Computational Geometry

Ausarbeitung für das Praktikum

Studienarbeit von

Maximilian Hempe

Roland Wilhelm

Dozent: Dr. Fischer

Hochschule München

Master Informatik

15. Juli 2013

Inhaltsverzeichnis

Abbildungsverzeichnis

Listingsverzeichnis

1 Einleitung

Diese Studienarbeit beschreibt das Praktikum zur Vorlesung Computational Geometry. Die Studenten sollen darin den Umgang mit Mathematischen Problemen in Zusammenhang mit Computern oder Robotern lernen. Dafür sollen relativ einfache Probleme mit einer beliebigen Programmiersprache und selbst erstellten Algorithmen gelöst werden. Im Praktikum werden nur die Aufgaben und Testdaten zur Verfügung gestellt, weitere Werkzeuge gibt es nicht.

Auf dieser Basis werden im Verlauf mehrere Aufgaben erarbeitet. Diese vertiefen die Themen der Vorlesung und gehen auf spezielle Sachverhalte intensiver ein. Ziel ist es meistens Schnittpunkte, ihre Anzahl und Flächen zu bestimmen. Das Praktikum zeigt, dass es beliebig komplex werden kann triviale Mathematische Probleme in performante Algorithmen zu konvertieren.

2 Aufgabe 1 - Anzahl von Schnittpunkten berechnen

In dieser Aufgabe soll eine Datei mit Liniensegmenten eingelesen werden und die Liniensegmente anschließend auf Schnittpunkte überprüft werden. Ein Liniensegment besitzt im Gegensatz zu einer Geraden einen Start- und einen Endpunkt. Diese Punkte sind je durch X- und Y-Koordinate definiert. Eine Zeile in der einzulesenden Datei enthält pro Zeile die beiden Koordinaten von Beginn und Ende des Segments. Ein Schnittpunkt ist dann vorhanden, wenn die Segmente mindestens einen gemeinsamen Punkt besitzen.

Der Algorithmus sollte zu Beginn möglichst einfach sein, deshalb wird jede Linie mit jeder anderen Linie auf einen Schnittpunkt getestet. Dadurch entsteht eine abstrakte Laufzeit von $O(n^2)$.

Um den Algorithmus testen zu können gab es drei verschiedene Files, die sich vor allem in der Anzahl der Input-Datensätze unterscheiden. Eine Datei beinhaltet 1000 Segmenten, die Zweite 10.000 Segmente und die Dritte 100.000 Segmente. Die tatsächliche Laufzeit verlängert sich bei Verzehnfachung des Inputs um ca. das 100 Fache.

2.1 Softwaredesign und Datenstruktur

Um die Anforderungen der Aufgabe zu Erfüllen, wurde das in Abbildung 1 dargestellte Klassendiagramm entworfen. Dabei wird jede Koordinate mit der Klasse `Point` als ein Punkt einer Strecke dargestellt. Zwei Punkte ergeben eine Strecke und wird durch die Klasse `Linie` repräsentiert. In der Klasse `Linie` werden des weiteren Funktionen zur Verfügung gestellt, ob sich zwei Strecken schneiden zuzüglich die Hilfsfunktion *ccw (Counter Clockwise)*, die entscheidet auf welcher Seite einer Strecke sich ein Punkt befindet. Mit der Klasse `LineFile` werden die verschiedenen Dateien eingelesen und jede Zeile bzgl. ihrer Koordinatenwerte analysiert und als Strecke abgespeichert. Während des einlesen wird die Strecke überprüft, ob sie eine Strecke oder nur ein Punkt ist und das Flag *is_line* entsprechend gesetzt. Außerdem kann ein Timer gestartet werden, um die Berechnungszeiten zu berechnen sowie Funktionen um die Berechnungen zu starten und das Ergebnis auszugeben.

2.2 Ablauf des Algorithmus

Nachdem die Segmente eingelesen sind, wird die Funktion zur Berechnung der sich schneidenden Linien aufgerufen. Darin enthalten ist die Zeitmessung, diese wird direkt am Funktionsbeginn gestartet und vor der Rückgabe beendet.

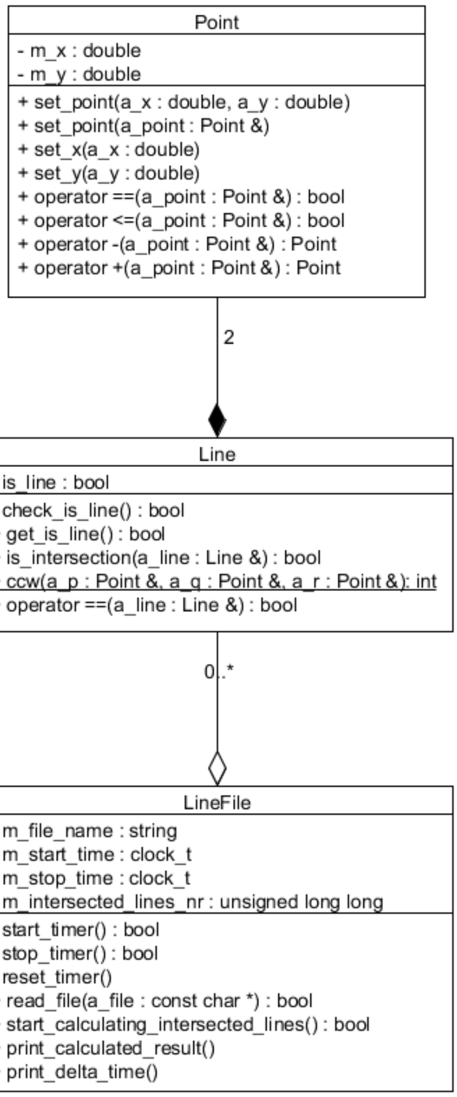

Abbildung 1: Klassendiagramm: Entwurf und Datenstruktur der Aufgabe 1

Jede Linie muss immer nur ein Mal mit jeder anderen Linie auf mindestens einen gemeinsamen Punkt getestet werden. Deshalb wird der Test in einer verschachtelten Schleife so realisiert, dass die innere Schleife immer nur nachfolgende Linien abfrägt. Doppelte Abfragen, wie Linie 3 mit Linie 4 und Linie 4 mit Linie 3, werden somit verhindert.

```
for(unsigned int i = 0; i < m_lines.size(); i++) {
  for(unsigned int j = i+1; j < m_lines.size(); j++) {

    if(m_lines[i]->is_intersection(m_lines[j]) == true) {
      m_intersected_lines_nr++;
    }
  }
}
```

Listing 1: Schleifenkostrukt zur Schnittpunktsuche

Ergibt der Test auf einen Schnittpunkt ein true, wird eine Membervariable inkrementiert. Der Algorithmus terminiert wenn das Array, das die Segmente beinhaltet, komplett geprüft wurde. Das Ergebnis ist die Anzahl der Schnittpunkte und wird durch die Membervariable festgehalten.

2.3 Test auf Schnittpunkte

Der Test ob es einen Schnittpunkt gibt erfolgt geschachtelt und wird durch die Funktion ccw(Punkt, Punkt, Punkt) unterstützt. Die Funktion bestimmt die Fläche, die durch die drei übergebenen Punkte aufgespannt wird. Je nach Vorzeichen der Fläche kann nun bestimmt werden, ob das Dreieck rechtsdrehend, linksdrehend oder flach ist. Flach bedeutet in diesem Fall, dass die drei Punkte auf einer Geraden liegen. Durch die Drehrichtung des Dreiecks kann bestimmt werden auf welcher Seite der Linie der dritte Punkt liegt.

```
int Line::ccw_max(Point &a_p, Point &a_q, Point &a_r){

  //x und y koordinaten der Punkte
  double result;

  result = (a_p.get_y()*a_r.get_x())-(a_q.get_y()*a_r.get_x());
  result += (a_q.get_x()*a_r.get_y())-(a_p.get_x()*a_r.get_y());
  result += (a_p.get_x()*a_q.get_y())-(a_p.get_y()*a_q.get_x());

  if (result < 0.0)
    return -1;
  else if(result == 0.0)
    return 0;
  else
    return +1;
}
```

Listing 2: ccw-Funktion - test der Drehrichtung

2.3.1 Kolinearität

Zuerst werden die beiden Linien auf Kolinierität geprüft, das würde bedeuten, dass kein Dreieck, das aus den vier Punkten konstruiert wird eine Fläche aufspannt. Falls die Linie ein Punkt ist, wird nun vereinfacht geprüft, ob dieser Punkt auf der anderen Strecke liegt. Ist das der Fall, hat man bereits einen Schnittpunkt gefunden, falls nicht gibt es für diesen Fall keinen Schnittpunkt.

```
...
if ( ccw_max(m_start, m_end, a_line->m_start) == 0 && ccw_max(m_start, m_end, a_line->m_end)
    == 0) {
  if ( m_start == m_end ) {

    //Punkt liegt auf Linie??
    if( ( m_start > a_line->m_start && m_start < a_line->m_end )
      || (m_start < a_line->m_start && m_start > a_line->m_end)){
      return true;
    }
    else
      return false;
...
```

Listing 3: Schnittpunkttest von kolinearen Linien - Sonderfall Punkt

Ist die Linie regulär, hat sie also einen vom Startpunkt verschiedenen Endpunkt, muss ein Überlappungstest durchgeführt werden. Hierfür wird der Verktor aus Start- und Endpunkt um 90 Grad und -90 Grad gedreht, sodass zwei zusätzliche Punkte über der Linie entstehen. Nun werden neue Dreiecke erstellt, Dreieck 1 mit Startpunkt, Startpunkt der anderen Linie und dem generiertem neuen Punkt p und Dreieck 2 mit Endpunkt, dem Startpunkt der anderen Linie und dem anderen erzeugten Punkt q. Verlaufen die beiden Dreiecke in gleicher Richtung, überdecken sich die Segmente und es gibt einen Schnittpunkt. Falls die zwei Dreiecke nicht gleichdrehend sind, werden zwei neue Dreiecke mit dem Endpunkt der anderen Linie anstelle der Startpunktes erstellt und diese wiederum gestestet. Sind auch diese Dreiecke nicht gleichdrehend, gibt es keinen Schnittpunkt.

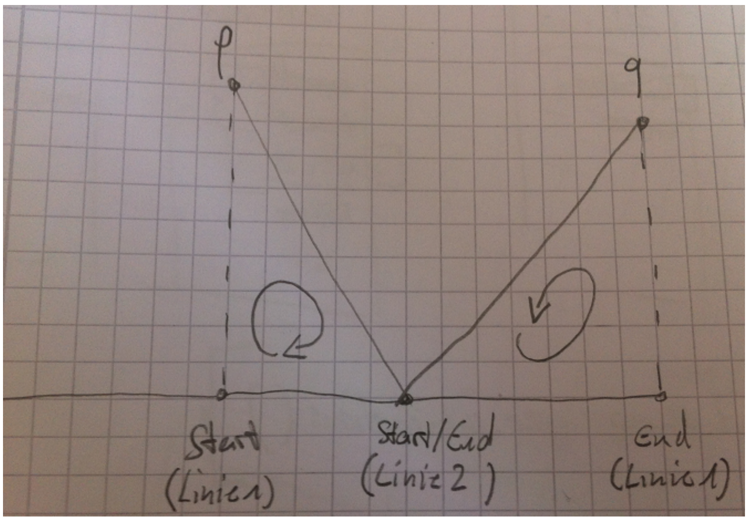

Abbildung 2: Struktur des Tests bei Kolinearität

```
...
else { //überlappungstest (line <-> line oder line <-> punkt)
```

```
      //p über m_start - drehung um -90°, q über m_end - drehung um 90° des gegengesetzten Vektor
    Point p(m_end.get_y()-m_start.get_y(), m_end.get_x()-m_start.get_x()),
          q(m_start.get_y()-m_end.get_y(), m_end.get_x()-m_start.get_x());

    //Start-Punkt auf der Linie (inkl Ränder)
    if( ccw_max(m_start, a_line->m_start, p)*ccw_max(m_end,q,a_line->m_start) >= 0 ){
      return true;
    }
    //End-Punkt auf der Linie (inkl Ränder)
    else if(ccw_max(m_start, a_line->m_end, p)*ccw_max(m_end,q,a_line->m_end) >= 0){
      return true;
    }
    else
      return false;
  }
}
...
```

Listing 4: Schnittpunkttest von kolinearen Linien

2.3.2 Normaler Schnittpunkt

Falls die beiden Strecken nicht auf der selben Gerade liegen, muss lediglich überprüft werden ob die beiden Punkte des einen Segments je auf verschiedenen Seiten der anderen Linie liegen. Dies kann wiederum mit der ccw-Funktion getestet werden. Die Betrachtung wird aus Sicht beider Linien gemacht, sonst würde eine seitlich versetzte Linie als Schnittpunkt gezählt.

```
...
else if ( (ccw_max(m_start, m_end, a_line->m_start)*ccw_max(m_start, m_end, a_line->m_end)) < 0
      && ccw_max(a_line->m_start, a_line->m_end, m_start)*ccw_max(a_line->m_start, a_line->m_end, m_end) < 0 ) {

  //->Schnittpunkt!!
  return true;
}
return false;
```

Listing 5: Schnittprüfung mit ccw-Funktion

Sind beide Tests negativ, gibt es keinen Schnittpunkt. Es kann also ein sonst generell gültiger Else-Fall erstellt werden, der false zurück liefert.

2.4 Erstellung Testdatei

Um die Funktionalitäten und Algorithmen zu testen, wurde eine Testdatei (data/-Strecken_test.dat) mit allen möglichen Szenarien erzeugt und das Programm damit gestartet. Das Listing 6 zeigt dabei das Resultat der Berechnungen. Es wurden 24 Zeilen (Strecken) eingelesen, dabei wurden mit 276 Streckenvergleichen 13 Schnittpunkte erkannt.

```
Start loading file C:/Users/Roalnd/Entwicklung/CG/cg/data/Strecken_test.dat
Start calculating file C:/Users/Roalnd/Entwicklung/CG/cg/data/Strecken_test.dat
Max. lines to compare: 276
-------------------------------------
File name: C:/Users/Roalnd/Entwicklung/CG/cg/data/Strecken_test.dat
Valid lines: 24
Invalid lines: 0
Compared lines: 276
Intersected lines: 13
DeltaT: 0 seconds
```

Listing 6: Ergebnisse für die Testdatei für alle möglichen Streckenarten

2.5 Präsentation der Ergebnisse

Nachdem die Testdatei richtige Ergebnisse liefert, werden nun die drei vorgegebenen Dateien geladen und die Berechnungen gestartet. Das Listing 7 zeigt die jeweiligen Ergebnisse mit ihren Schnittpunkten sowie die Berechnungszeiten.

```
Start loading file data/Strecken_1000.dat
Start calculating file data/Strecken_1000.dat
Max. lines to compare: 499500
Iterations: 10
-------------------------------------
File name: data/Strecken_1000.dat
Valid lines: 1000
Invalid lines: 0
Compared lines: 499500
Intersected lines: 111915
Average time: 0.0383 seconds
Min. time: 0.035 seconds
Max. time: 0.046 seconds
-------------------------------------
-------------------------------------
Start loading file data/Strecken_10000.dat
Start calculating file data/Strecken_10000.dat
Max. lines to compare: 49995000
Iterations: 10
-------------------------------------
File name: data/Strecken_10000.dat
Valid lines: 10000
Invalid lines: 0
Compared lines: 49995000
Intersected lines: 11695676
Average time: 5.2404 seconds
Min. time: 3.626 seconds
Max. time: 5.834 seconds
-------------------------------------
-------------------------------------
```

```
Start loading file data/Strecken_100000.dat
Start calculating file data/Strecken_100000.dat
Max. lines to compare: 4999950000
-------------------------------------
File name: data/Strecken_100000.dat
Valid lines: 100000
Invalid lines: 0
Compared lines: 4999950000
Intersected lines: 1155477297
DeltaT: 530.371 seconds
-------------------------------------
-------------------------------------
```

Listing 7: Ergebnisse für die unterschiedlichen Streackendateien

3 Aufgabe 2 - Berechnung von Flächen

In dieser Aufgabe werden die Flächen der einzelnen Bundesländer in Deutschland berechnet. Des Weiteren werden Algorithmen implementiert um Städte, angegeben durch x und y Werte, einem Bundesland zuzuordnen.

Dabei wird die Aufgabestellung zuerst Designet und die erforderlich Datenstruktur entwickelt. Anschließend wird ein *SVG-Parser* implementiert, um die Grafikdateien einzulesen und die erforderlichen Werte abzuspeichern. Dabei wird eine Testdatei erzeugt, um die Software und ihre Funktionalität zu testen. Schlussendlich wird das Programm auf die eigentliche Deutschlandkarte angewendet und die Ergebnisse präsentiert.

3.1 Softwaredesign und Datenstruktur

Um die Anforderungen der Flächenberechnung von Bundesländern sowie die Lokalisierung von Städten in Deutschland zu erfüllen, wurde das in Abbildung 3 dargestellte Klassendiagramm entworfen.

Jedes Bundesland wird durch eine Klasse `State` repräsentiert. Diese Klasse speichert die jeweilige Fläche sowie die dazugehörige Bounding Box des Bundeslandes ab. Diese Klasse kann dabei eine oder mehrere Flächen besitzen, um auch Bundesländer mit mehreren Flächen bzw. Bundesländer in Bundesländer darstellen zu können. Dies wird durch die Klasse `Area` behandelt. Jede Klasse `Area` hat dabei eine Fläche, Bounding Box sowie ein Flag *m_within_state*, welches kennzeichnet ob die Fläche wieder in einer Fläche liegt. Des Weiteren werden Städte durch die Klasse `City` abgebildet mit den Parametern Name der Stadt sowie deren Koordinaten. Mit der Methode *locate_and_push_city_to_state* wird die Stadt einem Bundesland zugeordnet und im entsprechenden Bundesland abgespeichert. Die Klasse `SvgFile` ist für das Einlesen und Parsen der SVG-Dateien verantwortlich. Hierdurch wird auch die Berechnung sowie die Darstellung der Ergebnisse gestartet.

3.1.1 Parsen der Grafikdatei

Um die erforderlichen Koordinatenwerte aus der SVG-Datei zu erhalten, wurde ein entsprechender Parser dafür entwickelt. Basis dafür war der XML-Parser *tinyXML2*, dieser besteht lediglich aus einer Quell- und Headerdatei und kann einfach in ein Projekt integriert werden. Dieser übernimmt das Lesen der XML-Struktur und gibt diese an den SVG-Parser weiter. Das Listing 8 zeigt das parsen entsprechender Zeilen sowie die Abspeicherungen der Koordinaten zu einer bestimmten Fläche bzw. Bundesland. Dabei werden aktuell folgende SVG-Kommandos unterstützt:

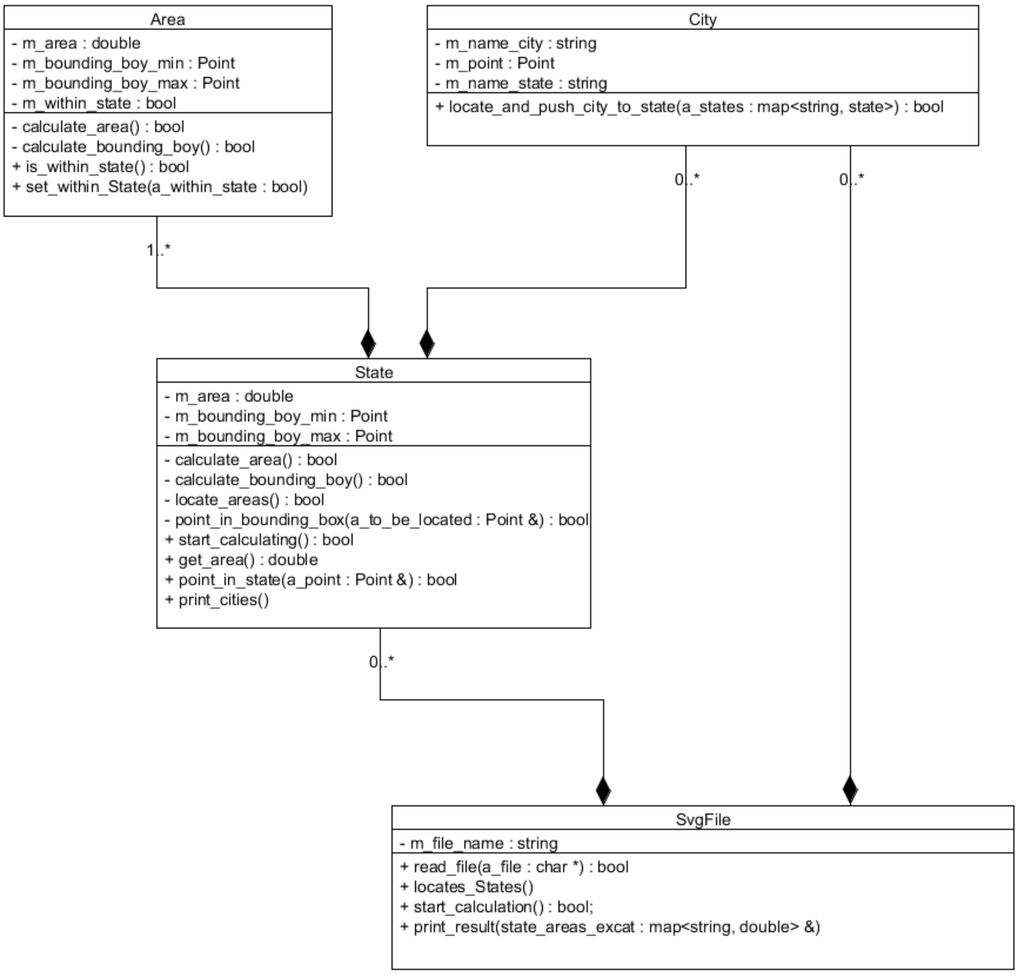

Abbildung 3: Klassendiagramm: Entwurf und Datenstruktur der Aufgabe 2

- M: Startpunkt des Polygons, die Werte x und y sind absolute Koordinaten.
- L: Zeichnet eine Strecke vom Startpunkt zum Endpunkt L. Dieser Endpunkt wird durch absolute Koordinatenwerte abgebildet.
- l: Hat die selbe Funktion wie L, der Endpunkt wird jedoch durch relative Werte abgebildet.
- H: Zeichnet eine horizontal Strecke. Angabe in absoluten Werten
- z: Schließt das Polygon, indem vom aktuellen Punkt zum Startpunkt eine Strecke gezeichnet wird.

Die Zeile 2 im abgebildeten Quellcode zeigt, wie jede SVG relevante Zeile nach den entsprechenden Kommandos aufgesplittet wird. Die wird dann der Funktion *svg2Point(...)* übergeben. In dieser wird der String in absolute Koordinaten konvertiert und anschließend aus den Punkten ein Point-Objekt erzeugt. Diese Point-Objekte werden solange zu der Fläche Area hinzugefügt, bis ein z-Kommando das Ende einer Fläche kennzeichnet und diese Flächen zu einem Bundesland (State) zuordnet. Dieser Vorgang wiederholt sich solange, bis alle Koordinaten der Bundesländer abgespeichert sind.

```
for(;;) {
  string::size_type diff = splitString(str, "MLlHz", &start);
  if(diff > 0) {
    // Token available
    string sub = str.substr(start, diff);
    if( (sub == string("z")) && (area.empty() == false) ) {
      // Push filled vector of points to the state and clear it for the next area
      m_states[string(id)].push_area_to_state(area);
      area.clear();
    }
    bool result = svg2Point(id, sub, &offset, area);
    if(result == false) {
      return false;
    }
    start += diff;
  }
  else {
    break;
  }
}
```

Listing 8: Parsen der SVG-Datei und Abspeicherung der Fläche

3.1.2 Flächenberechnung

Um den Flächeninhalt eines Polygons (hier Area) zu berechnen, wird die **Gaußsche Trapezformel**[1] verwendet. Hierbei werden alle Koordinaten, die zu einer bestimmten Fläche (Klasse Area) zugeordnet wurden, durch den Algorithmus berechnet (siehe Listing 9).

```
for(; iterPoints != (m_points.end() - 1); iterPoints++) {

  double x1 = iterPoints->get_x();
  double y1 = iterPoints->get_y();

  double x2 = (iterPoints+1)->get_x();
  double y2 = (iterPoints+1)->get_y();

  area += (( y1 + y2 ) * ( x1 - x2 ));
}

m_area = fabs(area) / 2;
```

[1] http://de.wikipedia.org/wiki/Gaußsche_Trapezformel

Listing 9: Flächenberechnung in der Klasse Area

Nachdem die einzelnen Flächen berechnet worden sind, wird jetzt die Gesamtfläche eines Bundeslandes berechnet. Listing 10 zeigt die Vorgehensweise. Dabei wird zuerst in jedem Flächen-Objekt nachgeschlagen, ob sich die Fläche in einer Fläche oder außerhalb liegt. Danach werden je nach Lage der Fläche, die Flächen addiert oder subtrahiert. Bis schlussendlich alle Flächen eines Bundeslandes behandelt worden sind und die Gesamtfläche feststeht.

```
for(; iterAreas != m_areas.end(); iterAreas++) {

  if(iterAreas->is_within_state() == true) {

    m_area -= iterAreas->get_area();
    os << " - " << iterAreas->get_area();
  }
  else {

    m_area += iterAreas->get_area();
    os << " + " << iterAreas->get_area();
  }

}
```

Listing 10: Berechnung der Fläche eines Bundeslandes in der Klasse State

3.1.3 Lokalisierung der Städte

Um vorgegebene Städte mit ihren Koordinaten einem Bundesland zuzuordnen, muss überprüft werden, ob sich die Koordinate in einem bestimmten Polygon befindet. Dies kann durch die Anzahl der Schnittpunkte bestimmt werden. Dabei wird ein Punkt außerhalb des Bundeslandes bestimmt und eine Strecke zu der gesuchten Stadt gezogen. Die Anzahl der Schnittpunkt, die die Strecke dabei verursacht gibt Aufschluss, ob sich der Punkt innerhalb oder außerhalb befindet. Ist die Anzahl der Schnittpunkte gerade befindet sich Stadt außerhalb, ansonsten innerhalb. Um einen Punkt außerhalb des Bundeslandes bestimmen zu können, kann die berechnete Bounding Box genutzt werden.

Das Listing 11 zeigt die Klasse `City`. Wird nun für eine Stadt das entsprechende Bundesland gesucht, werden alle Bundesländer nacheinander abgesucht.

```
for(; iterStates != a_states.end(); iterStates++) {

  cout << " Trying State: " << iterStates->second.get_name() << endl;
  bool result = iterStates->second.point_in_state(m_point);
  if(result == true) {

    cout << "State --> " << iterStates->first << endl;
```

```
    m_name_state = iterStates->first;
    iterStates->second.push_city_to_state(m_name_city, *this);
  }

}
```

Listing 11: Lokalisierung einer Stadt in der Klasse City

Dabei wird in der Klasse `State` (siehe Listing 12) zuerst durch die ermittelte Bounding Box eines Bundeslandes überprüft, ob sich der Punkt innerhalb der Bounding Box befindet. Wenn nicht kann die Stadt nicht in diesem Bundesland liegen, ansonsten muss es näher untersucht werden. Um eine nähere Untersuchung durchführen zu können, werden die Flächen eines Bundeslandes mittels einer Hilfsstrecke auf die Anzahl der Schnittpunkte überprüft. Dabei werden alle Strecken dieser Fläche mit der Hilfsstrecke auf Schnittpunkt überprüft und mitgezählt. Sind die Schnittpunkte aller Flächen ungerade befindet sich die Stadt innerhalb des Bundeslandes. Ist die Anzahl gerade liegt die Stadt nicht in diesem Bundesland und das nächste Bundesland kann untersucht werden.

```
bool result = point_in_bounding_box(a_point);
if(result == false) {

  cout << "   Point out of bounding box of State: " << m_name << endl;
  return false;
}

...

for(; iterAreas != m_areas.end(); iterAreas++) {

  int intersections = 0;
  iterAreas->point_in_area(a_point, &intersections);
  total += intersections;
}

if(total % 2 == 1)
  return true;
```

Listing 12: Lokalisierung einer Stadt in der Klasse State

3.2 Anwendung auf eine Testdatei

Um die beschriebene Funktionalitäten und Algorithmen zu testen, wurde eine Testdatei (data/Test.svg) erstellt. Hier wurden verschiedene Objekte gezeichnet um vor allem die Fläche in Fläche Berechnungen zu testen sowie die Lokalisierungen von vorgegebenen Punkten.

Das Listing 13 präsentiert das Ergebnis der Berechnungen. Das Ergebnis wird exemplarisch mit dem State *Dreieck*erläutert. Dabei wird die dazugehörige Bounding

Box, welche für die Hilfsstrecke benötigt wird, abgebildet. Die berechnete Fläche *Area* sowie die vorher ermittelte richtige Fläche *Exact* werden miteinander verglichen und die Differenz angezeigt. Die Stadt *PunktinDreieck* mit ihren Koordinaten wurde dabei im Bundesland *Dreieck* lokalisiert.

Am Schluss werden nochmals alle Bundesländer mit ihren Städten aufgelistet. Hierbei wird gezeigt, dass nicht alle Städte (hier: PunktInQuadrat und PunktAussen) einem Bundesland zugeordnet werden konnten. Diese werden dann mit *unknown* gekennzeichnet.

```
---- States and its calculated Areas -----
State:  Dreieck
  Bounding Box:          Min: X:       8   Y:       1  Max: X:      11   Y:       8
  Area:                 10.5
  Exact:                10.5
  Difference(%):           0
City: -->  PunktInDreieck    X:      10   Y:     1.5
State: -->  Dreieck
------------------------------------------
State:  ParallelOhneQuadrat
  Bounding Box:          Min: X:       1   Y:       1  Max: X:      11   Y:       9
  Area:                   39
  Exact:                  39
  Difference(%):           0
City: -->  PunktInParallel    X:       3  Y:       4
State: -->  ParallelOhneQuadrat
------------------------------------------
State:  Rechteck
  Bounding Box:          Min: X:       4   Y:       8  Max: X:     5.5   Y:       9
  Area:                  1.5
  Exact:                 1.5
  Difference(%):           0
City: -->  PunktInRechteck    X:       5  Y:     8.5
State: -->  Rechteck
------------------------------------------
------------------------------------------
------------ All Cities --------------
City: -->  PunktInQuadrat    X:       5   Y:       5
State: -->  Unknown
------------------------------------------
City: -->  PunktInParallel    X:       3  Y:       4
State: -->  ParallelOhneQuadrat
------------------------------------------
City: -->  PunktInRechteck    X:       5  Y:     8.5
State: -->  Rechteck
------------------------------------------
City: -->  PunktInDreieck    X:      10   Y:     1.5
State: -->  Dreieck
------------------------------------------
City: -->  PunktAussen     X:       1  Y:     8.5
State: -->  Unknown
------------------------------------------
------------------------------------------
```

Listing 13: Ergebnisse für die Testdatei

3.3 Präsentation der Ergebnisse

Nach dem alle Tests abgeschlossen und überprüft wurden, wird jetzt die eigentliche Datei (data/DeutschlandMitStaedten.svg) der Berechnungen unterzogen. Wie die Ergebnisse interpretiert werden, kann im Abschnitt 3.2 nachgelesen werden.

Die Abweichung der Flächen um ca. 15 Prozent von den berechneten zu den statistischen ermittelten Werten, lassen sich durch die Genauigkeit der Deutschlandkarte erklären.

```
---- States and its calculated Areas -----
State:  Baden-Wuerttemberg
  Bounding Box:           Min: X:  94.851  Y:  541.497 Max: X: 300.713  Y:   770.81
  Area:               30522.3
  Exact:              35751.7
  Difference(%):    -14.6269
City: -->  Stuttgart     X:     215  Y:     648
State: -->  Baden-Wuerttemberg
------------------------------------------
State:  Bayern
  Bounding Box:           Min: X:  200.02  Y:  463.246 Max: X: 527.364  Y:  800.258
  Area:               60026.1
  Exact:              70549.2
  Difference(%):    -14.9159
City: -->  Muenchen     X:  366.968  Y:     700
State: -->  Bayern
------------------------------------------
State:  Berlin
  Bounding Box:           Min: X: 460.803  Y:  240.679 Max: X: 501.897  Y:  272.113
  Area:               766.233
  Exact:               891.75
  Difference(%):    -14.0754
City: -->  Berlin     X:     477  Y:     256
State: -->  Berlin
------------------------------------------
State:  Brandenburg
  Bounding Box:           Min: X: 345.648  Y:  149.207 Max: X: 570.941  Y:   374.78
  Area:               25275.9
  Exact:              29477.2
  Difference(%):    -14.2525
City: -->  Potsdam     X:  458.763 Y:  260.757
State: -->  Brandenburg
------------------------------------------
State:  Bremen
  Bounding Box:           Min: X: 173.713  Y:   149.34 Max: X: 204.515  Y:  210.922
  Area:               340.931
  Exact:               404.23
  Difference(%):    -15.6591
City: -->  Bremen     X:  193.766  Y:   200.55
State: -->  Bremen
------------------------------------------
```

```
State:  Hamburg
 Bounding Box:          Min: X:  250.76  Y:  136.688 Max: X:  286.18  Y:  172.063
 Area:              633.325
 Exact:              755.16
 Difference(%):    -16.1336
City: -->  Hamburg     X:  265.845 Y:   156.03
State: -->  Hamburg
-----------------------------------------
State:  Hessen
 Bounding Box:          Min: X: 120.308  Y:  350.145 Max: X: 282.316  Y:  581.665
 Area:              17977.5
 Exact:             21114.7
 Difference(%):    -14.8578
City: -->  Wiesbaden     X:  148.823 Y:  508.371
State: -->  Hessen
-----------------------------------------
State:  Mecklenburg-Vorpommern
 Bounding Box:          Min: X:  303.89  Y:   34.735 Max: X: 538.295  Y:  199.904
 Area:              19658.8
 Exact:             23174.2
 Difference(%):    -15.1694
City: -->  Schwerin     X:  357.852  Y:  150.095
State: -->  Mecklenburg-Vorpommern
-----------------------------------------
State:  Niedersachsen
 Bounding Box:          Min: X:  60.544  Y:  120.329 Max: X: 366.113  Y:  387.501
 Area:              40633.5
 Exact:             47618.2
 Difference(%):    -14.6683
City: -->  Hannover     X:  253.549  Y:  273.477
State: -->  Niedersachsen
-----------------------------------------
State:  Nordrhein-Westfalen
 Bounding Box:          Min: X:   0.254  Y:  259.852 Max: X: 231.536  Y:  481.384
 Area:              28966.4
 Exact:             34083.5
 Difference(%):    -15.0135
City: -->  Duesseldorf     X:  56.8154 Y:  397.708
State: -->  Nordrhein-Westfalen
-----------------------------------------
State:  Rheinland-Pfalz
 Bounding Box:          Min: X:  11.071  Y:  421.405 Max: X: 167.465  Y:  624.789
 Area:              16913.6
 Exact:             19847.4
 Difference(%):    -14.7818
City: -->  Mainz     X:  148.399 Y:  523.635
State: -->  Rheinland-Pfalz
-----------------------------------------
State:  Saarland
 Bounding Box:          Min: X:  24.194  Y:  553.172 Max: X:  93.351  Y:  607.571
 Area:              2179.76
 Exact:             2568.65
 Difference(%):    -15.1397
City: -->  Saarbruecken    X:      60   Y:     589
State: -->  Saarland
-----------------------------------------
State:  Sachsen
```

```
  Bounding Box:          Min: X: 389.565  Y:  346.651 Max: X: 591.247  Y:  500.279
  Area:               15667.9
  Exact:              18413.9
  Difference(%):    -14.9126
City: -->  Dresden     X:  499.891 Y:  405.764
State: -->  Sachsen
------------------------------------------
State:  Sachsen-Anhalt
  Bounding Box:          Min: X: 302.718  Y:  207.164 Max: X: 470.115  Y:  421.815
  Area:               17450.5
  Exact:              20445.3
  Difference(%):    -14.6475
City: -->  Magdeburg     X:  370.996 Y:  294.677
State: -->  Sachsen-Anhalt
------------------------------------------
State:  Schleswig-Holstein
  Bounding Box:          Min: X: 139.262  Y:     0.25 Max: X: 345.952  Y:  175.469
  Area:               13456.4
  Exact:              15763.2
  Difference(%):    -14.6337
City: -->  Kiel     X:  275.173  Y:  78.4392
State: -->  Schleswig-Holstein
------------------------------------------
State:  Thueringen
  Bounding Box:          Min: X: 259.291  Y:  351.324 Max: X: 439.172  Y:   498.94
  Area:               13724.6
  Exact:              16172.1
  Difference(%):    -15.1341
City: -->  Erfurt     X:  335.381  Y:  418.908
State: -->  Thueringen
------------------------------------------
------------------------------------------
------------ All Cities --------------
City: -->  Muenchen     X:  366.968  Y:     700
State: -->  Bayern
------------------------------------------
City: -->  Berlin     X:     477   Y:     256
State: -->  Berlin
------------------------------------------
City: -->  Stuttgart     X:     215  Y:     648
State: -->  Baden-Wuerttemberg
------------------------------------------
City: -->  Saarbruecken    X:      60   Y:     589
State: -->  Saarland
------------------------------------------
City: -->  Wiesbaden     X:  148.823 Y:  508.371
State: -->  Hessen
------------------------------------------
City: -->  Mainz     X:  148.399 Y:  523.635
State: -->  Rheinland-Pfalz
------------------------------------------
City: -->  Duesseldorf     X:  56.8154 Y:  397.708
State: -->  Nordrhein-Westfalen
------------------------------------------
City: -->  Bremen     X:  193.766  Y:   200.55
State: -->  Bremen
------------------------------------------
```

```
City: -->  Erfurt     X:  335.381  Y:  418.908
State: -->  Thueringen
------------------------------------------
City: -->  Dresden     X:  499.891 Y:  405.764
State: -->  Sachsen
------------------------------------------
City: -->  Magdeburg     X:  370.996 Y:  294.677
State: -->  Sachsen-Anhalt
------------------------------------------
City: -->  Hannover     X:  253.549  Y:  273.477
State: -->  Niedersachsen
------------------------------------------
City: -->  Hamburg     X:  265.845 Y:   156.03
State: -->  Hamburg
------------------------------------------
City: -->  Kiel     X:  275.173  Y:  78.4392
State: -->  Schleswig-Holstein
------------------------------------------
City: -->  Schwerin     X:  357.852  Y:  150.095
State: -->  Mecklenburg-Vorpommern
------------------------------------------
City: -->  Potsdam     X:  458.763 Y:  260.757
State: -->  Brandenburg
------------------------------------------
------------------------------------------
```

Listing 14: Ergebnisse für die Deutschlandkarte

4 Aufgabe 3 - Line Sweep

Der Line Sweep Algorithmus ist ein Versuch die Schnittpunkt suche zu beschleunigen. Die Laufzeit wird nun outputsensitiv, also abhängig von der Anzahl der gefundenen Schnittpunkte. Es wird eine fiktive vertikale Linie erzeugt, die sich Punkt für Punkt durch die Linien arbeitet. Bei der Art wird zwischen Anfangs-, End- und Schnittpunkt unterschieden, je nach Art des Punktes werden eine andere Aktionen durchgeführt. Schnittpunkte können immer nur zwischen benachbarten Segmenten auftreten. Der Algorithmus terminiert, wenn alls Punkte bzw. Events bearbeitet wurden.

Es gibt beim Line Sweep allerdings einige Funktionseinschränkungen, die die Gleichzeitigkeit von Ereignissen betreffen. Diese beherrscht er nur mit sehr aufwendig zu implementierenden Mechanismen. Um die Aufgabe nicht unnötig komplexer zu gestalten werden diese nicht eingebaut und darauf geachtet folgende Regeln einzuhalten:

- X-Koordinaten von Schnitt- und Endpunkten sind paarweise verschieden
- Länge der Segmente >0
- nur echte Schnittpunkte
- keine Linien parallel zur Y-Achse
- keine Mehrfachschnittpunkte
- keine überlappenden Segmente

4.1 Initialisierung

Die Sweep Line besteht aus mehreren Queues, die den aktuellen und zukünftigen Zustand verwalten. Die erste Queue ist die Eventqueue, hier werden die Punkte abgelegt und mit der Information versehen, auch welcher Linie sie liegen. Bei Schnittpunkten werden die beiden Linien vermerkt, die sich dort schneiden.

```
class Event {
private:
  MyEventtype m_type;
  Point m_punkt;
  Line* m_seg;
  Line* m_seg2; //falls schnittpunkt
...
```

Listing 15: Attribute der Klasse Event

In der zweiten Liste werden die Segmente verwaltet, die sich aktuell mit der Seep Line schneiden. An einem Startpunkt wird die damit verbundene Linie an der richtigen Stelle in diese Queue eingefügt und auf Schnittpunkte mit ihren Nachbarn überprüft. Erreicht die Sweep Line einen Endpunkt, wird die entsprechende Linie entfernt und die neuen Nachbarn auf einen Schnittpunkt getestet. Wird ein Schnittpunkt behandelt, müssen die beiden Segmente auf der dieser Schnittpunkt liegt die Positionen tauschen. Gefundene Schnittpunkte werden in die Output Liste und als neues Event in die Eventqueue eingefügt. Hier ist besonders darauf zu achten, dass die Schnittpunkte an der richtigen Position einsortiert werden.

```
class Sweep {
private:
  list<Event> eventqueue;
  list<Line> segmentqueue;
  Event *ereignis;
  list<Event> output;
...
```

Listing 16: Attribute der Klasse Sweep

Um die Eventqueue zu initialisieren wurde eine neue Funktion in die Klasse LineFile eingefügt. Diese fügt alls Punkte ein und definiert ob es sich um einen Start- oder Endpunkt handelt. Anschließend wird die Liste durch eine in der STL enthaltende Funktion nach der X-Koordinate sortiert.

```
void LineFile::sweepiniteventqueue(){
  for(unsigned int i = 0; i < m_lines.size(); i++) {
    m_sweep.addevent(m_lines[i]->getstart(),m_lines[i]);
    m_sweep.addevent(m_lines[i]->getend(), m_lines[i]);
  }

}
...
void Sweep::addevent(Point* a_point, Line* a_line){
  //a_point ist startpunkt
  if ( a_point == a_line->getstart() ){
    eventqueue.push_front( Event(a_point, a_line, STARTPUNKT) );
  }
  //a_point ist endpunkt
  else {
    eventqueue.push_front( Event(a_point, a_line, ENDPUNKT) );
  }
  print_eventqueue();
}
```

Listing 17: Initialisierung der Sweep Line

4.2 Behandlung von Events

Da die Eventqueue immer nach aufsteigenden X-Koordinaten sortiert sein muss, kann der aktuelle X-Wert der Sweep Line durch die X-Koordinate des ersten Elements in der Eventqueue bestimmt werden. Wird die Sweep gestartet, wählt sie das erste Event aus der Queue und führt je nach Art des Punktes die entsprechende Aktion aus. Anschließend wird das Event aus der Queue entfernt und wieder das erste ausgewählt. Dieser Vorgang wir solange wiederholt, bis keine Events mehr in der Eventqueue enthalten sind. Dann können die Schnittpunkte aus der Outputqueue ausgelesen werden. Da der Vorgang des Auswählens und Löschen von Events bei allen Punkten gleich ist, wird dieser im Folgenden nicht weiter erwähnt.

```
void Sweep::calcinters(){
bool test = false;
list<Event>::iterator neweve;

  eventqueue.sort();
  while( (test = eventqueue.empty()) == false) {
    neweve = eventqueue.begin();
    ereignis = &(*neweve);

    switch(ereignis->gettype()){
      case STARTPUNKT:
        leftendpoint( );
        break;

      case ENDPUNKT:
        rightendpoint( );
        break;

      case INTERSECTION:
        treatintersection( );
        break;

      default:
        exit(1);
        break;
    }
    delevent(ereignis);
  }
}
```

Listing 18: Schleife der Sweep Line

4.2.1 Startpunkte

Wie schon kurz beschrieben, wird bei der Behandlung eines Startpunktes das neue Segment in die Segmentqueue eingefügt. Es ist wichtig, dass es an der richtigen Stelle eingefügt wird, da nur zwischen Nachbarn ein Schnittpunkt liegen kann. Ist die Nachbarschaft nicht korrekt sortiert, kommt es zu schwer nachvollziehbaren Fehlern.

```
Line* Sweep::addseg(Line* a_seg) {
  list<Line>::iterator newseg;

  if ( segmentqueue.empty() ) {
    segmentqueue.push_front(*a_seg);
    return a_seg;
  }
  else {
    for ( newseg = segmentqueue.begin(); newseg != segmentqueue.end(); ++newseg ) {
      if ( newseg->get_yvalue(getxposition()) > a_seg->get_yvalue(getxposition()) ) {
        segmentqueue.insert (newseg, *a_seg);
        return a_seg;
      }
    }
  }
  segmentqueue.insert (newseg, *a_seg);
  segmentqueue.unique();
  return a_seg;
}
```

Listing 19: Einfügen von Segmenten in die Sweep Line

Anschließend werden die beiden Nachbarn des gerade eingefügten Segments gesucht. Falls diese existieren, wird wie in Aufgabe 1 getestet ob ein Schnittpunkt zwischen dem Segment und seinem Nachbarn existiert. Existiert ein Schnittpunkt, werden erst anschließend die genauen Koordinaten bestimmt, dadurch soll die Laufzeit optimiert werden. Für den berechneten Schnittpunkt wird am Ende ein Event erzeugt, das sowohl in die Outputqueue als auch in die Eventqueue einsortiert wird. Dieser Vorgang wird anschließend für den zweiten Nachbarn wiederholt, soweit dieser existiert.

```
...
if ( neighbourA != NULL ) {
    if( aktseg->is_intersection_max(neighbourA) == true ) {
      interp1 = aktseg->intersectionpoint(neighbourA);
      Event Inter1(interp1,aktseg,neighbourA);
      addinter( Inter1 );
      addevent( Inter1 );
    }
  }
...
```

Listing 20: Schnittpunktprüfug bei Startpunkten

4.2.2 Endpunkte

Handelt es sich beim aktuell behandelten Event um einen Endpunkt, muss das entsprechende Segment aus der Queue entfernt werden und die anschließend neuen

Nachbarsegmente auf Schnittpunkte überprüft werden. Um diesen Vorgang durchführen zu können, muss man zuerst die Nachbarn der Linie bestimmen. Das suchen von Elementen wird immer mit der STL-Funktion find() durchgeführt. Nun kann das Segment entfernt werden. Existieren beide Nachbarn, kann versucht werden den Schnittpunkt zu bestimmen. Falls ein Schnittpunkt zwischen den Nachbarn besteht, wird dieser wie beim Startpunkt beschrieben in die Output- und Eventqueue eingefügt. Man muss nun allerdings darauf achten, dass man diesen Schnittpunkt möglicherweise schon berechnet hat. Es dürfen also nur Punkte in die Eventqueue eingefügt werden, deren X-Koordinate größer ist als die aktuelle Position der Sweep Line.

```
void Sweep::rightendpoint(){
  Line *segA=NULL, *segB=NULL;
  Point interp1;

  segA = getneighbour_high(ereignis->get_line());
  segB = getneighbour_low(ereignis->get_line());

  delseg(ereignis->get_line());
  if ( segA != NULL && segB != NULL) {
    if ( segA->is_intersection_max(segB) == true ) {
      interp1 = segA->intersectionpoint(segB);
      Event Inter1(interp1, segB, segA);
      if(Inter1.get_x() > getxposition()) {
        addevent( Inter1 );
      }
      addinter( Inter1 );
    }
  }
}
```

Listing 21: Behandlung von Endpunkten

4.2.3 Schnittpunkte

Die Behandlung von Schnittpunkten klingt vorerst sehr einfach ist allerdings relativ komplex umzusetzen. Es müssen die Segmente, die sich am Schnittpunkt schneiden, in der Segmentqueue vertauscht werden und anschließend je mit ihrem neuen Nachbarn auf einen Schnittpunkt überprüft werden. Hierfür müssen zunächst beide Segmente in der Queue gefunden werden und man muss spezifizieren welche Line in der Queue einen früheren Platz belegt. Nun werden die beiden Nachbarn des Schnittpärchens bestimmt. Das Vertauschen der beiden Schnittsegmente übernimmt nun die STL-Funktion iter_swap$iterator, iterator$. Nun werden die beiden Segmente je mit ihrem neuen Nachbarsegment auf einen Schnittpunkt geprüft. Existiert ein Schnittpunkt, wird dieser, wie aus den anderen Events bekannt, als Event in Eventqueue und Outputqueue eingefügt. Auch hier gilt die Voraussetzung, dass nur Events eingefügt werden dürfen, deren X-Koordinate größer ist als die aktuelle Position der Sweep Line.

```
void Sweep::treatintersection(){
  Line *segA=ereignis->get_line(), *segB=ereignis->get_line2(), *neighbourA=NULL, *neighbourB
      =NULL;
  Point interp1, interp2;
  list<Line>::iterator it_A = find(segmentqueue.begin(),segmentqueue.end(), *segA);
  list<Line>::iterator it_B = find(segmentqueue.begin(),segmentqueue.end(),*segB);

  if ( *it_B == *(++it_A) ){
    neighbourA = getneighbour_high(segB);
    neighbourB = getneighbour_low(segA);
    //Swap Elements
    //segB vor segA einordnen und altes Element segB löschen
    iter_swap(--it_A, it_B);

  }
  else if ( *it_B == *(--(--it_A)) ) { //iterator wurde in if() verändert
    neighbourA = getneighbour_low(segB);
    neighbourB = getneighbour_high(segA);
    iter_swap(++it_A, it_B);

  }

  if ( neighbourA != NULL ) {
      if ( segA->is_intersection_max(neighbourA) == true ) {
        interp1 = segA->intersectionpoint(neighbourA);
        Event Inter1(interp1, neighbourA, segA);
        if(Inter1.get_x() > getxposition()) {
          addevent( Inter1 );
        }
        addinter( Inter1 );
      }
    }
  if ( neighbourB != NULL ) {
      if ( segB->is_intersection_max(neighbourB) == true ) {
        interp2 = segB->intersectionpoint(neighbourB);
        Event Inter2(interp2, neighbourB, segB);
        if(Inter2.get_x() > getxposition()) {
          addevent( Inter2 );
        }
        addinter( Inter2 );
      }
    }
}
```

Listing 22: Behandlung von Schnittpunkten

4.3 Ergebnisse des Line Sweep

Um den Algorithmus zu testen, wurde eine Datei mit sehr einfachen Beispieldaten erstellt. Dabei handelt es sich um drei Strecken, die sich in insgesamt drei Punkten schneiden. Das Besipiel ist in Abbildung 4 dargestellt und wurde gewählt, da der Ablauf dadurch sehr leich nachvollziehbar ist.

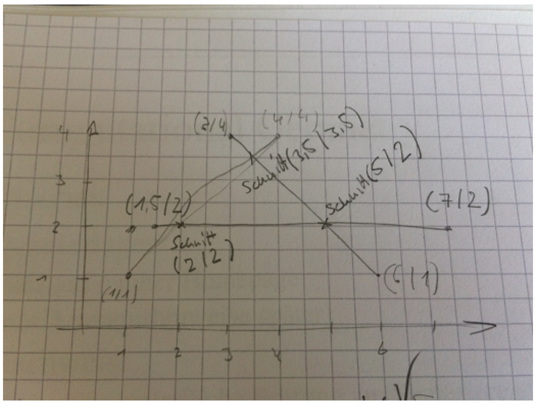

Abbildung 4: einfache Datensätze zum Test des Line Sweep

Das Ergebnis ist mit drei Schnittpunkten korrekt, wie in Listing 23 zu sehen ist sind allerdings die errechneten Schnittpunkte falsch. Zurückzuführen ist der Fehler auf die implementierung der *(*get_neighbour_high() und get_neighbour_low()) Funktion. Diese geben ungültige Werte zurück, wenn es sich bei der als Parameter übergebenen Linie um das obere bzw. untere Ende der Liste handelt. Für diesen Fall ist zwar eigentlich eine Abfrage vorgesehen, diese erfüllt allerdings nicht ihre Funktion. Wie die Fehlerkorrektur aussehen müsste, konnte nicht herausgefunden werden.

```
Event[P(1,1)-T1],
Event[P(1,1)-T1],
Event[P(4,4)-T2], Event[P(1,1)-T1],
Event[P(4,4)-T2], Event[P(1,1)-T1],
Event[P(3,4)-T1], Event[P(4,4)-T2], Event[P(1,1)-T1],
Event[P(3,4)-T1], Event[P(4,4)-T2], Event[P(1,1)-T1],
Event[P(6,1)-T2], Event[P(3,4)-T1], Event[P(4,4)-T2], Event[P(1,1)-T1],
Event[P(6,1)-T2], Event[P(3,4)-T1], Event[P(4,4)-T2], Event[P(1,1)-T1],
Event[P(1.5,2)-T1], Event[P(6,1)-T2], Event[P(3,4)-T1], Event[P(4,4)-T2], Event[P(1,1)-T1],
Event[P(1.5,2)-T1], Event[P(6,1)-T2], Event[P(3,4)-T1], Event[P(4,4)-T2], Event[P(1,1)-T1],
Event[P(7,2)-T2], Event[P(1.5,2)-T1], Event[P(6,1)-T2], Event[P(3,4)-T1], Event[P(4,4)-T2],
    Event[P(1,1)-T1],
Event[P(7,2)-T2], Event[P(1.5,2)-T1], Event[P(6,1)-T2], Event[P(3,4)-T1], Event[P(4,4)-T2],
    Event[P(1,1)-T1],
Event[P(1,1)-T1], Event[P(1.5,2)-T1], Event[P(3,4)-T1], Event[P(4,4)-T2], Event[P(6,1)-T2],
    Event[P(7,2)-T2],
Event[P(1.5,2)-T1], Event[P(3,4)-T1], Event[P(4,4)-T2], Event[P(6,1)-T2], Event[P(7,2)-T2],
Segment[P1(1/1), P2(4/4)],
Segment[P1(1/1), P2(4/4)],
Segment[P1(1/1), P2(4/4)], Segment[P1(1.5/2), P2(7/2)],
Event[P(1.5,2)-T1], Event[P(3,4)-T1], Event[P(4,4)-T2], Event[P(6,1)-T2], Event[P(7,2)-T2],
Event[P(1.5,2)-T1], Event[P(2,2)-T3], Event[P(3,4)-T1], Event[P(4,4)-T2], Event[P(6,1)-T2],
    Event[P(7,2)-T2],
Event[P(2,2)-T3], Event[P(3,4)-T1], Event[P(4,4)-T2], Event[P(6,1)-T2], Event[P(7,2)-T2],
Segment[P1(1/1), P2(4/4)], Segment[P1(1.5/2), P2(7/2)],
Schnitt(2,2),
```

```
Event[P(3,4)-T1], Event[P(4,4)-T2], Event[P(6,1)-T2], Event[P(7,2)-T2],
Segment[P1(1.5/2), P2(7/2)], Segment[P1(1/1), P2(4/4)],
Schnitt(2,2),
Segment[P1(1.5/2), P2(7/2)], Segment[P1(1/1), P2(4/4)],
Segment[P1(1.5/2), P2(7/2)], Segment[P1(1/1), P2(4/4)], Segment[P1(3/4), P2(6/1)],
Event[P(4,4)-T2], Event[P(6,1)-T2], Event[P(7,2)-T2],
Segment[P1(1.5/2), P2(7/2)], Segment[P1(1/1), P2(4/4)], Segment[P1(3/4), P2(6/1)],
Schnitt(1,1), Schnitt(2,2),
Event[P(6,1)-T2], Event[P(7,2)-T2],
Segment[P1(1.5/2), P2(7/2)], Segment[P1(3/4), P2(6/1)],
Schnitt(1.66667,2), Schnitt(1,1), Schnitt(2,2),
Event[P(7,2)-T2],
Segment[P1(1.5/2), P2(7/2)],
Schnitt(1.66667,2), Schnitt(1,1), Schnitt(2,2),
-------------------------------------
-----------Aufg 3 - Sweep------------
Intersected lines: 3
DeltaT: 0.015 seconds
-------------------------------------
```

Listing 23: Debugausgabe und Ergebnis des Line Sweep

5 Aufgabe 4 - Maximaler Kreis in einem konvexen Polygon

In dieser Aufgabe 4 werden für zwei vorgegebene konvexe Polygone der größtmögliche einbeschreibbare Kreis mit Linearer Programmierung berechnet. Die vorgegebenen Polygone sind in den Dateien *Polygon.txt* und *testpolygon.txt* abgespeichert. Nachdem das Problem beschrieben wurde, wird auf Basis der Testdatei *testpolygon.txt* ein lineares Programm definiert und hergeleitet. Dieses lineare Programm wird schlussendlich auf das Polygon in der Datei *Polygon.txt* angewendet und das Ergebnis dargestellt.

5.1 Herleitung Problem

Gegeben ist ein konvexes Polygon $P \subseteq \mathbb{R}^2$. Bei dem Problem in dieser Aufgabe suchen wir nach dem größtmöglichen Kreis $K \subseteq \mathbb{R}^2$, der vollständig in P enthalten ist; das heißt, es gilt $K \subseteq P$ wobei der Radius von K maximal ist. Damit der Kreis K mit Mittelpunkt $m := (m_x, m_y)$ und Radius r komplett in P enthalten ist, müssen folgende Bedingungen erfüllt sein:

- Der Punkt m hat mindestens den Abstand r von allen Strecken.
- Der Punkt m liegt innerhalb des konvexen Polygons P.

Um den Abstand r des Mittelpunktes m von einer Strecke P_i zu berechnen, wird die folgende Formel genutzt.

$$r = N_{0i} * [m - P_i]$$

Dabei beinhaltet die Matrix N_{0i} in der Zeile i den jeweiligen Normaleneinheitsvektor für die Strecke P_i.

Der Kreis K liegt also genau dann vollständig in P, wenn folgende Ungleichung für alle Strecken erfüllt ist.

$$N_{0i} * [m - P_i] \geq r, i = 1, ..., n$$

5.2 Lösung durch lineare Programmierung

Das Ziel ist es den größten Kreis zu finden. Wir definieren ein lineares Programm mit den Variablen m_x, m_y und r, indem die Variable r unter der Nebenbedingung

$$N_{0i} * [m - P_i] \geq r, i = 1, ..., n$$

maximiert wird. Die entsprechende Zielfunktion wird definiert als

$$f := \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

Zur Lösung des lineare Problems wird das Programm **MATLAB** (MATrix LABoratory) verwendet. Nachfolgend werden nur die wichtigsten Programmteile dargestellt und erklärt, dass komplette Programm **incircle.m** kann in der mitgelieferten Datei unter **matlab/incircle.m** begutachtet und getestet werden. Die jeweiligen konvexen Polygone werden durch das Programm als Matrizen in den **Workspace** importiert und stehen anschließend im **Command Window** zur Verfügung. Die Matrizen für die konvexen Polygonen P haben dabei folgende Strukturen:

$$P = \begin{bmatrix} 0 & 0 \\ 10 & 0 \\ 10 & 10 \\ 0 & 10 \end{bmatrix}$$

Die jeweiligen Zeilen der Matrix bilden den Startpunkt einer Strecke ab. Die Anzahl der Zeilen ist zugleich die Anzahl der zur Verfügung stehenden Strecken indem der Kreis eingebettet werden soll. Die Spalte 1 gibt die x-Werte und die Spalte 2 die y-Werte eines Startpunktes wieder. Um aus den einzelnen Punkten ein konvexes Polygon zu erhalten, werden die Startpunkte miteinander verbunden; also der Punkt in der Zeile wird mit dem Punkt in der Zeile 2 verbunden u. s. w. , der letzte Punkt (hier Zeile 4) wird mit dem Punkt in der Zeile 1 verbunden.

Das Listing 24 zeigt die Erstellung einer konvexen Hülle. Durch den MATLAB Befehl *convhulln(xy)* wird aus den vorgegebenen Strecken (xy) eine konvexe Hülle erzeugt und die entsprechenden Indizes zurückgegeben. Dadurch können dann in Zeilen 2 und 3 die jeweiligen Start- und Endpunkte einer Strecke bestimmt werden.

```
1  ecken = convhulln(xy);
2  A = xy(ecken(:,1),:);
3  B = xy(ecken(:,2),:);
```

Listing 24: Erstellen einer konvexen Hülle mit MATLAB

Nachdem die Start- und Endpunkte für alle verfügbaren Strecken berechnet worden sind, werden für diese in den Zeilen 1 - 3 der dazugehörige Normaleneinheitsvektor berechnet. Um sicherzustellen, dass alle Normaleneinheitsvektoren zum Mittelpunkt m zeigen, findet in den Zeilen 3 - 6 eine Überprüfung statt. Hier werden aus allen Startpunkten der x- und y-Mittelwert berechnet und als sicherer Mittelpunkt M_0

abgespeichert. Ist bei der anschließenden Subtraktion ein negativer Normaleneinheitsvektor vorhanden, wird dieser um 180 Grad gedreht.

```
N_p = (B - A) * [0 1; -1 0];
Betrag = sqrt(sum(N_p.^2,2));
N_p = N_p./[Betrag, Betrag];
M_0 = mean(A,1);
index = sum(N_p.*bsxfun(@minus, M_0, A), 2) < 0;
N_p(index,:) = -N_p(index, :);
```

Listing 25: Berechnung des Normaleneinheitsvektor mit MATLAB

Nachdem alle erforderlichen Daten berechnet worden sind, wird nun unter Einhaltung der Nebenbedingung, dass Ungleichungssystem aufgestellt. Mit dem MATLAB Befehl **linprog** kann ein Ungleichungssystem in der Form

$$A * x \leq b$$

gelöst werden. Dieser Befehl berechnet das Minimum eines linearen Problems, da hier aber ein maximaler Radius r gesucht wird, muss die Zielfunktion f entsprechend angepasst werden. Soll ein Maximum anstatt eines Minimum berechnet werden, müssen alle Koeffizienten der Zielfunktion f negiert werden. Die angepasste Zielfunktion lautet

$$f := \begin{pmatrix} 0 \\ 0 \\ -1 \end{pmatrix}$$

Durch Umformung der Nebenbedingung

$$N_{0i} * [m - P_i] \geq r$$

erhält man das angepasste Ungleichungssystem

$$-N_0 * m + r \leq -N_0 * P$$

für das Programm MATLAB. Das Listing 26 berechnet in den Zeilen 1 und 2 jeweils die linke Seite sowie rechte Seite. In den Zeilen 3 und 4 wird die dazugehörige Zielfunktion definiert.

```
1 b = -sum(N_p.*A, 2);
2 A = [-N_p.*ones(strecken_nr, 2), ones(strecken_nr, 1)];
3 f = zeros(3, 1);
4 f(3) = -1;
```

Listing 26: Erstellung der Ungleichung mit MATLAB

Die erzeugten Parameter werden in der ersten Zeile dem Befehl *linprog* übergeben, dieses liefert uns dann in den entsprechenden Rückgabewerten die Lösung des linearen Problems zurück.

```
[result, fval, exitflag, output] = linprog(f, A, b);
C = result(1:2)';
R = result(3);
```

Listing 27: Starten der Berechnung mit MATLAB

5.3 Ergebnisse

Im Abschnitt 5.2 wurde das entwickelte MATLAB-Programm schrittweise vorgestellt und erklärt. Hier werden nun die entsprechenden Lösungen für die Dateien *Polygon.txt* und *testpolygon.txt* vorgestellt.

Wird das Polygon in der Datei *testpolygon.txt* dem MATLAB-Skript *incircle* übergeben, werden die Werte $m = (5.0, 5.0)$ für den Mittelpunkt und $r = 5.0$ für den Radius des Kreises berechnet. Die Abbildung 5 bildet das Ergebnis grafisch ab.

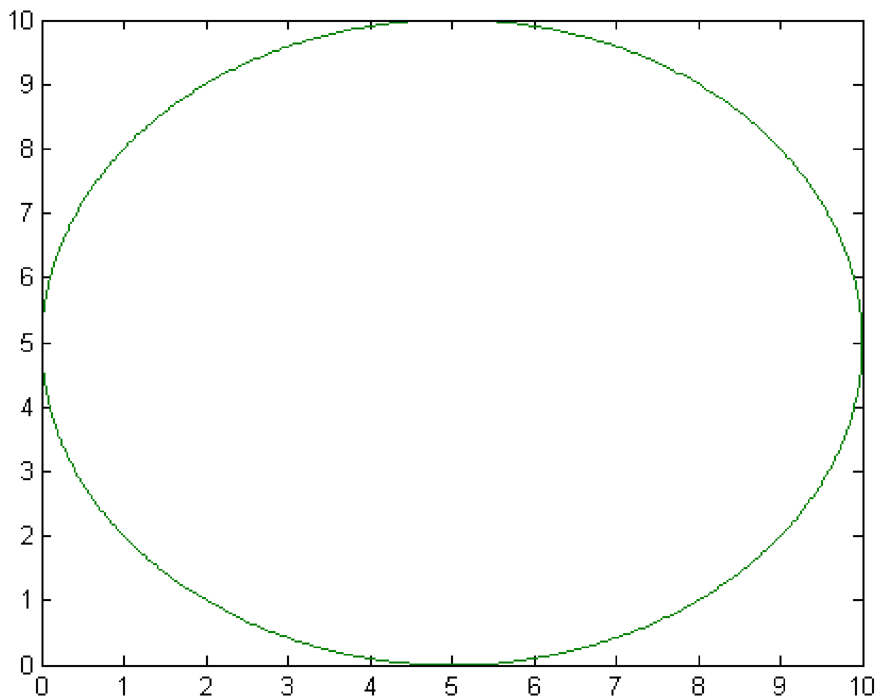

Abbildung 5: Testpolygon mit Kreis

Mit der Datei *testpolygon.txt* werden die Werte $m = (472.5705, 476.6642)$ für den Mittelpunkt und $r = 438.5922$ für den Radius des Kreises berechnet. Die entsprechende Grafik wird in der Abbildung 6 abgebildet. Das konvexe Polygon wird blau und der größtmögliche einbeschreibbare Kreis grün dargestellt.

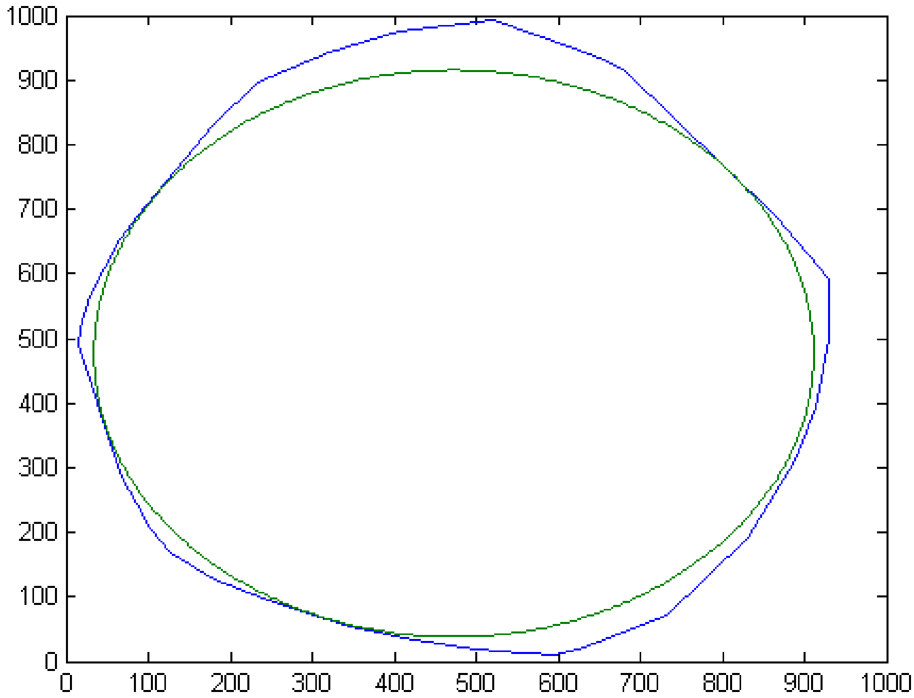

Abbildung 6: Polygon mit Kreis

6 Aufgabe 5 - Berechnung von konvexen Hüllen mit qhull

In dieser Aufgabe werden mit dem Programm *qhull*[2] zufällige Punktmengen erzeugt und mit diesen, eine konvexe Hülle in verschiedenen Dimensionen berechnet. Dabei wird zuerst ein Programm für die automatische Berechnung entwickelt um anschließend die Berechnungszeiten zu präsentieren.

6.1 Programm für die Berechnungen

Um für die verschiedenen Dimensionen und wechselnden Punkte die Berechnungszeiten zu ermitteln, wurde ein kleines Programm entwickelt. Das Programm nutzt die von *qhull* zur Verfügung gestellten C++ Schnittstellen und Bibliotheken. Die benötigten Bibliotheken können durch die jeweiligen Qt-Creator Projektdateien erzeugt werden. Die Projektdatei sowie die Berechnungsergebnisse sind im Ordner `aufgabe5` abgespeichert. Damit das Programm gestartet werden kann, muss es in dem *qhull* Ordner `src`
kopiert werden. Dadurch werden alle Abhängigkeiten für das Programm verfügbar gemacht.

6.2 Ergebnisse der Berechnungszeiten

In der Abbildung 7 werden die Berechnungszeiten für die Dimensionen 2 bis 5 in einem Koordinatensystem dargestellt. Dabei wurde eine maximale Punktemenge von 1.000.000 zufällig erzeugten Punkten genutzt. Gestartet wurde bei 0 und schrittweise um 100.000 Punkte bis zum Maximum erhöht. Die Punkte werden auf der x-Achse und die gemessenen Berechnungszeiten auf der y-Achse abgebildet. Die Dimensionen 2 und 3 werden in der obersten und die Dimensionen 4 und 5 in der untersten Grafik gezeigt.

Für die Dimensionen 6, 7 und 8 mussten die Punktemengen aufgrund von zu langen Berechnungszeiten und Arbeitsspeicher Problemen abgeändert werden. Die Berechnungszeiten werden in der Abbildung 8 in separaten Koordinatensystemen dargestellt. Dabei wurde eine maximale Punktemenge von 1000 Punkten mit einem Inkrementierungsschritt von 100 Punkten verwendet.

[2] http://www.qhull.org/

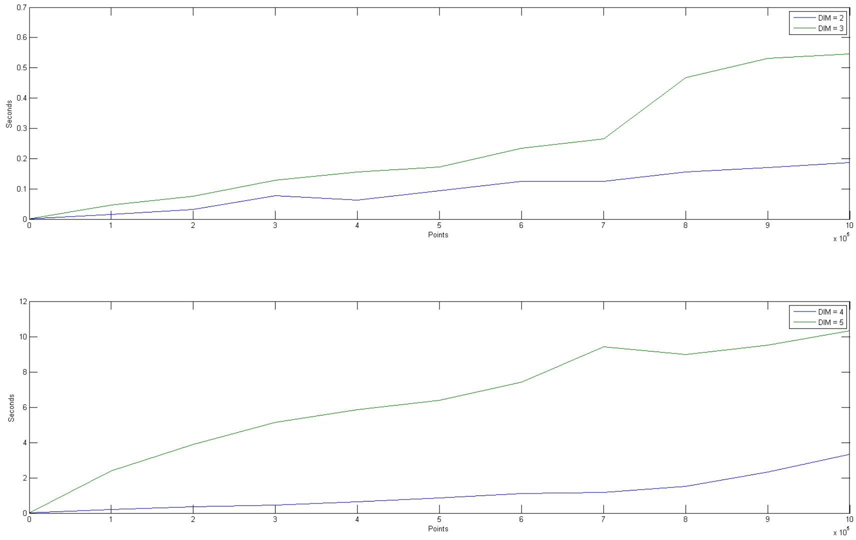

Abbildung 7: QHull Ergebnisse für die Dimensionen 2 - 5

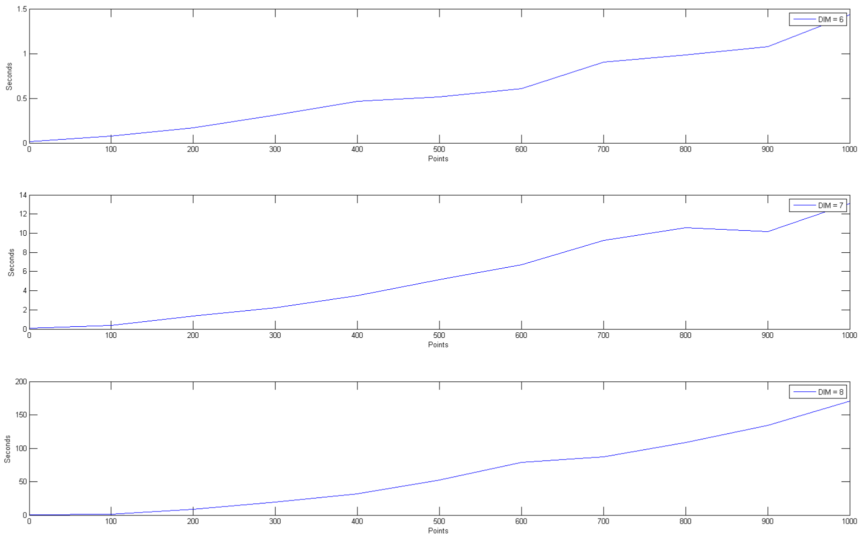

Abbildung 8: QHull Ergebnisse für die Dimensionen 6 - 8

7 Fazit

7.1 Zusammenfassung

Die Aufgaben konnten alle gelöst werden, allerdings erforderten gerade die ersten drei Aufgaben sehr viel Zeit. Insgesamt musste man teilweise sehr viel Zeit für Dinge aufwenden, die nicht direkt mit der eigentlichen Aufgabe zu tun haben. Beispielsweise war das parsen des Input Files in der zweiten Aufgabe extrem aufwendig, wenn man nicht Java verwendet. In Aufgabe 3 war die suche nach Funktionen der STL, die man verwenden kann und die auch funktionieren wie erwartet sehr aufwendig. Die eigentliche Aufgabe war dagegen weniger komplex.

Vermutlich hätte man sich viel Arbeit sparen können, wenn die Zusammenarbeit zwischen den Teams besser gewesen wäre. Man hätte dann gemeinsame Parser oder gleiche STL-Funktionen verwenden können. Tatsächlich wurden allerdings nur die Ergebnisse verglichen und teilweise die Algorithmen, allerdings erst nachdem man selbst eine Implementierung konstruiert hat.

Insgesamt hat das Praktikum sehr zum Verständnis und der Vertiefung des Vorlesungsstoff beigetragen. Durch die Aufgaben musste man sich mit den Problemen weiter auseinandersetzten und sich funktionierende Konzepte für die Realisierung überlegen. Selbst wenn die meisten Probleme weniger mit dem Stoff, als mit der Umsetzung in Programmcode zu tun hatten, ist der Rückblick trotzdem positiv.

7.2 Lessons Learned

Wir haben während des Praktikums gelernt, dass man durch Kooperation zu den besten Ergebnissen kommt. Durch die zusätzliche Kreativität und Inspiration, die man durch Diskussionen mit anderen Teams bekommt, gewinnt auch die Eigene Arbeit an Qualität.

Aufgabe 1 hat verdeutlicht, dass es nicht einen richtigen Algorithmus gibt. Wir haben es geschafft in einem Team 2 verschiedene Realisierungen zu Implementieren. Beide Ideen haben ihre Vor- und Nachteile und da wir uns nicht einigen konnten, welcher nun besser ist haben wir uns entschieden beide unabhängig zu verwenden.

Sowohl die Funktion des Line Sweep, als auch die STL-Funktionen zu Listen und Vectoren wurden in Aufgabe 3 verwendet und dadruch vertieft. In dieser Aufgabe konnte man erkennen, dass lineare Laufzeiten schlechter sein können als quadratische, da man die Art der Faktoren beachten muss. Bei Outputsensiven Algorithmen muss beachtet werden ob der Input einen sinnvollen Einsatz zulässt.