Hochschule München
Fakultät für Informatik

Computational Geometry

Praktikumsaufgabe 1

Team: Christopher Hinz, Tobias Gruber

1 Problemstellung

In dem Tar-File 'strecken.tgz' (s.u.) befinden sich Dateien mit jeweils 4 Koordinaten pro Zeile. Diese stellen jeweils die x- und y-Koordinaten eines Start- bzw. Endpunkts einer Strecke dar. Die Datei soll eingelesen und die Anzahl der sich schneidenden Strecken durch paarweises Testen herausgefunden werden. Zusätzlich wird die pro Datei aufgewendete Zeit gemessen. Es ist zu begründen, warum die Anzahl der vom Programm jeweils gefundenen Schnittpunkte korrekt ist.

2 Vorüberlegung

Um einschätzen zu können inwiefern die Ergebnisse sinnvoll sind sollen an dieser Stelle einige Vorüberlegungen getroffen werden.

Welche Anzahl an Schnitten sind zu erwarten bzw. in welchem Wertebereich sollten sich die Ergebnisse bewegen?
Die minimal zu erwartende Anzahl an Schnitten wäre für 1.000, 10.0000 und 100.000 Strecken jeweils null. Dies ist somit die untere Grenze des zu erwartenden Wertebereichs. Die maximale Anzahl an Schnitten würden in dem Fall auftreten, dass sich alle Strecken gegenseitig schneiden. Dann wäre nämlich für jede Strecke ein Schnittpunkt mit allen anderen Strecken gegeben. Das Problem ist in der Mathematik auch als Sektglasproblem bekannt. Die Berechnung erfolgt in Anlehnung an die Formel für die Dreieckszahlen. Für eine Anzahl von n Strecken ergeben sich somit k Schnittpunkte wie folgt:

$$k = \frac{(n-1) \cdot n}{2} \tag{1}$$

Damit wären für 1.000 Strecken maximal 499.500 (für 10.000 maximal 49.995.000, für 100.000 maximal 4.999.950.000) zu erwarten. Dies ist somit die obere Grenze des zu erwartenden Wertebereichs.

3 Umsetzung

3.1 Umsetzung der Aufgabenstellung

Das Programm verwendet die Funktion `read_dat()` um die Daten aus den Dateien einzulesen und über `pack_koords()` die Datensätze zu Strecken zusammenzufassen. Jede Strecke wird anschließend gegen alle anderen Strecken geprüft, ob sie diese schneidet, oder nicht. Zur Feststellung, ob sich die zwei Strecken schneiden wird ausgewertet, ob die Punkte p_1, p_2 der ersten Strecke zu den Punkten p_1, p_2 der zweiten Strecke im oder gegen den Uhrzeigersinn angeornet sind. Dafür wird folgende Formel verwendet, die im Ergebnis den Betrag des Kreuzproduktes aus den Richtungsvektoren

der drei Punkte enthält:

$$ccw(p,q,r) := \begin{vmatrix} p_1 & p_2 & 1 \\ q_1 & q_2 & 1 \\ r_1 & r_2 & 1 \end{vmatrix} = (p_1q_2 - p_2q_1) + (q_1r_2 - q_2r_1) + (p_2r_1 - p_1r_2) \tag{2}$$

Ist das Prdukt der ccw-Funktion der ersten Strecke mit den Endpunkten der zweiten Strecke, sowie das Produkt der ccw-Funktion der zweiten Strecke mit den Endpunkten der ersten Strecke kleiner oder gleich 0, so kann unter der Annahme, dass die Strecken nicht kollinear zueinander sind davon ausgegangen werden, dass sich die Strecken schneiden.
Sind die Strecken Kollinear angeordnet, so ergeben beide Kreuzprodukte 0. In diesem Fall muss noch überprüft werden, ob sich die Strecken überlappen. Bei Überlappung der Strecken lässt sich mindestens einer der Endpunkte der zweiten Strecke über ein Vielfaches zwischen 0 und 1 des ersten Streckenvektors addiert mit einem Endpunkt der ersten Strecke darstellen. Da zuvor bereits festgestellt wurde, dass die Strecken kollinear sind, kann das Skalar zum Richtungsvektor der zweiten Strecke über das Auflösen der parametrischen Darstellung durch die x-Komponenten der Orstvektoren aufgelöst werden.

$$\lambda = \frac{x_r - x_p}{x_q - x_p} \tag{3}$$

Liegt der Wert von λ zwischen 0 und 1, so überlappen sich die Funktionen. Nimmt λ den Wert 0 oder 1 an, schneiden sich die Strecken in in jeweils einem der Enpunkte.

```
#include <fstream>
#include <iostream>
#include <string>
#include <vector>


struct point{
    float x;
    float y;
};

struct line{
    point p1;
    point p2;
};


void read_dat(char* filename, unsigned int N, std::vector<float>& v){
    std::ifstream file;
    file.open(filename);
    float data;
    for(int i = 0; i < N; ++i){
        file >> data;
        v.push_back(data);
    }
```

```
26      file.close();
27  }
28
29
30  void pack_koords(std::vector<float>& source, std::vector<line>& target){
31      line temp;
32      for(unsigned int i = 0; i < source.size()/4; ++i){
33          temp.p1.x = source[0+i];
34          temp.p1.y = source[1+i];
35          temp.p2.x = source[2+i];
36          temp.p2.y = source[3+i];
37          target.push_back(temp);
38      }
39  }
40
41  float ccw(point p, point q, point r){
42      float res = (p.x*q.y - p.y*q.x)+ (q.x*r.y - q.y*r.x) + (p.y*r.x  - p.x
        *r.y);
43      return res;
44  }
45
46  bool line_intersect_check(line l1, line l2){
47      bool retval = false;
48      float ccw_res1 = ccw(l1.p1, l1.p2, l2.p1) * ccw(l1.p1, l1.p2, l2.p2);
49      float ccw_res2 = ccw(l2.p1, l2.p2, l1.p1) * ccw(l2.p1, l2.p2, l1.p2);
50      if(ccw_res1 <= 0.0f && ccw_res2 <= 0.0f){
51          retval = true;
52          if(ccw_res1 == 0.0f && ccw_res2 == 0.0f){
53              float lambda1 = (l1.p1.x-l2.p1.x)/(l2.p2.x-l2.p1.x);
54              float lambda2 = (l1.p2.x-l2.p1.x)/(l2.p2.x-l2.p1.x);
55              if ((lambda1 < 0 || lambda1 > 1) && (lambda2 < 0 || lambda2 >
        1)) retval = false;
56          }
57      }
58      return retval;
59  }
```

Listing 1: p1_lib.h: Bibliotheksfunktionen

```
1  #include "p1_lib.h"
2  #include <chrono>
3
4
5  int main(){
6      auto start = std::chrono::steady_clock::now();
7
8
9      std::vector<float> vec;
10     std::vector<line> lines_vec;
11     read_dat((char*)"strecken/s_1000_1.dat", 1000, vec);
12     pack_koords(vec, lines_vec);
13
14
```

```
15      int intersect_counter = 0;
16      for(unsigned int i = 0; i < lines_vec.size(); ++i){
17          for(unsigned int j = 0; j < lines_vec.size(); ++j){
18              if(i!=j){
19                  if(line_intersect_check(lines_vec[i], lines_vec[j])){
20                      ++intersect_counter;
21                  }
22              }
23          }
24      }
25
26      std::cout << "Strecken insgesamt: " << vec.size() << "\n" << "Schnitte
        zweier Strecken: " << intersect_counter << "\n";
27
28      auto end = std::chrono::steady_clock::now();
29      std::cout << "Runtime: "
30      << (float)std::chrono::duration_cast<std::chrono::microseconds>(end -
        start).count()/1000
31          + std::chrono::duration_cast<std::chrono::milliseconds>(end -
        start).count() << " ms\n";
32
33      return 0;
34  }
```

Listing 2: strecken.cpp: Aufgruf der Bibliotheksfunktionen

In Listing 3 wird die Ausgabe des Programmes zur Schnittpunktezählung aufgeführt. In der Ausgabe ist zu erkennen, dass die Anzahl der Schnitte für alle Fälle (1.000, 10.000, 100.000) im zu erwartenden Wertebereich liegt. Schnittpunkte:

- 1.000 Strecken: 0 <= 62.250 <= 499.500
- 10.000 Strecken: 0 <= 6.247.500 <= 49.995.000
- 100.000 Strecken: 0 <= 624.975.000 <= 4.999.950.000

```
# s_1000_1.dat
Strecken insgesamt: 1000
Schnitte zweier Strecken: 62250
Runtime: 14.408 ms
#s_10000_1.dat
Strecken insgesamt: 10000
Schnitte zweier Strecken: 6247500
Runtime: 580.073 ms
#s_100000_1.dat
Strecken insgesamt: 100000
Schnitte zweier Strecken: 624975000
Runtime: 55100.2 ms
```

Listing 3: testing.cpp: Ausgabe Konsole

3.2 Programm zum Testen der Funktion

Um die korrekte Funktionsweise des Programmes zu verifizieren werden verschiedene Testfälle definiert. Die Testfälle decken unterschiedliche Streckenanordnungen ab und sollen dadurch die in der Vorlesung besprochenen Grenzfälle für Streckenschnitte überprüfen. Abbildung 1 zeigt die implementierten Testfälle.

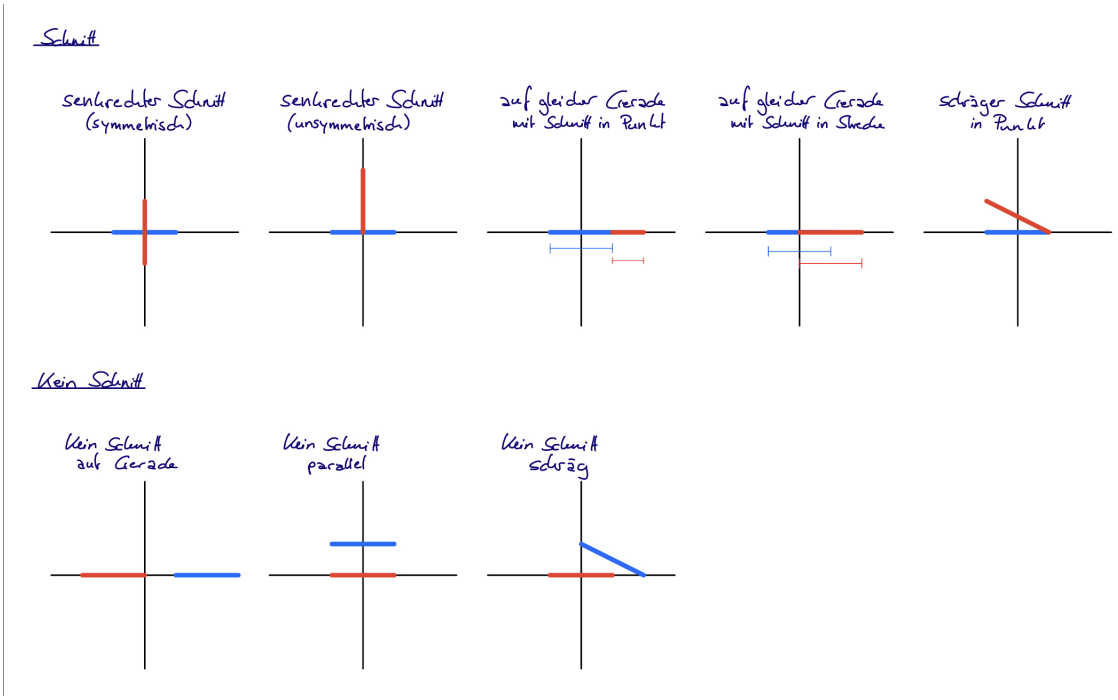

Abbildung 1: Testfälle für Streckenschnitte

Listing 4 zeigt die Implementierung der Testfälle in C++.

```
1  #include "p1_lib.h"
2
3  void print_result(line l1, line l2, bool b_soll){
4      std::cout << std::boolalpha << "Soll: " << b_soll << ", Ist: " <<
       line_intersect_check(l1, l2) << "\n";
5  }
6
7
8  int main(){
9
10     // Verschiedene Schnitte
11
12     // Strecken mit senkrechtem Schnitt (symmetrisch)
13     // p1 = (0,1) nach p2 = (0,-1)
14     // p1 = (1,0) nach p2 = (-1,0
15     print_result(line{.p1.x = 0, .p1.y = 1, .p2.x = 0, .p2.y = -1}, line{.
       p1.x = 1, .p1.y = 0, .p2.x = -1, .p2.y = 0}, true);
16
```

```
17      // Strecken mit senkrechtem Schnitt (unsymmetrisch)
18      // p1 = (0,2) nach p2 = (0,0)
19      // p1 = (1,0) nach p2 = (-1,0)
20      print_result(line{.p1.x = 0, .p1.y = 2, .p2.x = 0, .p2.y = 0}, line{.
        p1.x = 1, .p1.y = 0, .p2.x = -1, .p2.y = 0}, true);
21
22      // Strecken auf gleicher Gerade mit Schnitt in einem Punkt
23      // p1 = (1,0) nach p2 = (2,0)
24      // p1 = (1,0) nach p2 = (-1,0)
25      print_result(line{.p1.x = 1, .p1.y = 0, .p2.x = 2, .p2.y = 0}, line{.
        p1.x = 1, .p1.y = 0, .p2.x = -1, .p2.y = 0}, true);
26
27      // Strecken auf gleicher Gerade mit Schnitt in einer Strecke
28      // p1 = (0,0) nach p2 = (2,0)
29      // p1 = (1,0) nach p2 = (-1,0)
30      print_result(line{.p1.x = 0, .p1.y = 0, .p2.x = 2, .p2.y = 0}, line{.
        p1.x = 1, .p1.y = 0, .p2.x = -1, .p2.y = 0}, true);
31
32      // Strecken auf gleicher Gerade mit Schnitt in einer Strecke
33      // p1 = (0,0) nach p2 = (2,0)
34      // p1 = (1,0) nach p2 = (-1,0)
35      print_result(line{.p1.x = 1, .p1.y = 0, .p2.x = -1, .p2.y = 1}, line{.
        p1.x = 1, .p1.y = 0, .p2.x = -1, .p2.y = 0}, true);
36
37
38      // Verschiedene ohne Schnitt
39
40      // Strecken ohne Schnitt auf Gerade
41      // p1 = (-2,0) nach p2 = (0,0)
42      // p1 = (1,0) nach p2 = (3,0)
43      print_result(line{.p1.x = -2, .p1.y = 0, .p2.x = 0, .p2.y = 0}, line{.
        p1.x = 1, .p1.y = 0, .p2.x = 3, .p2.y = 0}, false);
44
45      // Strecken ohne Schnitt parallel
46      // p1 = (-2,0) nach p2 = (0,0)
47      // p1 = (-2,1) nach p2 = (0,2)
48      print_result(line{.p1.x = -2, .p1.y = 0, .p2.x = 0, .p2.y = 0}, line{.
        p1.x = -2, .p1.y = 1, .p2.x = 0, .p2.y = 2}, false);
49
50      // Strecken ohne Schnitt schraeg
51      // p1 = (-2,0) nach p2 = (0,0)
52      // p1 = (0,1) nach p2 = (2,0)
53      print_result(line{.p1.x = -2, .p1.y = 0, .p2.x = 0, .p2.y = 0}, line{.
        p1.x = 0, .p1.y = 1, .p2.x = 2, .p2.y = 0}, false);
54
55      return 0;
56  }
```

Listing 4: testing.cpp: Testen der Bibliotheksfunktionen

Listing 5 zeigt, dass für die Testfälle die erwarteten Ergebnisse erzielt werden. Das

bedeutet, dass für alle Streckenpaare, die wie in den Testfällen angeordnet sind, die Schnittstellenfunktion korrekt berechnet ob sie sich schneiden oder nicht.

```
Soll: true, Ist: true
Soll: true, Ist: true
Soll: true, Ist: true
Soll: true, Ist: true
Soll: true, Ist: true
Soll: false, Ist: false
Soll: false, Ist: false
Soll: false, Ist: false
```

Listing 5: testing.cpp: Ausgabe Konsole