Michaelmas Term 2024
Geometric Modelling

Practical 1
Incremental algorithm for 2-D convex hulls

1 Admin

There will be three Geometric Modelling practicals, set in four sessions. You should get your work signed off in the last half hour of a session or at the beginning of the following one.

When checking your work the demonstrator will want to see a working version of the program in action, as well as appropriate commenting of your code and sketches indicating the design steps. Print out and explain the test data you used and why this data was appropriate for each case studied. Try to make your report as concise as possible, perhaps in the form of appropriate comments to your code.

2 2-D convex hulls

This practical task will illustrate the calculation of a planar convex hull through the medium of the Java programming language.

A fairly standard Computational Geometry textbook that contains details about this algorithm is J. O'Rourke's *Computational Geometry in C*, Cambridge University Press, 1994. You may find it useful to read the appropriate sections before solving the practical tasks.

3 Getting the files

In order to get the necessary files either download the zip file from the website or copy everything from the appropriate directory:

```
% cp /usr/local/practicals/gmod/ConvexHull/* .
```

To compile your code (after making appropriate changes), you can use the command

```
% javac *.java
```

and to run it, use

```
% java  ConvexHull &
```

When running the code, you are presented with a window. Clicking the left mouse button over it defines a new point. A few left clicks will generate the point set whose convex hull needs to be calculated. Note the origin is in the top left.

You may find it easier to input points by reading them from a file rather than clicking the mouse across the window. The button *Read sample points* assumes a file called `sample.data` exists in the current directory and causes a set of integer coordinate pairs to be read in order from this file. A new point set is generated as a result; the convex hull algorithm will run on this data set. You do not need to recompile the code in order to run it on a new `sample.data` file.

The button *Clear all* clears the screen and allows you to start a new point set.

The action expected when clicking the button *Convex hull* is that the convex hull of the point set you have just input should be computed and drawn. This is your task.

4 What you need to do

This practical is about implementing the Incremental Algorithm presented in the lectures. The input points have integer coordinates, so you are required to use *only integer arithmetic* throughout the practical.

You will probably make frequent use of the basic predicates `leftOn` and/or `left`, also discussed in the lectures. These predicates are available to you as methods of the class `MyPoint`. In particular, `boolean left(MyPoint a, MyPoint b)` tells you whether the current point (the one from which you are calling the method) is left of the line `ab`. The `MyPoint` method `leftOn` and `collinear` are defined in a similar way.

For the compulsory parts of the practical you need only make changes to the method `hullIncremental()` in the `MyPointSet` class.

The class `MyPointSet` represents the set of points as a `Vector` of `Point`s (using the *generic container* `Vector<MyPoint>`). Therefore you can use methods pertaining to the `Vector` class. In particular, `MyPoint elementAt(int index)` gives us the point at `index`. Also, when we need to manipulate a vector of `MyPoint`s we can use the methods

- `void setElementAt(MyPoint p, int index)` to set a particular element
- `void insertElementAt(MyPoint p, int index)` to set an element and up–shuffle the existing data
- `void removeElementAt(int index)` to delete an element and down–shuffle the remaining data

The number of elements in a `Vector` is given by `int size()`.

You will see that the `hullIncremental()` method calls a sorting method `sortByXY()`. This should implement some lexicographic sorting method (bubble sort will do fine) to make sure that the points appear in increasing x-coordinate (any ties in x should be sorted by y).

Task 1 Familiarise yourself with the code and its output. The convex hull which corresponds to the current point set is to be stored in `Vector theHull`. You may want to repeatedly output the contents of `theHull` while you are debugging your code. You will be pleased to notice that the method `void enumerateHull()` does just that.

The most challenging aspect of storing the point set in a `Vector` is the fact that its circular structure needs to be simulated explicitly. This is not too difficult, but it's a mildly tedious task, which you need not worry about: it is already there for you, in the form of the function `int removeChain(int bottom, int top)`. Given a couple of integer indices (say `bottom` and `top`) into a vector, `removeChain` removes the elements between those indices. Note that, if `bottom > top`, then the elements `bottom+1, ..., m-1, 0, 1, ... , top-1` are removed (where `m` is the current size of the convex hull). The method `removeChain` also prints out the indices of the elements removed, to make it easier for you to keep track of the behaviour of the algorithm. It also returns the index the undeleted point at the bottom of the range which is either `bottom` or the last element of the vector.

Task 2 Preparing the structures used in the incremental algorithm is a non-trivial task.

The main part of the assignment is to code the incremental algorithm, through the function `void hullIncremental()`. It is assumed that you will be filling a `Vector` called `theHull`. Once computed, `theHull` is then copied into a `Polygon` by `Polygon hullThePolygon()` so that it can be displayed.

The `Vector` called `theHull` must be initialised with the first three points in the set. Be careful, though: an invariant of the program says that `theHull` should be stored in anti-clockwise order so you may need to rearrange these first three points. You may assume at this stage that the first three points are not collinear, but please revisit this assumption when you come to Task 4.

Also, when adding any new point, it is assumed that it is the next one in the order by x, and that the point most recently added to `theHull` is visible from the new one. Arrange the first three points so that the one at index 2 is guaranteed to be visible from the point you are about to add.

Task 3 Once initialised correctly, `theHull` can now be filled with the rest of the data. The points are visited in increasing order of their x-coordinate.

The point most recently added to `theHull` should be visible from the point you are about to add. Starting from the most recently added point, work clockwise to find the `bottom` index of the chain you want to remove. When do you know you have found the point which lies at the boundary between light and darkness? Similarly, then work anti-clockwise to find the `top` of the chain. Again, when do you know you have found the boundary point?

You may find the methods `int hullnext(int i)`, `hullprevious(int i)` helpful here. They essentially let you increment and decrement an index on `theHull`, but wrap around at the ends. Calling `theHull.elementAt(int i)` gives you the given point on the current hull.

Once you have found the two boundaries between light and darkness, then simply call the `removeChain` method, in order to remove all the lit edges; they lie between

the two points found above. Remember that the function returns the index of the last valid element left in `theHull`; this will be useful when you need to know which is the first slot available for you to insert the new element into.

In order to make your code simple, try to work with the points' *indices* as often as possible. (You will hardly ever need to convert them to `MyPoint` – except in the cases where you actually need to manipulate with actual x and y coordinates.)

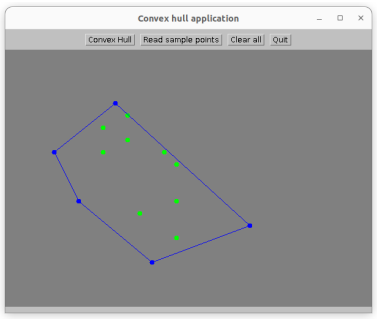
At the end, `theHull` should be the smallest convex polygon enclosing all the points input initially. Figure 1(a) shows an expected screen-shot using the file `sample.data` as input.

Task 4 It is an *optional* task of this practical to write special case code for dealing with duplicate points and collinear points. What happens if you click the mouse twice (or several times) at the same point? When does this affect the behaviour of your algorithm? List all the situations where a set of points has a valid convex hull which can't be computed by your program. What should your program do if *all* the points are collinear?

Now try to deal with the data contained in the file `box.data`. Figure 1(b) shows this awkward data set being dealt with in a satisfactory manner.

Please provide some screen-shots of your results. Use `Applications -> Accessories -> Take Screen Shot` or similar to help with this.

Joe Pitt-Francis
Joe.Pitt-Francis@cs.ox.ac.uk

(a) 

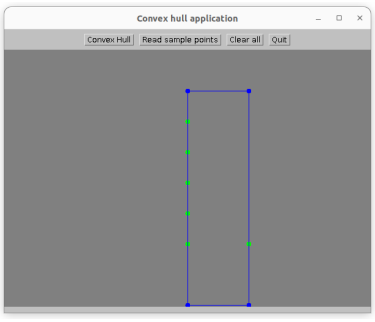
(b) 

Figure 1: The results from (a) Task 3 on the file `sample.data.0`, (b) Task 4 on the file `box.data`. Note that in the latter case the hull ring contains *only* four points.