Course Real-Time Rendering
Winter term 2019/20

Lab Class **January 06-07, 2020**

Collection of Exercises 4

Solutions due January 22, 2020

In this lab class you will implement the kD-tree data structure for efficient 2-dimensional range queries. The design explanation guide you towards an implementation. 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 22, 2020, 11:59 pm** to our **moodle** website. The solutions will be evaluated by your professor using own test cases, but also considering the design of the solution. Five (5) points consider the solution design, and the other 15 points granted will be proportional to the number of correct answers to problem 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 a single .cpp or .java file (i.e. `exercise412.cpp` and `exercise412.java`). **Note that exercise 4.1 (classes) and 4.2 (main function) are included in one file this time.** 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 4.1

[5 points]

Implement a class `boundingbox` that is formed by two (2D) points and provides at least the following methods:

- `contains(point)` that checks if a point is fully inside this bounding box.
- `inside(boundingbox)` that checks if the bounding box itself is fully inside another bounding box.
- `intersect(boundingbox)` that checks if the bounding box itself intersects with another bounding box.

The class `boundingbox` defines also the range that will be used during the range search.
Implement a class `kdnode` that encapsulates the following information:

- a flag whether it is an internal node or a leaf node.
- two pointers to its left and right children (in case it is an internal node).
- the dimension of the splitting line (in case it is an internal node).
- the position of the splitting line (in case it is an internal node).
- a (2D) point (in case it is a leaf node).
- a range of type `boundingbox` that stores its corresponding bounds (and implicitly the bounds of its subtree).

Hint: Include the necessary methods to be called from kdtree.

Implement a class `kdtree` that encapsulates the following information and methods:

- a `private` pointer to a kdnode (the root node of the kD-tree)
- a `public` method `build(P)` that builds the kD-tree from a set P of (2D) points.
- a `private` method `reportSubtree(kdnode* n)` that returns the set of all (2D) points which are stored in the leaves of the subtree.
- a `public` method `search(boundingbox range)` that performs a range search on the kD-tree and returns a set of (2D) points.

Hints: You have to implement also `private` methods that work in a recursive manner like `build(P, int depth)` and `search(kdnode* n, boundingbox range)`.
Implement a method `kdtree::count(range)`, which counts all points in the given range. Extend your data structures such that this method does not work output sensitive like `kdtree::search(range)` but with $O(\sqrt{n})$ time complexity instead.

Exercise 4.2

[15 points]

Write a main program that makes use of your classes implementation. Given a set of n 2D points and a 2D range query, report the points (or count the points) satisfaying the query rectangle.

Input The input starts with a single text line containing an integer number n, with $1 <= n <= 1000$. The next n lines contains the coordinates x and y (floating point) of each input point separated by one blank space. The last line of the input contains the query rectangle given by two points: (x_0, y_0) and (x_1, y_1), with $x_0 <= x_1$ and $y_0 <= y_1$, followed by a keyword `[PRINT|COUNT]`. Assume that points are not duplicated. However, they may share the same x-coordinate or y-coordinate.

Output If `keyword=PRINT`, the ouput represents the set of points inside the query rectangle. The points should be sorted in lexicographic order before being printed. Thus, they are sorted by y-axis (primary key) and x-axis (secondary key). Each output text line contains the coordinates of one 2D point, separated by one blank space: (x_i, y_i), rounded to two decimals. If `keyword=COUNT`, the output is a single line containing the number of points inside the query rectangle. You have to call the proper method `kdtree::count(range)` with $O(\sqrt{n})$ time complexity. Otherwise the result is not valid.

Sample input 1

```
4
3.176 2
0.5 0.5
4 4
2 4
1 1.5 4 4 PRINT
```

Sample output 1

```
3.18 2.00
2.00 4.00
4.00 4.00
```

Sample input 2

```
4
3.176 2
0.5 0.5
```

```
4 4
2 4
1 1.5 4 4 COUNT
```

Sample output 2

```
3
```