Course Real-Time Rendering
Winter term 2019/20

Lab Class **December 2 and 3, 2019**

Collection of Exercises 3

Solutions due to: January 08, 2020

In this lab class you will implement the *Line-Sweep* algorithm. The algorithm will be based on a queue structure Q and a status structure T. Though we recommend to use C++, it is also possible to use JAVA as programming language. We suggest Visual Studio Community C++ 2019 (for Windows, it is free), Eclipse (Windows/Linux/Mac), Sublime (Windows/Linux/Mac) or build tools like *make*, *cmake*, *javac* as programming environment.

Submission and Grading

The exercise can be solved as a group of one or two students. We encourage you to thoroughly solve the exercise on your own. Plagiarism can lead to an exclusion from the course. If you have any problems or questions don't hesitate to ask your professor Rhadamés Carmona (rhadames.elias.carmona.suju@uni-weimar.de, Bauhausstr. 9a, third floor, guest room).

The solution for this exercise have to be submitted until **January 08, 2020, 11:59 pm** to our **moodle** website. The solutions will be evaluated by your professor using own test cases. The total number of points granted will be proportional to the number of correct answers to problems instances. The grading will be announced on moodle.

For grading, your implementations will be tested on the following online compilers:

- C++ will be tested on: http://cpp.sh/
- JAVA will be tested on: https://www.jdoodle.com/online-java-compiler/

For submission, you have to implement your solutions as single .cpp or .java files (i.e. `exercise31.cpp` and `exercise32.cpp` or `exercise31.java` and `exercise32.java`). The first line(s) must include your name(s) as comment in order to document authorship(s).

As an example, if you are a group of two students and submit in C++, a submission file should be structured as follows:

```
// author: <forename> <family name>
// author: <forename> <family name>

#include <iostream>

// function definitions can be placed here

int main(){
/*
your implementation starts here
*/
return 0;
}
```

As another example, if you are a single student and submit in JAVA, a submission file should be structured as follows:

```
// author: <forename> <family name>

public class MyClass {
    public static void main(String args[]) {
/*
your implementation starts here
*/
    }
}
```

We strongly recommend to test your code on these online compilers before submission. In particular, the output format of your implementations should be exactly as it is specified in the exercise. For grading, the output will be compared line by line with an online software (e.g. https://text-compare.com/). In order to obtain full grade, the output generated by your implementation has to be identical with the expected output specified in the exercise.

Exercise 3.1

[15 points]

Implement the *Line-Sweep* algorithm.

Input The input consists of one problem instance, starting with a single line containing an integer number n, $0 < n <= 1000$. The value of n represents the number of segments of the input. It follows with n lines. Each line contains one segment, represented by a unique `ID` and the coordinates x and y of each of its end points, in floating point format: `ID x0 y0 x1 y1`, with $-350 <= x, y <= 350$. The `ID` is a sequence of one or more alphabetic characters in lower case `a, b, c, d, ... x, y, z`. Assume that any pair of end points do not share the same x-coordinate or y-coordinate general position. Also, assume that intersection points do not share the same x-coordinate or y-coordinate with input points or with another intersection point. Finally, assume that segments are not parallel to the x-axis and y-axis.

Output Print the status structure T in one text line, and the queue structure Q in the next text line *after* processing each event. For printing T, just write the `ID` of the segments, from left to right, separated by one blank space. For printing Q, write the coordinates x and y of each event point rounded to one decimal, sorted by its priority (sorted in decreasing order by y-axis). After processing the last event, both T and Q are empty, and there is nothing else to print.

Sample input (See Fig. 1)

```
4
a -1 -2 8 7
b -3 8 4 1
c 2 11 14 -1
d 5 3 6 4
```

Sample output

```
c
-3.0 8.0 8.0 7.0 6.0 4.0 5.0 3.0 4.0 1.0 14.0 -1.0 -1.0 -2.0
b c
8.0 7.0 6.0 4.0 5.0 3.0 4.0 1.0 14.0 -1.0 -1.0 -2.0
b c a
7.0 6.0 6.0 4.0 5.0 3.0 4.0 1.0 14.0 -1.0 -1.0 -2.0
b a c
6.0 4.0 5.0 3.0 3.0 2.0 4.0 1.0 14.0 -1.0 -1.0 -2.0
b a d c
```

```
5.0 3.0 3.0 2.0 4.0 1.0 14.0 -1.0 -1.0 -2.0
b a c
3.0 2.0 4.0 1.0 14.0 -1.0 -1.0 -2.0
a b c
4.0 1.0 14.0 -1.0 -1.0 -2.0
a c
14.0 -1.0 -1.0 -2.0
a
-1.0 -2.0
```

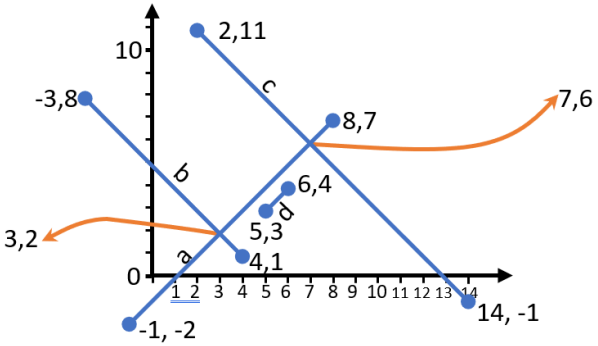

Abbildung 1: Sample input for exercise 3.1

Exercise 3.2

[5 points]

Extend the algorithm to deal with the following degenerate case: several pairs of input segments may share one of their endpoints $sep = (x_{shared}, y_{shared})$. Shared endpoints should **not** be treated as line intersections. Thus the *Line-Sweep* algorithm has to create one special event which involves two segments. We have some cases to deal with:

- If the shared endpoint is the upper endpoint of both segments, we have to insert both segments in the status structure T. The priority between these segments is given by the x-coordinate of the lower endpoint of both segments. The segment with lower x-coordinate should appear before the segment with the higher x-coordinate in T. Also, Q should be updated accordingly.

- If the shared endpoint is the upper endpoint of one segment (A) and the lower endpoint of the other segment (B), just replace A by B in T, and update Q accordingly.
- If the shared point is the lower endpoint of two segments, remove both segment from T and update Q accordingly.

Sample input (See Fig. 2)

```
4
u 4 3 9 8
r 13 2 3 5
so 3 5 5 10
nice 5 10 13 2
```

Sample output

```
so nice
9.0 8.0 3.0 5.0 4.0 3.0 13.0 2.0
so nice u
8.0 7.0 3.0 5.0 4.0 3.0 13.0 2.0
so u nice
3.0 5.0 4.0 3.0 13.0 2.0
r u nice
5.3 4.3 4.0 3.0 13.0 2.0
u r nice
4.0 3.0 13.0 2.0
r nice
13.0 2.0
```

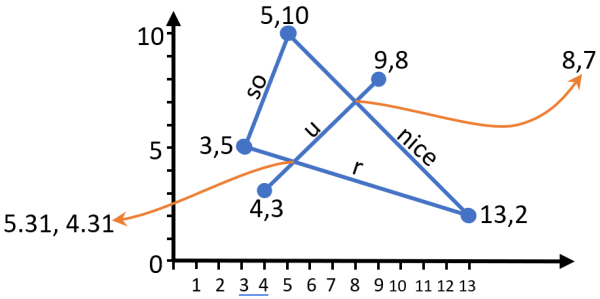

Abbildung 2: Sample input for exercise 3.2