.

Bonus [10]: **Peels on point clusters**. Select K random points in your set of points. K is supplied by user, and should be much less than the number of points total. For example, perhaps K=5. Then divide all the points in the set into 5 sets, where the points in each set is closest in distance to one of the 5 random points. Perform a hull peel on each of the 5 sets, and colour the peels on each of the K sets a different colour.

2. 2D Triangulation [25 + 8 = 33]

(a) Initialization*:* [4] See 1(a) above.
(b) 2D Triangulation: [10(trisection)+8(cleanup)=18] Implement the trisection/cleanup triangulation algorithms discussed in class. The command line should be: “triangulate N”, where N is the number of random points to use (OR, use a right-click GLUT menu selection). Draw the result. Save the triangles in a data structure. Count and print the number of random vertices and triangles created. In the case of the trisection, draw the progress of the cleanup algorithm as it runs. Count the number of times triangles were restructured, and print this number at the end.
(c) Lattice: [3] Apply the above to an N-by-N lattice of points. You might wish to have a special command to set up this data.

Bonus [10]: Do one of the following:
(1) Triangulated Painting. Implement an interactive application that creates images like the ones on this web site:
http://jonathanpuckey.com/projects/delaunay-raster/

Initially, read in an image file (see assignment 1, Q1). Then you will interactively select points on the image with your mouse. Every time you click a point, you add a new point to the triangulation. The triangulation is immediately recomputed and cleaned-up. Disallow duplicating the same vertices. Also, note that you need at least 3 non-collinear points to begin doing triangulation. Implement 2 different colour schemes for triangles: (i) Gouraud shading, where the colour of the vertex for a triangle is the colour of the pixel selected. (ii) Flat shading (same vertex colouring as Gouraud). Save your final triangulated image as an image file (via command key or menu; see Q1, assignment 1).
(2) Colour-coded triangles. Use a colour coding scheme in which each triangle's colour is chosen based on the triangle's area. For example, triangles with larger areas may be brighter than smaller triangles. Or, you may have a preset colour table that maps areas to colour indices. See Wikipedia for different ways to compute triangle areas. Note that you should be able to see triangle colours change as the cleanup is applied.

3. 3D Convex Hull [25 + 8 = 33]

(a) Initialization: [2] K unique random points in 3-space are generated (K is a command-line parameter; default 100). You should set appropriate bounds in X, Y, and Z directions.
(b) Centre of gravity: [2] Find the centre of gravity (avg coordinate) of the points in (a), and draw it as a large red dot or polygon.
(c) Convex hull: [15] Implement the greedy convex hull algorithm for 3 dimensions. Note that, instead of finding the distances between points and lines, you'll find distances between points and planes. You should store the hull in a data structure of polygons. A simplification is that you can assume that there will never be more than 3 co-planar points on a hull polygon. If you find more than 3 (i.e. fast distance of point to plane is zero for 4+ points), delete the extra points so that you have 3 distinct points on the plane that are not collinear).
(d) Rendering: [6] Draw the 3D hull from (c) by defining an appropriately sized volume with glOrtho. The hull should be centered in the middle of the volume, and the volume should be centered on the origin (0,0,0), with equal ranges positive and negative on the 3 axes. You will also need to use the Z-buffer. One command key will draw the hull using flat-shaded randomly coloured polygons. Another key will display the hull as a wire frame. Use the scheme in the file "rotate2.c" to spin your 3D hull interactively.

Bonus [10]: 3D spherical hull peel. Define a loop that creates a new set of random points a distance d from the origin (0,0,0), computes and renders its hull, and then repeats the process a given number of times for smaller and smaller d values. The result should be a set of concentric hulls. They will tend to look like spheres if you use many points. Render the result using transparent surfaces (OpenGL's alpha channel). You may also want to spin your object in 3D (see (d) above). This object is also easy to apply OpenGL lighting onto, because vertex normals are identical to vertex coordinates on a unit sphere centered at (0, 0, 0).

Comments:

Each question is worth 25 execution marks, plus 8 programming style/quality marks. Each question has an optional bonus part worth 10 marks. See the marking template for this assignment on the web site.

Implement your programs modularly. Make good use of functions and data structures. Test modules separately before integrating them into the full program. Test your programs on small data sets before using large ones. Use simple hand-computed test cases with known solutions to ensure your modules are working correctly. Make generous use of print statements for debugging. Be careful with formulas:

just one wrong sign in an equation can cause major headaches! Be aware of “end cases” for hulls and triangulations. Make sure your random points are **unique**!

For full marks, the program should process a general number of points specified by the user. This requires dynamic data structures. However, you can still get an excellent grade if you use fixed-sized data structures. As usual, you might want to use them first, and then generalize them after your program is working bug-free. Examples of dynamically allocated arrays and tables in C are in the 3P98 example folder on the web (eg. multidim.c, newsample.c and random_circle.c).

Finally, a friendly reminder to do your own work. Do not use algorithms or code from the internet. Be sure to follow instructions and implement the required algorithms. No credit will be given for alternative algorithms (Delaunay triangulation,...).