Pflichtenheft

Pflichtenheft Fussgänger Simulation FuSi (FuSi)

Version 5

Autor des Dokuments	vonop1, bohnp1, jaggr2	Erstellt am	20.09.2013
Dateiname	Pflichtenheft_FuSi.docx		
Seitenanzahl	18	© Team vonop1, bohnp1, jaggr2	

Historie der Dokumentversionen

Version	Datum	Autor	Änderungsgrund / Bemerkungen
1	20.09.2013	Team	Ersterstellung
2	27.10.2013	Team	Erweiterung um Brainstorming-Experimente
3	20.12.2013	Team	Beschrieb Detailkonzepte
4	10.01.2013	Team	Beschrieb Detailkonzepte
5	17.01.2013	Team	Diverse Detailkorrekturen

Inhaltsverzeichnis

1 Einleitung

1.1 Allgemeines

1.1.1 Zweck und Ziel dieses Dokuments

Dieses Pflichtenheft beschreibt die Rahmenbedingungen für die geplante Fussgänger Simulation.

1.1.2 Ausgangslage

Bei der Bewegung von Fussgängermengen entstehen viele interessante Phänomene von Selbst-Organistation. Das Studium dieser Phänomene ist z.B. bei der Planung von Fluchtwegen in Stadien etc. von grosser Bedeutung.

Es soll eine Software erstellt werden, welche die Bewegung von Fussgängern bei Vorhandensein von Hindernissen und Wänden simuliert. Die Bewegungen sollen grafisch visualisiert werden und es sollen wichtige Kenngrössen berechnet werden.

2 Konzept und Rahmenbedingungen

2.1 Ziele

Es soll eine Simulation von Fussgängerströmen erstellt werden. Die Fussgänger werden in einer Ihnen bekannten Landschaft positioniert. Von dieser Position aus sollen sie sich zu einem Zielpunkt bewegen. Das Verhalten bei Wegkonflikten von mehreren Fussgänger oder auch das Herdenverhalten soll damit studiert werden können.

2.2 Benutzer / Zielgruppe

Verhaltensforscher mit erweiterten Programmierkenntnissen

2.3 Ressourcen

Projektablage: GitHub

Dokumentation: Microsoft Word, Structorizer

Entwicklungsumgebung: Idea Intellj

Java-Bibliotheken: Java 1.7, Java2D Graphics (Darstellung, Editor), XML DOM-Parser (Konfigurationsdatei)

2.4 Übersicht der Meilensteine

Vorbereitungsphase	
Installation Entwicklungsumgebung	27.09.2013
Freigabe Pflichtenheft	27.09.2013
Implementierung und Test	
Anzeigen einer Landschaft aus einer Konfigurationsdatei	25.10.2013
Ein Fussgänger marschiert konfliktfrei vom Start- zum Zielpunkt	04.11.2013
Bearbeiten von Landschaften in einem Editor	29.11.2013
Mehrere Fussgänger marschieren Ihren Weg konfliktfrei vom Start bis ins Ziel	20.11.2013
Projektabschluss	
Reserve und Abschlusspräsentation	17.01.2013

2.5 Java Namenskonventionen

- Alle Namen sind in English und in CamelCase Notation
- Klassen haben ein C als Präfix und sind ein Nomen
- Interfaces haben ein I als Präfix und sind ein Nomen
- Get* und Is* Methoden sind immer readonly, verändern als nichts am Objekt

3 Grundkonzept

3.1 Aufbau der Simulationsumgebung (Welt)

Die Simulationsumgebung (nachfolgend bezeichnet als Welt) besteht aus einem kartesischen 2D Koordinatensystem. In dieser Welt gibt es 0 bis n Hindernisobjekte, dargestellt als Kreise, Rechtecke, Polygone, etc., sowie 1 bis n Personen, die alle einen Start- und Zielpunkt haben. Das Ziel der ganzen Simulation ist es, möglichst realitätsnah die Bewegungen der Fussgänger in der vorgegebenen Welt zu simulieren.

Zum Berechnen der Wege definieren wir folgende Grundbedingungen

- Jeder Fussgänger hat ein x-y-Ort und eine Richtung
- Jedes Hindernisobjekt stellt mit seinen äusseren Ecken einen Wegpunkt dar, anhand dem sich die Personen für Ihre Wegsuche orientieren
- Aus allen Wegpunkten generieren wir einen Graphen, bei dem keine Kante durch ein Hindernis führt.
- Mit Hilfe des Djikstra-Algorithmus errechnen wir für die Personen den kürzesten Weg durch den Graphen
- In Situationen, wo zwei Personen den gleichen Punkt betreten, wollen wir in einem ersten Schritt per Zufall entscheiden, wer gehen und wer warten muss.
- Die Simulation verfolgt das Prinzip eines Taktmodelles, in der jeder Takt z.B. eine Sekunde darstellt.

Softwaretechnisch wollen wir uns auf 2 Threads beschränken. Einer der Threads zeichnet mit Java2D das GUI, der andere führt die Simulation durch. Wir möchten mit dieser Entkoppelung erreichen, dass wir später nicht mehr zwingend jeden Schritt, sondern z.B. nur jeden dritten Schritt der Simulation zeichnen lassen können. Auf Grund einiger Tests wissen wir bereits, dass das Zeichnen mit vielen Objekten beachtliche CPU-Ressourcen in Anspruch nehmen kann und wir so eine weitere Möglichkeit haben, die Performance zu verbessern.

3.2 Ausbaustufen

Weiter wollen wir folgende Optimierungen implementieren respektive ausprobieren:

- Über alle Wegpunkte stellen wir einen gewichteten Graph als Basisgraph zusammen. Dieser Basisgraph soll die Performance verbessern, weil so nicht jedes Mal die komplette Welt neu berechnet werden muss, sondern das Person X in den Basisgraph einsteigen kann und ihn am Punkt, der ihrem Zielpunkt am nächsten kommt, ihn wieder verlässt.

- Bei der Berechnung, welche Personen sich in der Nähe von Person x befinden, entsteht ein Algorithmus der Komplexitätsklasse $O(n^2)$. Zur Optimieren legen wir ein virtuelles Viereckraster über die Welt, in das sich jede Person in ihre jeweilige Zelle einschreibt. Dadurch können wir direkt über die eigene und 9 benachbarten Zellen auslesen, welche Personen sich in der Nähe befinden.

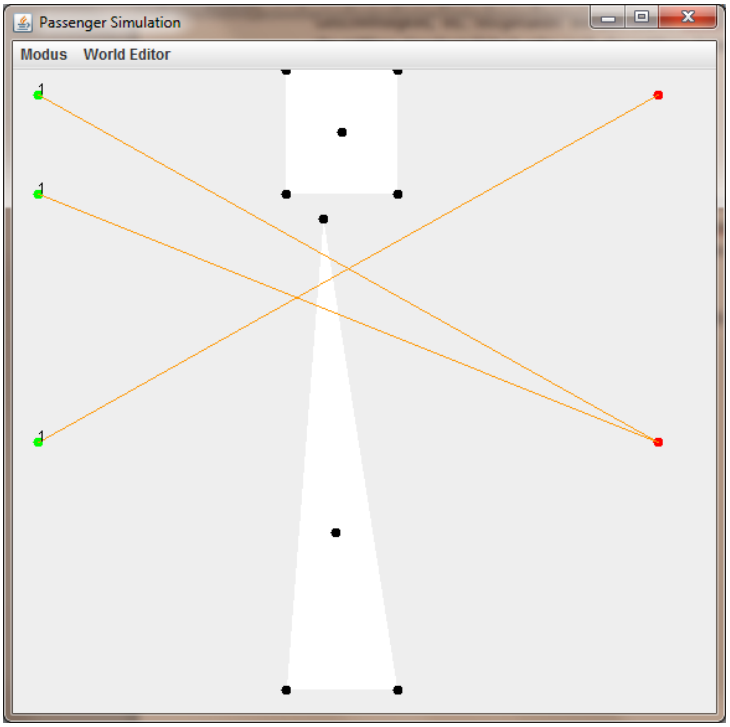
- Jede Personen soll über ein bestimmtes Bewegungsmodell verfügen. Damit wollen wir eine realitätsnahere Entscheidungsfindung umsetzen, als uns einfach den Zufall zu verlassen. Beispiele:
    - Der „Ellbögler": Versucht lieber die andere Person wegzudrängen als auszuweichen
    - Der „Zurückhaltende": Hält sich eher zurück und gibt anderen den Vortritt
    - Der „PolizistImEinsatz": Der Dauervortrittshaber
    - Der „Scheuche": Weicht immer aus, will niemandem zu nahe kommen.

- Für jede Personen sollen statistische Daten festgehalten werden. Damit liesse sich dann eine Statistik erstellen, die die Minimum Anzahl Schritte und dem gegenüber die effektiv benötigten Schritte der Personen darstellt.

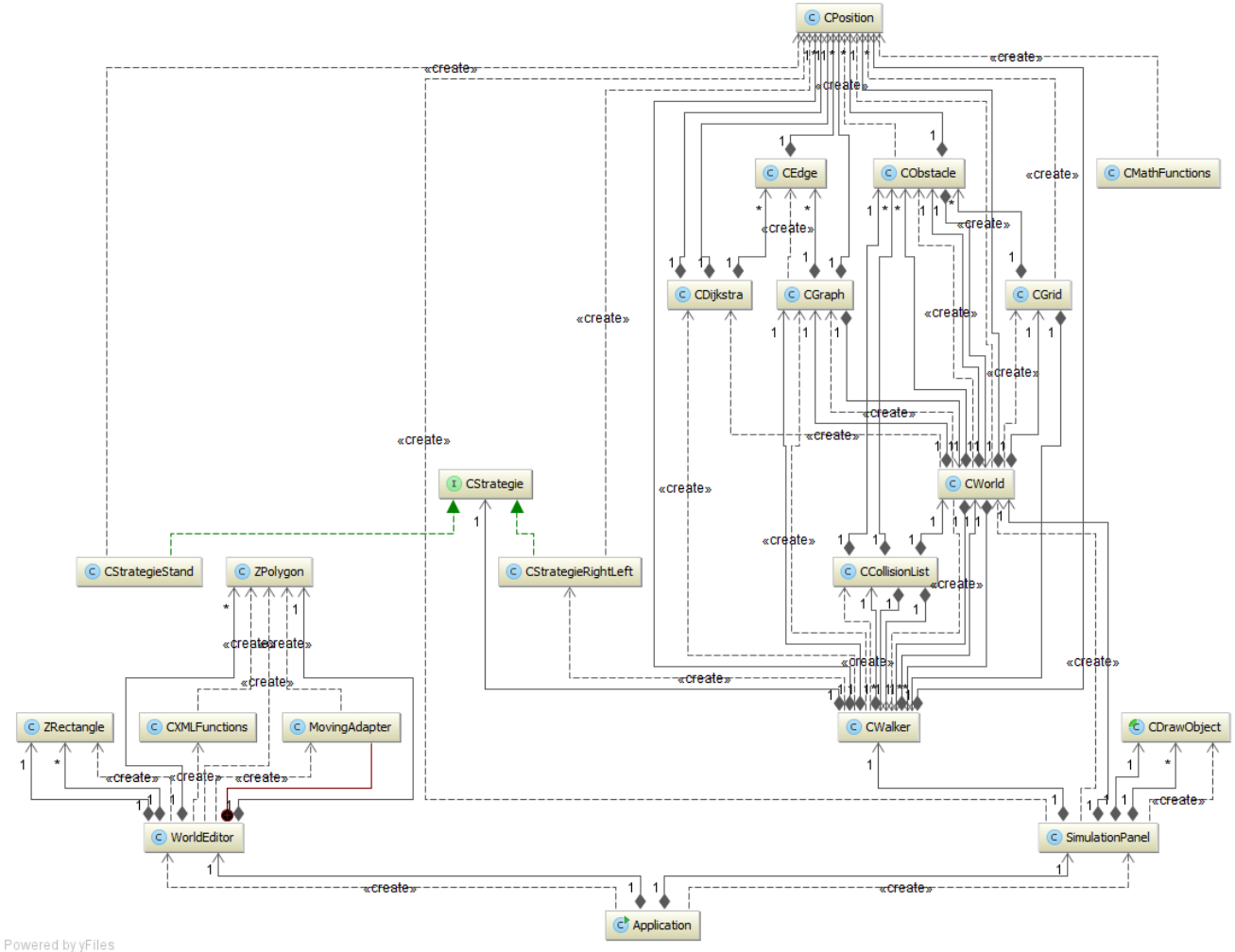
3.3 Der Landschaftseditor

Mit Hilfe des Landschaftseditors soll grafisch eine Welt zusammengestellt werden können. Das Ziel ist es, dass per Drag&Drop Objekte platziert und in Form und Grösse bearbeitet werden können. Ebenfalls sollen Walker mit Start- und Zielpunkt eingefügt werden können. Die erstelle Landschaft speichern wir in eine XML-Datei ab, die dann in der Simulation als Welt geladen werden kann.

Das zum Landschaftseditor passende XSD findet sich im Anhang. Ebenfalls befindet sich dort eine Beispiel-XML-Datei zum besseren Verständnis. Nach dem Laden dieser XML-Datei im Landschaftseditor präsentiert sich dem Anwender nachfolgende Anzeige:



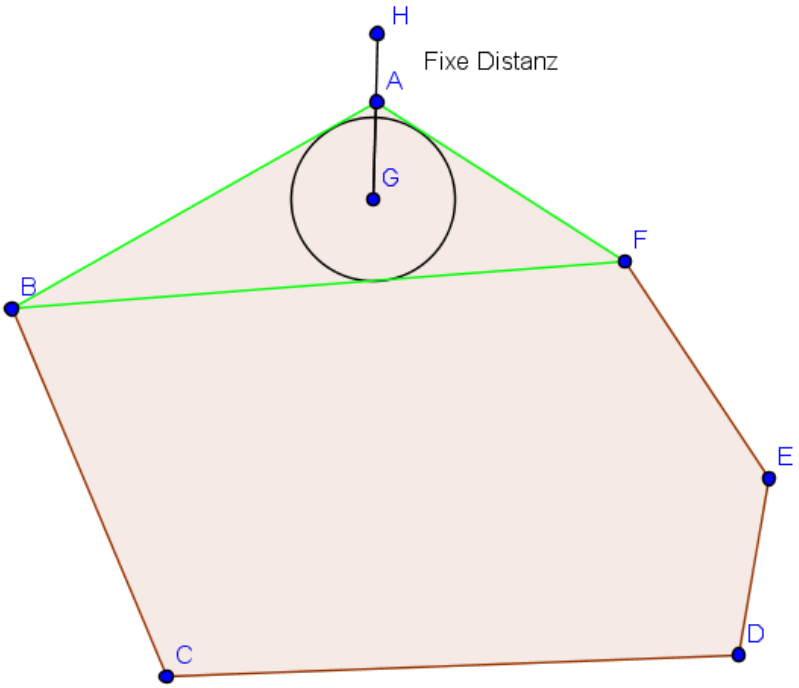
3.4 Domainmodell der Simulation



4 Detailkonzepte

4.1 Finden der Wegpunkte eines Hindernisses

Zum Finden eines Wegpunktes wird aus den zwei Stecken die ihm Punkten Enden ein Dreieck gebildet. Vom Inkreis-Mittelpunkt dieses Dreiecks wird zum Ecken des Hindernisses eine Strecke gezogen, diese wird dann um einen bestimmten Wert verlängert.



4.2 Finden des Weges vom Start zum Ziel eines Fussgängers

Zur Berechnung des Weges bauen wir über alle Wegpunkte aller Hindernisse einen ungerichteten gewichteten Graph auf. Der Graph ist ungerichtet, weil der Fussgänger die jeweiligen Graphkanten in beide Richtungen passieren können soll. Die Gewichtung des Graphs ist in der Fussgängersimulation immer die jeweilige Länge der Kante. Dieser Graph wird nachfolgend zum besseren Verständnis jeweils Grundgraph genannt.

Mit Hilfe dieses Grundgraphen wird nun für jeden Fussgänger folgende Berechnung gemacht:

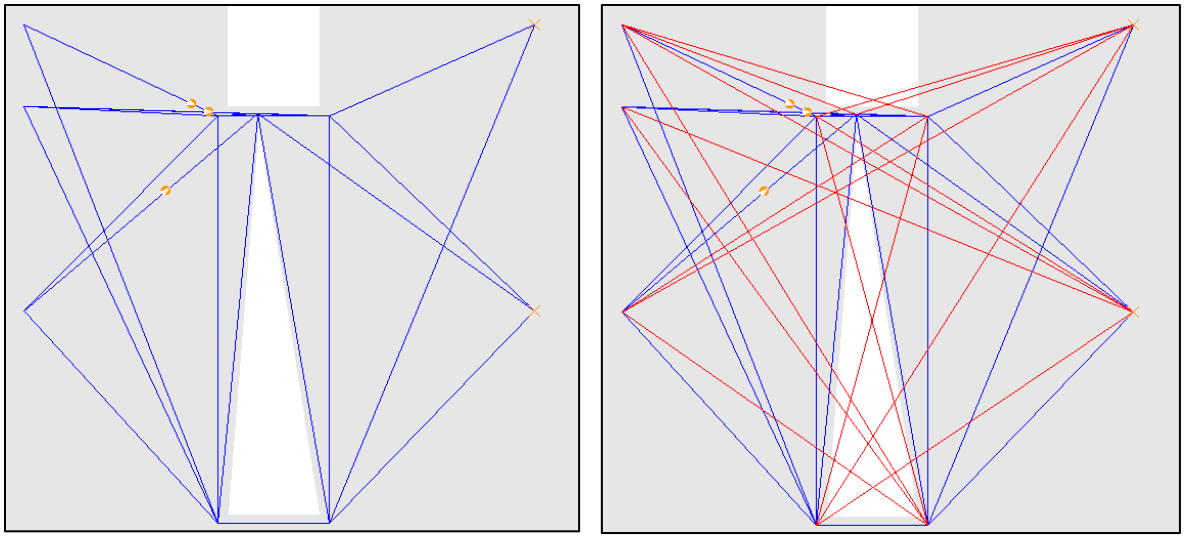
1. Einfügen des Start- und Zielpunktes in den Grundgraphen. Im Beispiel des Startpunktes wird mit diesem eine Kante zu allen vorhandenen Punkten des Grundgraphen eingefügt. Bei diesem Einfügen wird geprüft, ob sich die Kante mit einem Hindernis schneidet, und wird nur im überschneidungsfreien Fall effektiv in den Graph aufgenommen. Für den Zielpunkt passiert dasselbe.

2. Mit dem im ersten Schritt erweiterten Graphen wird nun der kürzeste Pfad berechnet. Für die Berechnung verwenden wir den Dijkstra-Algorithmus, der die Lösung des Problems mit einer Laufzeit von O(n * log(n)) ermitteln kann.

3. Der kürzeste Pfad wird anschliessend auf dem Fussgänger gespeichert. Der Fussgänger wird dann in der Simulation Punkt für Punkt dieses Pfades ablaufen.

Die Simulation ermöglicht ebenfalls die Darstellung aller aktiven Graphkanten. Diese ist mittels der Taste G ein- und ausschaltbar. Ebenfalls können die verworfen, weil mit einem Hindernis schneidenden, Kanten angezeigt werden. Die Taste hierfür ist T.

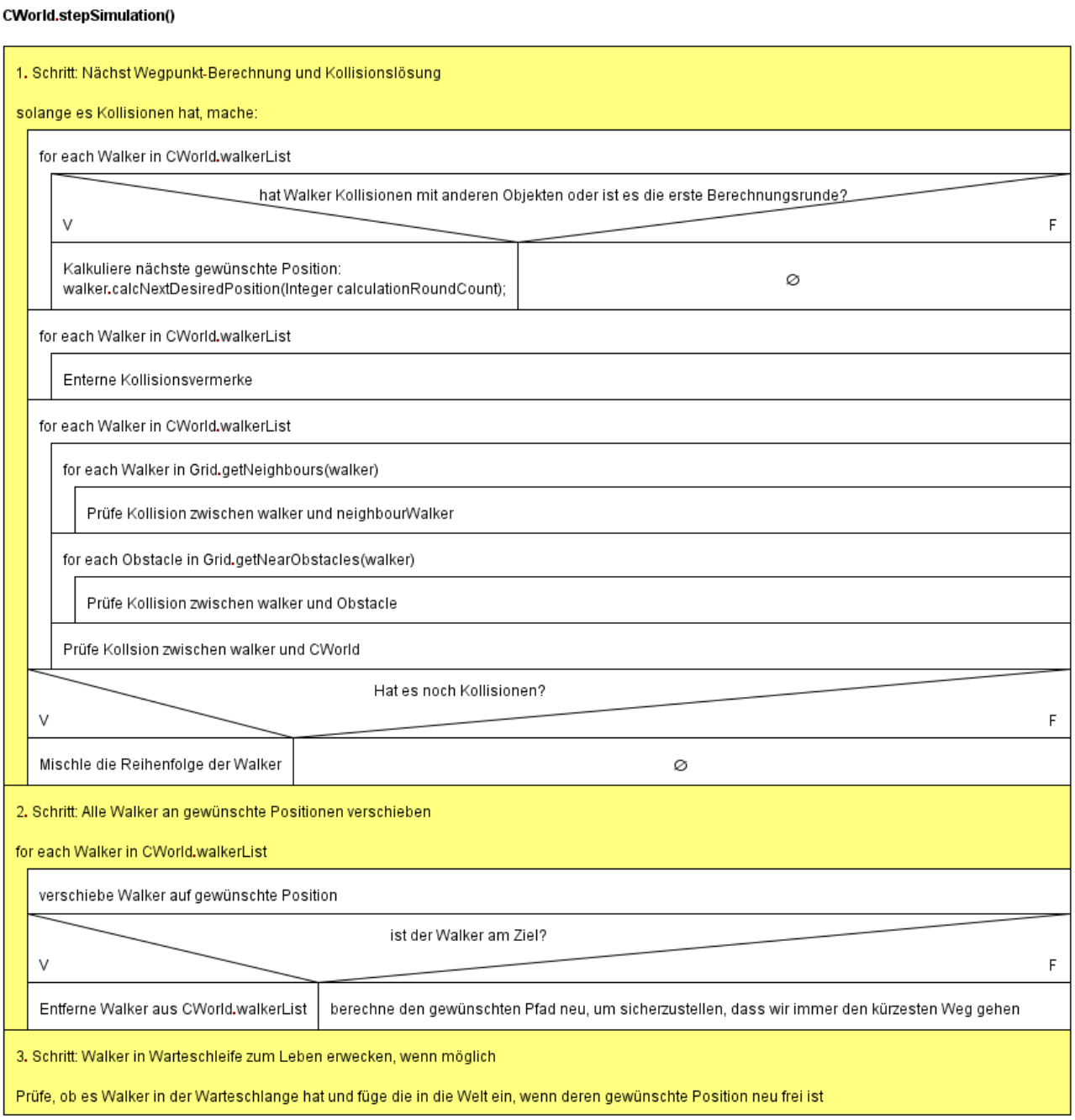
Im linken Screenshot sind nur die aktiven, blauen Kanten zu sehen, während im rechten Screenshot auch die verworfen, roten Kanten eingeblendet sind:



Diese Darstellung ist ebenfalls nützlich, um ein Verhalten des Fussgängers besser nachvollziehen zu können.

4.3 Berechnungsablauf der Simulation

Per GUI Timer wird jeweils die Berechnung eines Simulationsschrittes ausgelöst. Innerhalb des Simulationsschrittes findet dann nachfolgende Berechnung statt, die im Wesentlichen aus den drei gelb markierten Schritten besteht:



4.4 Nachbarschaftserkennung mittels Grid

Damit ein Fussgänger erkennen kann, wer in seiner unmittelbarer Nähe ist, müsste er theoretisch all Fussgänger und Hindernisse durchiterieren, um diese Frage zu beantworten. Dies würde aber eine Laufzeit von ungefähr $O(n^2)$ bedeuten, wenn jeder Fussgänger dies machen muss. Zur Optimierung dieses Verhaltens führen wir ein Grid über die ganze Welt ein, in das sich jedes Hindernis und jeder Fussgänger einschreiben muss, wenn er seine Position ändert. Damit wird es möglich, dass ein Fussgänger durch eine simple Berechnung direkt abfragen kann, welches seine Nachbarn sind.

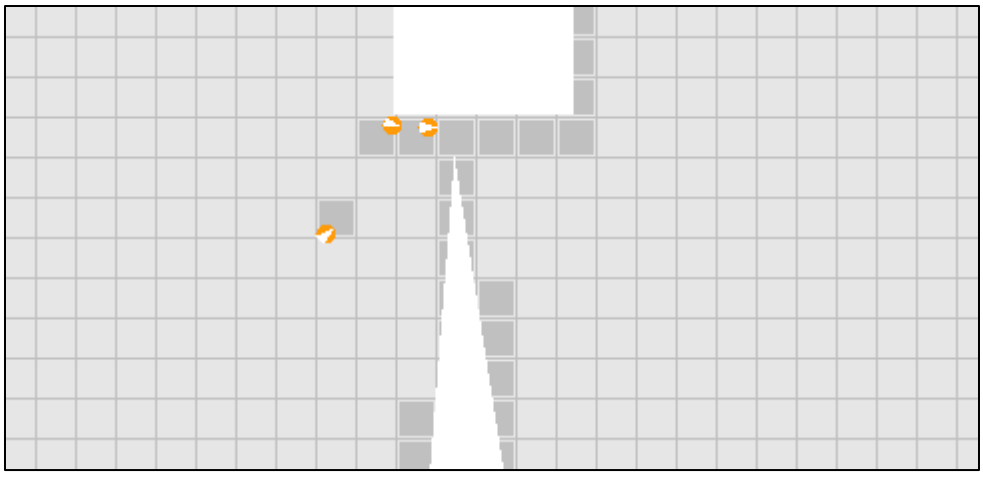
4.4.1 Berechnungsformel

GridSpalte = Y-Position / GridGrösse;

GridZeile = X-Position / GridGrösse;

4.4.2 Screenshot der Implementation

Die grauen Felder markieren die Grid-Zellen, in denen aktuell mindestens ein Fussgänger oder ein Hindernis eingeschrieben ist.



4.5 Kollisionen

4.5.1 Kollisionserkennung

In der Kollisionserkennung wird für jeden Fussgänger geprüft, ob seine gewünschte nächste Position sich mit der gewünschten nächsten Position eines anderen Fussgängers oder eines Hindernisses schneidet. Zur Optimierung dieser Berechnung kommt hier die vorher erläuterte Nachbarschaftserkennung zum Zug, sprich jeder Fussgänger prüft ausschliesslich mit seinen Nachbarn auf Kollisionen.

Zur Berechnung einer Kollision wird die Distanz zwischen den beiden betroffenen Punkten gerechnet und entsprechend berücksichtigt, dass Objekt A und Objekt B beide eine gewisse Grösse haben und einen genug grossen Radius um sich frei haben müssen. Zu diesen beiden Grössen berücksichtigen wir zusätzlich einen Minimalabstand von einem Punkt. Die verwendete Formel lautet:

```
hatKollision = PunktA.getDistanceTo(PunktB) < GrösseObjektA + GrösseObjektB + 1
```

4.5.2 Kollisionsliste

Wenn eine Kollision erkannt wird, wird dies in einer Kollisionsliste vermerkt. Jeder Fussgänger, jedes Objekt und auch die Welt selber hat eine Referenz zu dieser Kollisionsliste. Die Welt ist auch Kollisionskandidat, weil ein Fussgänger mit dem Rand der Welt kollidieren kann, den er natürlich nicht überschreiten darf. Die Referenz zur Liste wird jeweils während der Erkennung entsprechend bei allen betroffenen Beteiligten hinterlegt.

Die Idee der Kollisionsliste ist, in den Ausweichstrategien entsprechend intelligenter reagieren zu können. Ein Anwendungsszenario wäre zum Beispiel, dass wenn ein Fussgänger merkt, dass er mit Fussgängern und Hindernissen komplett umgeben ist, er entsprechend sein Ausweichverhalten auf „Stehen bleiben" ändern kann. Andernfalls könnte er ermitteln, ob es einen für ihn mehr oder weniger passenden Ausweg aus der Konfliktsituation gibt.

4.6 Ausweichstrategien eines Fussgängers

Jeder Fussgänger hat seine eigene Ausweichstrategie, die darüber entscheidet, was im Kollisionsfall passieren soll. Die Infos, die die Strategie dabei zur Verfügung hat, sind alle Eigenschaften des Walkers sowie in welcher Berechnungsrunde die Kollsionslösungsschleife ist. Ausserdem besteht ab der 2. Berechnungsrunde pro Walker eine Liste zur Verfügung, in der steht, mit wem alles der jeweilige Walker kollidiert.

Als Ausbaustufe ist vorgesehen, dass jeder Fussgänger dann mehrere Ausweichstrategien haben kann, wobei jede der Strategie entsprechend ihrer Gewichtung zum Zug kommt. Die Gewichtung soll pro Fussgänger dann individuell sein, damit wir eine möglichst natürliche Verhaltensweise abbilden können, zum Beispiel dass Person X im Blockierungsfall lieber Stehen bleibt als ausweicht.

Die bereits vorhandenen Strategien werden nachfolgend erläutert, damit besser verstanden werden kann, wieso sich die Fussgänger in der Simulation so verhalten, wie sie das tun.

4.6.1 Ausweichstrategie Warten

Wenn eine Kollision mit einem anderen Fussgänger oder Hindernis auftritt, wartet der entsprechende Fussgänger einfach, bis der gewünschte Weg wieder frei wird. Dieses Verhalten kann dazu führen, dass ein Fussgänger, gerade wenn er sich mit einem Hindernis kollidiert, einfach für immer Stehen bleibt. Da sich in jedem Berechnungszyklus aber das Verhalten ändern kann, wird er früher oder später ein anderes Verhalten annehmen, wodurch er sich wieder weiter bewegen kann. Ohne andere Ausweichstrategien führt dieses Verhalten aber grundsätzlich zu einer Blockade.

4.6.2 Ausweichstrategie LeftRight

Bei dieser Ausweichstrategie wird in erster Instanz versucht den Weg welcher über den Dijkstra berechnet wurde zu gehen. Ist ein anderer Fussgänger im Weg, so wird versucht mit einer möglichst kleinen Abweichung vom Weg der nächste Schritt zu machen. Die Richtung des nächsten Schritt wird mit folgender Formel berechnet (Winkel in Bogenmass):

$\varphi = \varphi_{opt}$ +/- $(\eta - 1) * 0.1$

wobei:

φ = Winkel bezogen auf X-Achse für nächsten Schritt.
φ_{opt} = Winkel wenn gewünschter Weg weiter möglich wäre.
+/- = Zufällig gewällt ob + oder -.
η = Nummer des Versuches einen Ausweichspunkt zu finden.

4.6.3 Weitere Ausweichstrategien implementieren

Die Ausweichstrategien sind alle vom Interface IStrategy vererbt. Weitere Strategien können dadurch einfach durch eine weitere Implementationsklasse hinzugefügt werden.

4.7 Darstellen, Steuern und Laden der Simulation

Die Simulation wird grösstenteils via Tastatur gesteuert. Die nachfolgenden Unterkapitel erläutern die einzelnen Aspekte dazu.

4.7.1 Aufbau der grafischen Darstellung

Zum Zeichnen der Simulation verwenden wir die Simulationpanel-Klasse, die auf dem AWT JPanel aufbaut. Jedes Mal, wenn der Inhalt neu gezeichnet werden muss, wird die Methode onPaint aufgerufen. In dieser wird jedes Mal entsprechend den aktuellen Einstellung jedes Hindernisse, jeder Fussgänger und jedes zusätzliche sichtbare Zeichenobjekt gezeichnet.

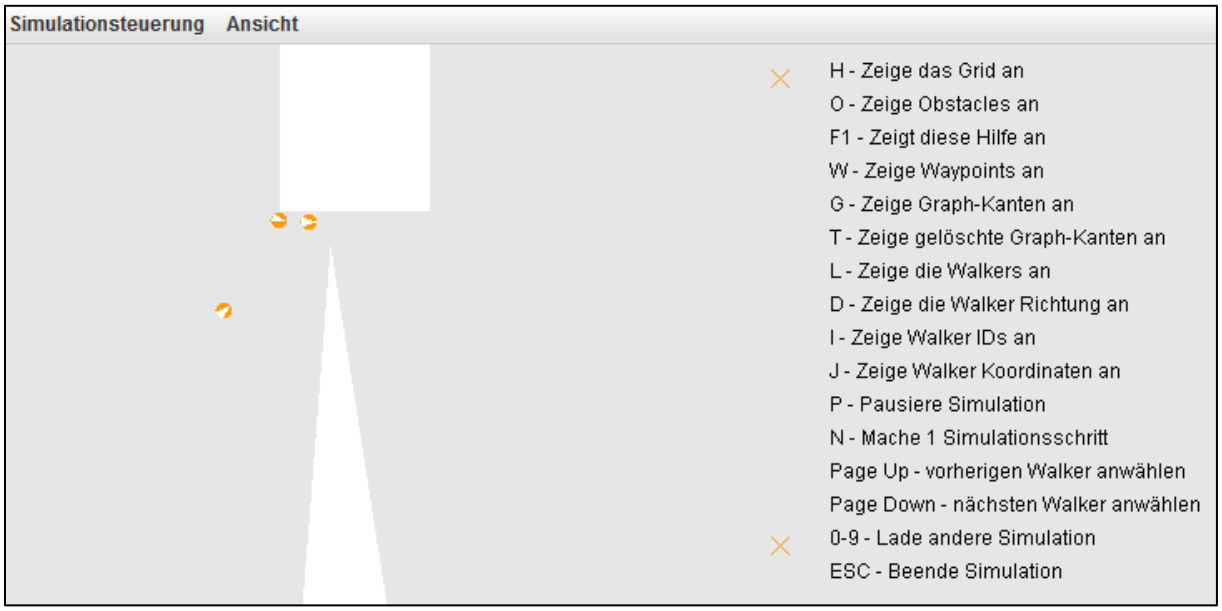
Die Simulation kann das Neuzeichnen der Darstellung mittels repaint()-Methode manuell auslösen. Dies wird zurzeit beispielsweise nach jedem kompletten Berechnungsschritt gemacht. Als Ausbaumöglichkeit ist es aber denkbar, dass dies weiter entkoppelt wird. Zum Beispiel könnte bei einer Grosssimulation, die selber schon viel Rechenzeit verbraucht, nur noch jedes zweite Mal oder allgemeiner jedes x-te Mal neu gezeichnet werden.

4.7.2 Elemente der grafischen Darstellung

Die Darstellung ist so aufgebaut, dass sie aus einzelnen Elementen besteht. Im Programmcode dient dazu die Klasse CDrawObject.

Jedes Element wird der definierten Reihenfolge nach und nach gezeichnet und ist mittels einer bestimmten Taste ein- und ausschaltbar, sprich es werden nur die Elemente gezeichnet, die im Status Eingeschaltet sind.

Zum Anzeigen der Tastenbelegung gibt es ein Hilfe-Element, das mittels F1 eingeblendet werden kann und dann auf der rechten Seite erscheint:



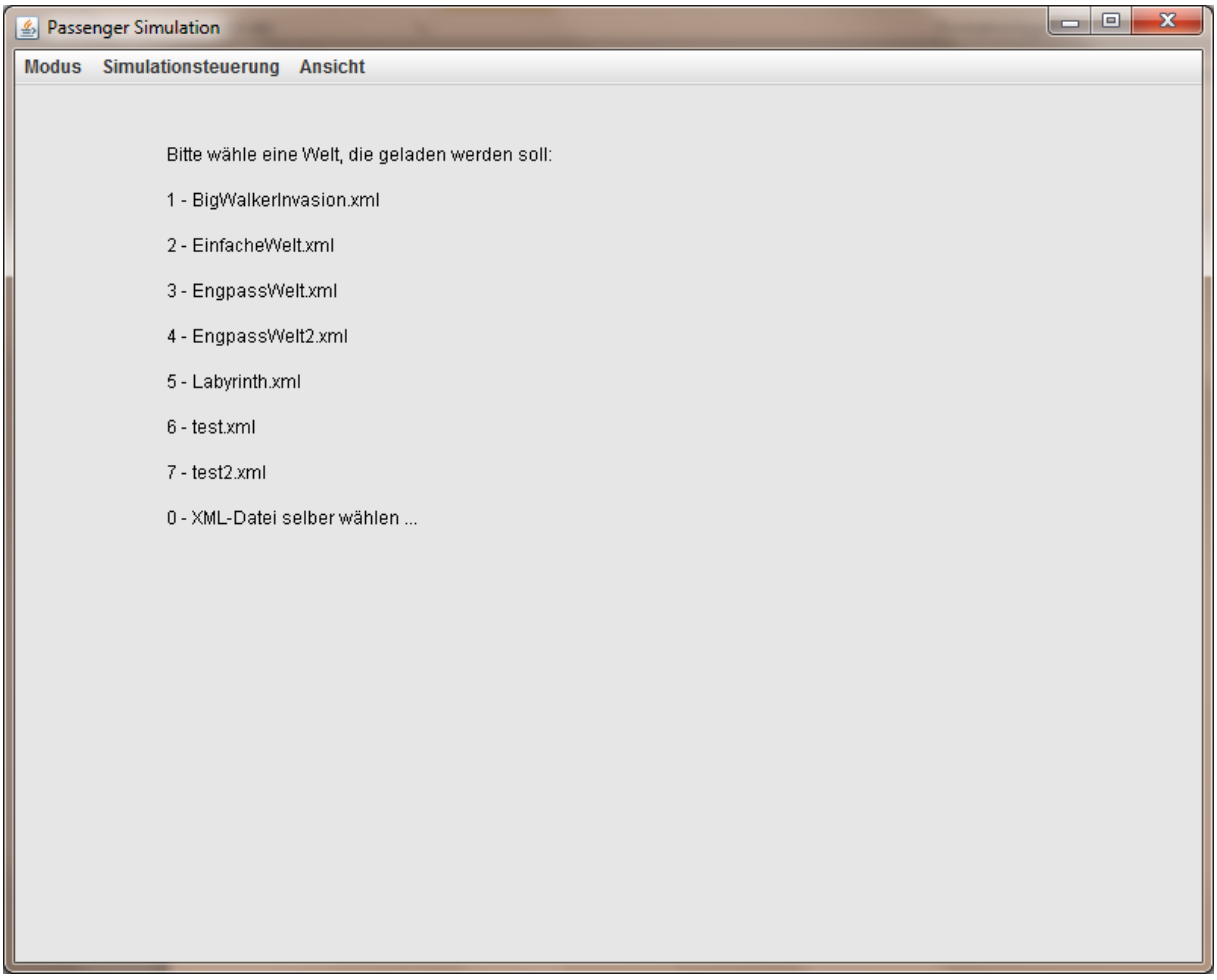
4.7.3 Steuern der Simulation

Zum Ausführen der einzelnen Berechnungsschritte existiert auf dem SimulationPanel ein Timer, der alle x Millisekunden die nächste Berechnung auslöst. Mittels der Taste P für Pausieren kann dieser Timer aktiviert respektive deaktiviert werden. Wenn der Timer deaktiviert ist, kann mit der Taste N von Hand der nächste Berechnungsschritt ausgeführt werden.

4.7.4 Laden und Verlassen der Simulationswelt

Zum Laden der verschiedenen Simulationswelten sind die Tasten 0 bis und mit 9 vorgesehen. Zum Verlassen einer Simulationswelt dient die ESC-Taste.

Wenn keine Simulationswelt geladen ist, erscheint eine Übersicht, welche Welten mit den jeweiligen Zahlentasten geladen werden können:



5 Anhang / Ressourcen

5.1 XSD-Schema des Landschaftseditors

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name ="Config">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="objList" type="tyObjList"/>
        <xs:element name="walkerList" type="tyWalkerList"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

    <xs:complexType name="tyObjList">
      <xs:sequence>
        <xs:element name="obj" minOccurs="0" maxOccurs="unbounded" type="tyObj"/>
      </xs:sequence>
    </xs:complexType>

    <xs:complexType name="tyObj">
        <xs:sequence>
            <xs:element name="point" minOccurs="2" maxOccurs="unbounded" type="tyPoint"/>
        </xs:sequence>
    </xs:complexType>

  <xs:complexType name ="tyPoint">
    <xs:attribute name ="x" use ="required" type="xs:nonNegativeInteger"/>
    <xs:attribute name ="y" use ="required" type="xs:nonNegativeInteger"/>
  </xs:complexType>

  <xs:complexType name ="tyWalkerList">
    <xs:sequence>
      <xs:element name="walker" minOccurs="0" maxOccurs="unbounded" type="tyWalker"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name ="tyWalker">
    <xs:sequence>
      <xs:element name="source" type="tyPoint"/>
      <xs:element name="target" type="tyPoint"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

5.2 Beispiel Config-XML-Datei

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<Config>
   <objList>
      <obj>
         <point x="220" y="0"/>
         <point x="310" y="0"/>
         <point x="310" y="100"/>
         <point x="220" y="100"/>
      </obj>
      <obj>
         <point x="250" y="120"/>
         <point x="220" y="500"/>
         <point x="310" y="500"/>
      </obj>
   </objList>
   <walkerList>
      <walker>
         <source x="20" y="20"/>
         <target x="520" y="300"/>
         <count c="1"/>
      </walker>
      <walker>
         <source x="20" y="300"/>
         <target x="520" y="20"/>
         <count c="1"/>
      </walker>
      <walker>
         <source x="20" y="100"/>
         <target x="520" y="300"/>
         <count c="1"/>
      </walker>
   </walkerList>
</Config>
```

Die vorhergehende XML-Datei sieht dann geladen wie folgt aus: