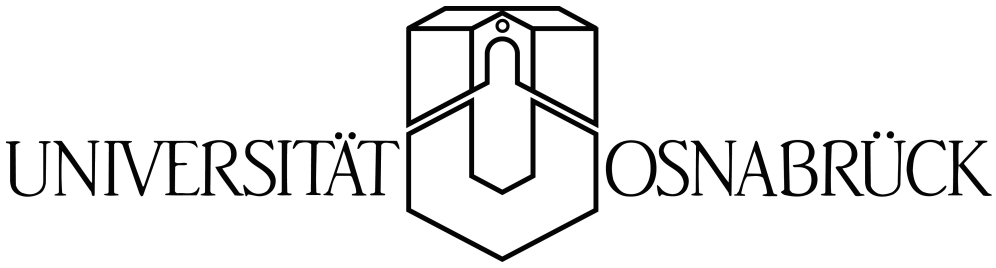

Interaktive Visualisierung von Zugablaufplänen in SVG am Beispiel des Bahnhofs Amsterdam-Schiphol

Diplomarbeit
von
Patrick Fox

betreut von
Prof. Dr. Oliver Vornberger
Prof. Dr. Peter Brucker

Fachbereich Mathematik/Informatik
Universität Osnabrück

27.01.2005

Vorwort

Diese Diplomarbeit entstand an der Universität Osnabrück im Institut für Informatik. Sie ist der schriftliche Teil der Diplomprüfungen für meinen Abschluss als Diplom-Mathematiker.

Danksagung

Ich möchte mich bei allen bedanken, die zum Gelingen dieser Diplomarbeit beigetragen haben. Besonderer Dank gilt den folgenden Personen:

- Herrn Prof. Dr. Oliver Vornberger für die gute Betreuung der Arbeit und seine Hinweise und Anregungen.
- Herrn Prof. Dr. Peter Brucker und Thomas Kampmeyer für die gute Betreuung seitens der Mathematik und für die Bereitstellung der benötigten Daten
- Ralf Kunze für seine gute Betreuung, die konstruktiven Vorschläge und das Korrekturlesen dieser Arbeit
- Tobias Schwegmann und Dorothee Langfeld für das Korrekturlesen dieser Arbeit
- Friedhelm Hofmeyer für die Bereitstellung eines Computers und der benötigten Software im Institut für Informatik

Ein ganz besonderer Dank gilt meinen Eltern, die mir das Mathematikstudium ermöglicht haben und mich während der Zeit unterstützt haben.

Warenzeichen

Inhaltsverzeichnis

Abbildungsverzeichnis

Quellcodeverzeichnis

1 Einleitung

Die vorliegende Diplomarbeit entstand auf Grundlage eines Projekts der Optimierungsgruppe um Prof. Dr. Brucker. Ziel des Projektes war es, für den Bahnhof Amsterdam-Schiphol einen Fahrplan zu erstellen, der das erhöhte Zugaufkommen in den nächsten Jahren berücksichtigt. Die Aufgabe wurde mit einem speziell entwickelten Programm gelöst, welches Algorithmen aus der Optimierungstheorie verwendet. Es arbeitet mit Textdateien als Eingabe, die verschiedene Informationen über den Gleisplan und über die Züge enthalten. Dies sind zum Beispiel die Geschwindigkeitstabellen der verschiedenen Zugtypen, vorgegebene Ein- und Ausfahrtszeiten an den Modellgrenzen oder die Zuglänge der einzelnen Züge.

Das Programm erzeugt den Fahrplan ebenfalls in Form einer Textdatei. Er besteht aus einer chronologisch geordneten Liste, die die Durchfahrtszeiten aller Züge durch definierte Punkte im Gleisplan enthält, jeweils mit aktueller Geschwindigkeit.

Aus der Art der Ausgabe ergibt sich das Problem, dass diese nur sehr schwer zu analysieren ist. Um nachzuweisen, ob überhaupt ein gültiger Fahrplan erzeugt wurde, wäre es nötig die Fahrpläne aller Züge zu vergleichen. Auch lässt sich sehr schwer prüfen, ob sich an manchen Stellen Züge stauen.

Daher wäre es eine große Hilfe, den Fahrplan zu visualisieren, um eine bessere Analyse durchführen zu können und Fehler besser zu erkennen.

1.1 Aufgabenstellung

Ziel der Visualisierung ist es, die Zugbewegung bezüglich der Geschwindigkeit möglichst genau zu animieren. Das setzt eine längentreue Darstellung des Gleisplans voraus.

Die Erzeugung und Darstellung übernimmt eine Applikation, die beliebige Fahrpläne beliebiger Gleispläne darstellt. Der Gleisplan und der Fahrplan sind die Parameter, die für eine Erzeugung benötigt werden. Sie werden in Dateiform eingelesen.

Bei der Darstellung der Visualisierung besteht die Aufgabe darin, neben Funktionen zur Navigation und zum Zoomen, zusätzliche Informationen über die Züge anzuzeigen, um die Visualisierung besser analysieren zu können.

1.2 Aufbau der Arbeit

Zu Beginn der Arbeit wird der Bahnhof Amsterdam-Schiphol vorgestellt. In Kapitel 2 wird beschrieben, wie man den Gleisplan des Bahnhofs und die Zugbewegungen in einem mathematisch, logischen Modell zusammenfassen kann. Es wird

ein Überblick über die vorhandenen Daten gegeben und das graphenbasierte Modell erläutert, mit welchem das Optimierungsprogramm arbeitet.

Kapitel 3 gibt eine kurze Einführung in die *eXtensible Markup Language* (XML), da sie zum Einen Grundlage von *Scalable Vetcor Graphics* (SVG) ist und zum Anderen als Sprache zur Beschreibung des Gleisplans eingesetzt wird. Es wird beschrieben, wie über das *Document Object Model* (DOM) sprachenunabhängig darauf zugegriffen werden kann. Die Applikation greift beim Verarbeiten der XML-Gleisplandatei und bei der Erzeugung des SVG-Dokuments mit Java genauso über DOM auf die XML-Struktur zu, wie das ECMAScript während der Anzeige der SVG-Grafik. Neben einer kurzen Zusammenfassung der wichtigsten Eigenschaften von SVG werden einige für die Visualisierung wichtige Aspekte näher erläutert.

Das anschließende Kapitel 4 erläutert die Grundlagen der Bézierkurven und deren Verwendung als Weg-Zeit-Funktion.

In Kapitel 5 wird das *Standard Widget Toolkit* (SWT) vorgestellt, welches für die Darstellung der Programmoberfläche verwendet wurde. Es werden die Vorteile von SWT in Verbindung mit SVG erläutert, die den Ausschlag gaben, nicht die üblichen Toolkits Swing oder AWT zu verwenden.

Die folgenden drei Kapitel gehen näher auf die Realisierung der Applikation ein. Kapitel 6 beschreibt die Art und Form der Eingabedateien, die die Applikation verarbeitet. Im 7. Kapitel wird der Aufbau der Klassenhierarchie der Applikation an Hand eines UML-Diagramms skizziert und die Abhängigkeit der Klassen beschrieben. In Kapitel 8 wird der Vorgang erläutert, wie die Applikation aus den Eingabedaten das SVG-Dokument erzeugt und widmet sich anschließend der erzeugten SVG-Datei. Neben dem Aufbau der Grafik werden die Interaktionsmöglichkeiten näher beschrieben.

Im abschließenden Fazit werden die Ergebnisse der Visualisierung zusammengefasst und eine Bewertung gegeben. Dabei wird im Speziellen auf die Verwendbarkeit von SVG eingegangen.

2 Das allgemeine Modell

Wie in der Einleitung bereits erwähnt, wurde das Optimierungsproblem speziell für den Bahnhof Amsterdam-Schiphol formuliert. Für das Jahr 2007 ist geplant, 27 Züge pro Stunde in jeweils beiden Richtungen durch den Bahnhof Amsterdam-Schiphol fahren zu lassen. Da das aktuelle System, welches die Planung des Fahrplans bislang übernimmt, für das erhöhte Zugaufkommen nicht mehr ausgelegt ist, wurde ein Optimierungsprogramm entwickelt, dass den Fahrplan der Züge in einem Bereich um den Bahnhof berechnet. Das folgende Kapitel erläutert, wie man das Problem in ein allgemeines Modell fassen kann [BrKaSt2003].

2.1 Der Bahnhof Amsterdam-Schiphol

Der Bahnhof Amsterdam-Schiphol liegt unter dem gleichnamigen internationalen Flughafen in einem Tunnel. Der Tunnel umfasst vier Gleise, jeweils zwei in beide Richtungen. Nordöstlich spalten sich diese Gleise in zwei Richtungen auf. In entgegengesetzter Richtung wird zunächst der Bahnhof Hoofddorp erreicht, der Start- und Endpunkt einiger Nahverkehrszüge ist. Bei der Bereitstellung dieser Züge ist es erforderlich, an dieser Stelle einige wenige Gleisstücke in beiden Richtungen zu befahren. Hinter Hoofddorp beginnt eine Hochgeschwindigkeitstrecke. Der Bereich, den das Modell betrachtet, umfasst ca. 25 km und schließt die nächstgelegenen Haltestellen ein, die von Amsterdam-Schiphol in beiden Richtungen erreicht werden.

Die Einfahrtszeiten der Züge in diesen Bereich und die Aufenthaltsdauer darin sind vorgegeben, um es in den landesweiten periodischen Bahnfahrplan zu

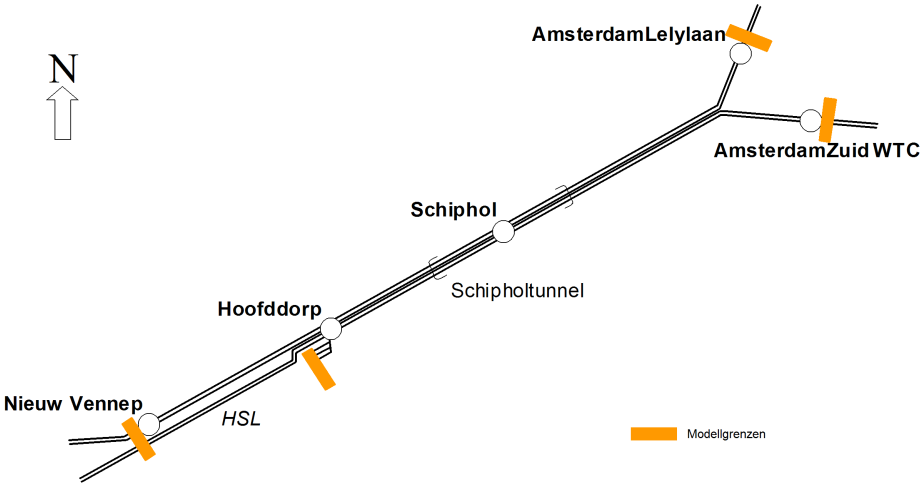

Abbildung 2.1: Modell des Bahnhofs Amsterdam-Schiphol

integrieren, der sich stündlich wiederholt. Hochgeschwindigkeitszüge dürfen nur auf entsprechenden Gleisen fahren, alle anderen Züge benutzen alternative Gleise. Das Ziel ist es, einen Fahrplan zu generieren, der für jeden der 54 Züge pro Stunde eine Trasse durch den Gleisplan enthält, die zeitliche Abfolge der Züge beschreibt und dabei die Restriktionen beachtet.

2.2 Das Optimierungsmodell

Ein Gleisplan ist in sogenannte Blöcke aufgeteilt. Ein Block wird für einen Zug über ein Signal freigegeben oder gesperrt. Ein grünes Signal bedeutet Freigabe, ein rotes Signal sperrt einen Block. Einem roten Signal ist aber immer ein gelbes Signal vorgelagert, welches dem Zugführer signalisiert, dass er für den einfahrenden Block eine Freigabe besitzt, aber vor dem nächsten Signal zum Stehen kommen muss.

Fährt ein Zug in einen Block ein, wird der Block exklusiv für den Zug gesperrt und es darf kein weiterer Zug einfahren. Das bedeutet, dass bei entsprechend großen Blöcken ein unnötig großer Abschnitt nicht von anderen Züge befahren werden kann. Daher wurden die Blöcke im Modell nochmals in kleinere Segmente aufgeteilt.

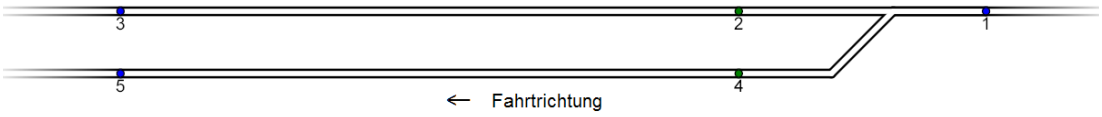

Abbildung 2.2: Segmentierung von Blöcken

In Abbildung 2.2 ist ein Ausschnitt aus dem Gleisbild zu sehen. Durch die Signalknotenpaare 1, 3 und 1, 5 werden zwei Blöcke definiert. Die Knoten 2 und 4 teilen die beiden Blöcke jeweils in zwei Segmente auf. Wenn ein Zug im Block auf dem oberen Gleis den Knoten 2 vollständig durchfahren hat, kann der Block, der über die Weiche auf das untere Gleis führt, bereits freigegeben werden. Ein Folgezug müsste somit nicht warten, bis der gesamte obere Block freigegeben wird.

Das Optimierungsprogamm verwendet als Repräsentation des Gleisplans einen gerichteten Graphen. Die Knoten stellen die Signale und Segmentierungspunkte dar, die Gleise finden sich in den Kanten wieder.

Die Berechnung der Zeiten und Geschwindigkeiten der Züge erfolgt auf Grundlage von Geschwindigkeitstabellen, die für jeden Zugtypen angeben, wie schnell ein Zug beschleunigen und bremsen kann. Die Tabellen bestehen aus einer diskreten Liste von Zeitpunkten, denen jeweils ein Geschwindigkeitswert zugeordnet ist.

Das Optimierungsprogramm erzeugt im Fahrplan für jeden Zug eine Liste von Zeitpunkten mit zugehörigen Geschwindigkeiten. Diese Zeitpunkte sind allerdings auf die Durchfahrtszeiten der Knoten beschränkt. Das bedeutet, dass die Geschwindigkeit zwischen zwei benachbarten Knoten jeweils interpoliert werden muss. Neben der Geschwindigkeit in beiden Knoten ist der Abstand zwischen den beiden Knoten bekannt, sowie die Zeitdauer, die ein Zug zwischen den Knoten benötigt. Aus der Menge der Parameter ergibt sich, dass die Interpolationsfunktion, die den zurückgelegten Weg zwischen zwei Knoten in Abhängigkeit von der Zeit angibt, ein Polynom dritten Grades ist. Aus dieser Weg-Zeit-Funktion lässt sich die Geschwindigkeit berechnen.

Da beim Optimierungsmodell die Geschwindigkeitsberechnung nur eindimensional in Gleisrichtung erfolgt, enthält es keinerlei Information über den Abstand der Gleise oder Beginn und Ende einer Weiche. Daher wurde der Gleisplan für die Visalisierung durch die sogenannten Abbiegepunkte ergänzt, etwa an den Stellen, wo Weichen einen Gleiswechsel ermöglichen. Um eine bessere Übersicht zu erzielen, ist der Abstand der Gleise allerdings nicht maßstabsgetreu umgesetzt.

3 XML und SVG - die Grundlagen

In der Applikation wird als wesentliches Datenformat die *eXtensible Markup Language* (XML) verwendet, sowohl bei den Eingabedaten, als auch bei der Ausgabe der Visualisierung im SVG-Format. Daher werden im Folgenden die Grundlagen von XML und SVG zusammengefasst und die notwendigen Techniken zur Verarbeitung von XML beschrieben.

3.1 eXtensible Markup Language

Die *eXtensible Markup Language* (XML) ist ein plattformunabhängiger Standard, der 1998 vom W3C[1] verabschiedet wurde [W3C2005a]. XML ist eine Metasprache, die dazu dient, eigene Auszeichnungssprachen und Dokumenttypen zu beschreiben. XML stellt dabei eine Struktur zur Verfügung, mit der man über Tags eigene Elemente definieren kann. Die Anzahl der verschiedenen Elemente legt so den Umfang des Dokumenttyps fest. Ein wesentliches Merkmal von XML ist seine Repräsentierung in Textform. Auf diese Weise lässt sich XML sehr einfach lesen und editieren.

3.2 Definition des Dokumenttyps

Die Definition eines Dokumenttyps erfolgt entweder durch eine *Document Type Definition* (DTD) oder durch ein *XML-Schema* [W3C2005g]. Beide Ansätze verfolgen das Ziel, die Struktur einer XML-Datei zu beschreiben.

Eine XML-Datei besteht aus einer Menge von Elementen mit ihren Attributen und ihrem Inhalt. Eine Strukturierung erfolgt dadurch, dass Elemente andere Elemente enthalten können. Dadurch entsteht eine Baumstruktur mit einem eindeutigen Wurzelelement. Eine DTD und ein XML-Schema definieren neben den möglichen Element- und Attributnamen auch, in welcher Form die Elemente ineinander verschachtelt werden können.

Für eine DTD existiert eine eigene Syntax und Semantik, die innerhalb des XML-Standards festgelegt ist. XML-Schema wurde als eigener Standard zur Definition von Dokumenttypen entwickelt. Da XML-Schema eine Weiterentwicklung der DTD darstellt, bietet es mehr Möglichkeiten als eine DTD, macht es aber auch komplexer. Zum Beispiel lässt sich in einem XML-Schema die Anzahl der Kinder eines Elements über Zahlwerte genau angeben, in einer DTD lässt sich der Wert nur auf 0, 1 oder * (viele) einschränken. Weiter lassen sich die Element- und Attributwerte nur in XML-Schema auf bestimmte Datentypen einschränken. Ein großer Vorteil von XML-Schema ist, dass es ebenfalls in XML formuliert ist.

[1] World Wide Web Consortium [W3C2005]

Mit der Definition der Struktur eines XML-Dokumentes wird es erst möglich, dass es von einem Programm eingelesen werden kann. Das Programm muss sich nämlich darauf verlassen können, dass das XML-Dokument entsprechend des XML-Schemas oder der DTD aufgebaut ist, um die enthaltenen Daten verarbeiten zu können. Dazu kann das Dokument beim Einlesen automatisch darauf überprüft werden, ob es entsprechend des XML-Schemas oder der DTD aufgebaut ist. Das ist Aufgabe eines sogenanntes *Parsers*.

3.3 Parser

XML besitzt eine relativ einfache Syntax. Das wird durch die klare Auftrennung der Auszeichnungen des Inhalts, den sogenannten *Tags* und dem Inhalt selbst durch spitze Klammern erreicht. Dadurch lassen sich XML-Dokumente, die von einem Programm verarbeitet werden sollen, besonders einfach einlesen. Diese Aufgabe übernimmt der *Parser*.

Ein XML-Dokument muss zunächst nach den syntaktischen Grundregeln korrekt aufgebaut sein, um eingelesen werden zu können. Man spricht dann davon, dass das Dokument *wohlgeformt* (wellformed) ist. Darüberhinaus kann ein Parser prüfen, ob das Dokument *gültig* (valid) ist, also einer DTD oder einem XML-Schema genügt.

Der Parser arbeitet das XML-Dokument Event-basiert ab. Je nach gefundenem XML-Objekt werden dem aufrufenden Programm alle Informationen wie Inhalt und Attributwerte über das Element geliefert. Das bedeutet, dass die Informationen direkt verarbeitet werden müssen, da sie beim nachfolgenden Element dem Parser nicht mehr zur Verfügung stehen. Da der Parser sequenziell durch das XML-Dokument läuft und jeweils nur einen kleinen Teil betrachtet, ist er sehr schnell und benötigt wenig Speicherplatz.

Ein Parser kann in Java über die *Simple API for XML* (SAX) angesprochen werden [SAX2004]. Sie ist nicht vom W3C als Standard definiert, sondern wurde ursprünglich im Rahmen der XML-DEV Mailingliste entwickelt. Sie stellt aber einen de-facto Standard dar, um in Java XML-Dokumente zu verarbeiten. Seit der Version 1.4 ist ein XML-Parser, der SAX implementiert, bereits in Java enthalten.

Bei einem SAX-Parser können `Handler` angemeldet werden, die Methoden enthalten, die bei den verschiedenen Ereignissen wie z.B. dem Öffnen oder Schliessen eines Elements aufgerufen werden.

Zur Fehlerbehandlung stellt SAX das Interface `ErrorHandler` bereit. Es definiert folgende Methoden, die ein ErrorHandler implementieren muss:

```
void warning(SAXParseException e);
void error(SAXParseException e);
void fatalError(SAXParseException e);
```

Ein `FatalError` liegt vor, sobald der Parser auf eine Zeichenfolge gestoßen ist, die nicht dem gültigen syntaktischen Aufbau einer XML-Datei entspricht. Eine der anderen beiden Methoden wird aufgerufen, wenn der Aufbau des Dokuments zwar syntaktisch korrekt ist, aber nicht der DTD entspricht. Welche Methode aufgerufen wird, ist von der Art des Fehlers abhängig. Einen Überblick über die Fehlertypen gibt die XML-Spezifikation [W3C2005a].

3.4 Document Object Model

Das *Document Object Model* (DOM) ist wie XML selbst ein W3C-Standard. Es sieht einen anderen Ansatz als SAX vor, um auf die XML-Struktur zuzugreifen. Im Gegensatz zur sequentiellen Abarbeitung des Dokuments wird das komplette XML-Dokument zunächst in den Speicher eingelesen und in einer baumartigen Objekthierarchie, dem sogenannten DOM-Tree abgelegt [W3C2005f].

Um in Java ein XML-Dokument in Form eines DOM-Trees einlesen zu können wird ein `DocumentBuilder` benötigt, der durch eine `DocumentBuilderFactory` erzeugt wird. Dieser muss zunächst mitgeteilt werden, dass sie einen `DocumentBuilder` liefern soll, der das XML-Dokument gegen eine eingebundene DTD validiert.

```
DocumentBuilderFactory factory =
           DocumentBuilderFactory.newInstance();
factory.setValidating(true);
DocumentBuilder builder = factory.newDocumentBuilder();
```

Ein `DocumentBuilder` greift beim Einlesen eines XML-Dokumentes auf einen SAX-Parser zurück. Ebenso wie der XML-Parser ist der `DocumentBuilder` in der Laufzeitumgebung von Java enthalten. Der `DocumentBuilder` und der SAX-Parser sind dabei so eng miteinander verzahnt, dass beim `DocumentBuilder` ebenfalls ein `ErrorHandler` angemeldet werden kann, der intern an den SAX-Parser weitergereicht wird.

```
builder.setErrorHandler(StationErrorHandler);
```

Das `document` wird schliesslich mit dem Methodenaufruf `builder.parse(uri)` erzeugt, der in `uri` den Pfad zu der XML-Datei enthält.

Das DOM definiert nicht nur die Struktur eines XML-Dokuments, sondern enthält auch eine umfangreiche Schnittstelle, um einen DOM-Tree zu manipulieren und zu verarbeiten. Man kann sich über diese Schnittstelle sehr variabel im DOM-Tree bewegen und auf die einzelnen Elemente zugreifen. Auf diese Weise lässt er sich sehr flexibel bearbeiten und dynamisch verändern. Um mit dem DOM in Java

arbeiten zu können, wird eine DOM-Implementation benötigt, die die Interfaces im Paket `org.w3c.dom` implementieren. Eine Implementation der Interfaces ist in der Laufzeitumgebung von Java nicht enthalten. In der Applikation wird die DOM-Implementation verwendet, die im Xerces2 Java Parser der Apache Foundation enthalten ist [Apac2004].

Das *Document Object Model* ist für interaktive Anwendungen sehr gut geeignet. Allerdings ist der Speicheraufwand für große XML-Dateien sehr hoch und der Zugriff auf das DOM recht langsam, da die Elemente und auch die Attribute selbst speicherintern als Objekte in einer Baumstruktur vorgehalten werden.

Das DOM ist eine sprachunabhängige Schnittstelle. Neben Java implementieren viele Programmier- und Skriptsprachen diese Schnittstelle. Speziell die Skriptsprachen ermöglichen es, ein XML-Dokument nicht nur vor, sondern gerade während seiner Anzeige zu verändern. Das beste Beispiel hierfür ist *Dynamic HTML*[2] (DHTML). HTML ist sehr ähnlich zu XML und es existiert mit XHTML eine HTML-Variante, die XML-konform ist. Eine HTML-Seite lässt sich ebenfalls über DOM ansprechen. In Verbindung mit JavaScript oder ähnlichen Skriptsprachen lassen sich die Elemente eines HTML-Dokuments dynamisch, d.h. während der Anzeige verändern. Die Möglichkeiten sind dabei nahezu unbegrenzt. Sie reichen vom einfachen Austauschen angezeigter Textinhalte über dynamische Navigationsleisten bis zu komplexen Animationen.

3.5 ECMAScript

JavaScript wurde 1995 von Netscape eingeführt und lizenziert und war anfangs eine proprietäre Sprache. Microsoft entwickelte zeitgleich eine eigene JavaScript-Variante namens JScript, um sich den Lizenzvorgaben von Netscape zu entziehen. Netscape ließ JavaScript 1997 von der *European Computer Manufacturers Association* (ECMA) unter dem Namen ECMAScript als Industriestandard definieren, um ihn gegenüber Microsofts JScript, das viele plattformabhängige Erweiterungen besitzt, in den Status eines offiziellen Standards zu versetzen [Ecma2004].

ECMAScript stellt heutzutage eine Schnittmenge von JavaScript und JScript dar, die die Schnittstelle zum DOM vollständig umsetzt und so geeignet ist, nicht nur HTML-Dokumente, sondern beliebige XML-Dokumente, wie zum Beispiel SVG, manipulieren zu können. Alle gängigen Webbrowser beherrschen ECMAScript. Das SVG-Plugin von Adobe enthält seit der Version 3 eine eigene Implementation von ECMAScript, um nicht vom ECMAScript des Browsers abhängig zu sein. Es ist die offene ECMAScript-Implementation, die für Mozilla entwickelt wurde.

ECMAScript ähnelt von der Syntax her Java, ist aber nicht so mächtig. Wie ECMAScript in SVG verwendet wird, beschreibt Kapitel 3.6.6.

[2] *Hypertext Markup Language*

3.6 Scalable Vector Graphics

SVG wurde vom W3C anfang 2003 in der Version 1.1 als Recommendation verabschiedet. Die Version 1.2 befindet sich zur Zeit in der Entwicklung. Inzwischen existieren auch eine SVG-Versionen für mobile Geräte, SVG Tiny für Mobiltelefone und SVG Basic für PDAs. Diese Versionen sind auf die grundlegenden Eigenschaften von SVG reduziert worden, um den begrenzten Ressourcen gerecht zu werden.

SVG ist eine in XML formulierte Sprache zur Beschreibung von 2D-Grafiken mittels Vektorgrafik. Sie ermöglicht beliebig skalierbare und sogar in vielen Teilen animierbare Vektorbilder. SVG ähnelt in vielen Punkten dem populären Flashformat. Allerdings bietet SVG darüber hinaus einige Vorteile. Es basiert komplett auf dem leicht lesbaren und weit verbreiteten XML-Format. Und es ist auf Grund der Standardisierung des W3C öffentlich frei nutzbar und harmoniert daher auch mit anderen Standards wie DOM, CSS und ECMA-Script.

SVG ist ein Vektorgrafikformat. Im Gegensatz zu Rastergrafiken, die die Bildinformation pixelweise enthalten, wird eine Vektorgrafik aus geometrischen Grundformen zusammengesetzt. Um eine Vektorgrafik eindeutig zu beschreiben, genügt es, eine mathematische Beschreibung der geometrischen Objekte zu definieren. Beispielsweise wird ein Kreis über seinen Mittelpunkt und Radius eindeutig bestimmt. Hinzu kommt lediglich eine Beschreibung der Form und Farbe der Begrenzungslinie und die Art der Füllung des Kreises.

Der Vorteil, der aus diesem Ansatz resultiert, ist die geringe Dateigröße einer Vektorgrafik. Allerdings lassen sich nur solche Grafiken in einem Vektorgrafikformat beschreiben, die aus einer überschaubaren Anzahl von geometrischen Grundobjekten zusammengesetzt werden können. Rastergrafiken ließen sich nicht sinnvoll in einem Vektorformat beschreiben, da für jedes Pixel ein eigenes Objekt angelegt werden müsste, was zu einem erheblichen Mehraufwand an Speicher führen würde.

Ein weiterer großer Vorteil von Vektorgrafiken liegt darin, dass sie stufenlos skalierbar sind und so in jeder Skalierungsstufe eine gleichbleibende Auflösung bieten. Das wird dadurch erreicht, dass die Grafik in jeder Skalierungsstufe aus der mathematischen Beschreibung der grafischen Objekte mit einer entsprechenden Auflösung neu berechnet wird. Auf diese Weise lassen sich die Treppeneffekte, die beim Skalieren von Rastergrafiken enstehen, vollständig vermeiden.

Im Folgenden wird an Hand von einigen Beispielen ein sehr kurzer Überblick über den Funktionsumfang von SVG gegeben. Auf einige spezielle Eigenschaften, die in der Applikation verwendet wurden, wird näher eingegangen.

Eine sehr umfangreiche Beipielsammlung, die viele Aspekte von SVG berücksicht, findet sich unter [Mei2005].

3.6.1 Grundgerüst eines SVG-Dokuments

Als XML-Datei enthält eine SVG-Datei zunächst den XML-Prolog mit Verweis auf die DTD von SVG. Im Wurzelelement `svg` wird in den Attributen `width` und `height` die Größe der Grafik angegeben.

```
<?xml version="1.0" standalone="no" ?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
  "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="100%" height="100%">
</svg>
```

Unterhalb des Wurzelelements kann ein `defs`-Element definiert werden, dass das ECMAScript für die Interaktion mit dem Benutzer und CSS-Angaben zur Formatierung enthält, die man in der gleichen Art verwendet, wie man es von HTML-Dokumenten gewohnt ist. Da es sich bei ECMAScript und CSS nicht um XML handelt, wird es jeweils in einem CDATA-Bereich eingeschlossen. Des Weiteren können im `defs`-Bereich Prototypen von Elementen angelegt werden, die später mehrfach verwendet werden können.

```
<defs>
  <script type="text/ecmascript">
    <![CDATA[
      function init()
      {
        alert('Hello World!');
      }
    ]]>
  </script>

  <style type="text/css">
    <![CDATA[
      .node { fill:blue; stroke:black; }
      .edge { stroke:black; stroke-width:4px; fill:none; }
    ]]>
  </style>

  <path id="edge" d="M 0,0 l 500,0" class="edge" />

  <circle id="node" cx="0" cy="0" r="5" class="node" />
</defs>
```

Nach dem `defs`-Bereich beginnt der eigentliche grafische Bereich. Hier werden nun die grafischen Grundelemente definiert, aus denen sich die SVG-Grafik zusammensetzen soll.

3.6.2 Grundelemente

Als grafische Grundelemente dienen einfache Vektorgrafikelemente wie z.B. Linie, Rechteck, Kreis, Polygon und das Pfad-Element, die entsprechend formatiert und frei positioniert werden können. Das Pfad-Element stellt das flexibelste Grafikelement dar. Mit ihm können offene oder geschlossene Linienobjekte definiert werden, die nicht nur aus gradlinigen Teilstücken zusammengesetzt sein müssen, sondern auch Bézierkurven zweiten und dritten Grades sowie elliptische Kurvenelemente enthalten können.

Beispiel:

```
<polygon fill="blue" stroke="red" stroke-width="3"
  points="150 75, 179 161, 269 161, 197 215, 223 301,
          150 250, 77 301, 103 215, 31 161, 121 161" />
<ellipse cx="200" cy="80" rx="80" ry="50"
  fill="green" />
<rect x="50" y="50" width="100" height="100" fill="yellow"
  stroke="black" stroke-width="3" />
<path d="M 60,120 c 20,-30 60,-30 80,0 c 40,60 100,60 140,0"
  stroke="orange" fill="none" stroke-width="5" />
```

Diese Anzeige dieser vier Elemente ist in Abbildung 3.1 dargestellt.

Ein weiteres spezielles Grundelement stellt Text dar, der in vielen Parametern formatiert werden kann. Dazu zählt beispielsweise Schriftart, Schriftgrösse, Schriftschnitt, Laufrichtung, etc. Die übrigen Grundelemente können als Pfad verwendet werden, an dem sich Text ausrichten lässt.

Da SVG im XML-Format vorliegt, hat es den großen Vorteil, dass man nach Text suchen kann. Browser und Plugins, bzw. Suchmaschinen haben so die Möglichkeit, den Textinhalt einer SVG-Grafik zu indizieren.

Beispiel:

```
<line x1="120" y1="50" x2="120" y2="140"/>
<text x="60" y="40" style="font-size:18px; fill:#333;">
  Buendiger Text!
  <tspan x="120" dy="30" style="text-anchor: start">
```

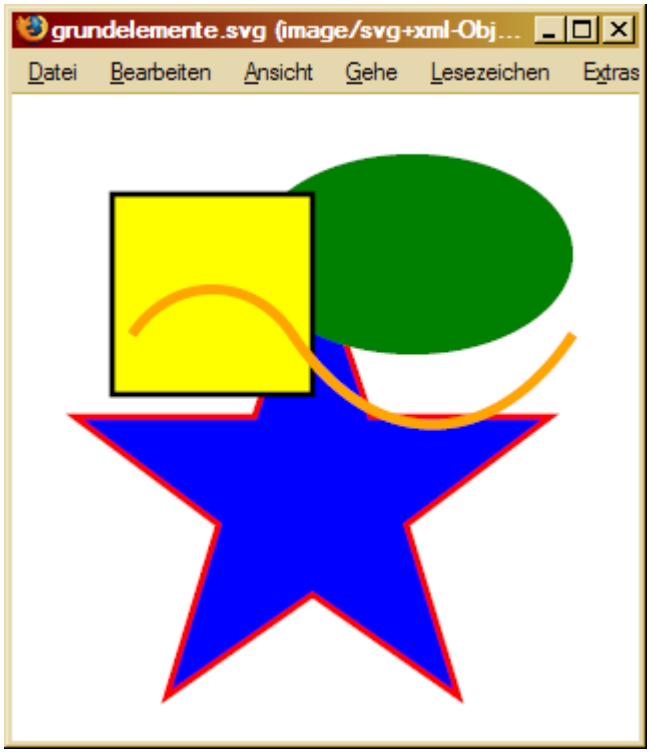

Abbildung 3.1: Einige SVG-Grundelemente

```
    linksbuendig</tspan>
  <tspan x="120" dy="30" style="text-anchor: middle">
    zentriert</tspan>
  <tspan x="120" dy="30" style="text-anchor: end">
    rechtsbuendig</tspan>
</text>
```

Abbildung 3.2 zeigt das Beispiel im Browser, bei dem ein Text in mehrere Teile aufgeteilt ist, wobei die Ausrichtung der einzelnen Teile variiert.

Zur weiteren Strukturierung eines SVG-Dokuments kann das g-Element verwendet werden, welches mehrere Elemente zu einer Gruppe zusammenfasst. Dies erhöht nicht nur die Lesbarkeit der SVG-Datei, sondern hilft auch, auf eine Menge von Objekten gleichzeitig zuzugreifen, um sie beispielsweise zu formatieren oder zu transformieren.

```
<g transform="scale(0.5)">
  <path id="edge1" d="M 0,0 l 500,0" class="edge" />
  <path id="edge2" d="M 50,80 l 500,0" class="edge" />
  <path id="edge3" d="M 50,150 l 500,0" class="edge" />
</g>
```

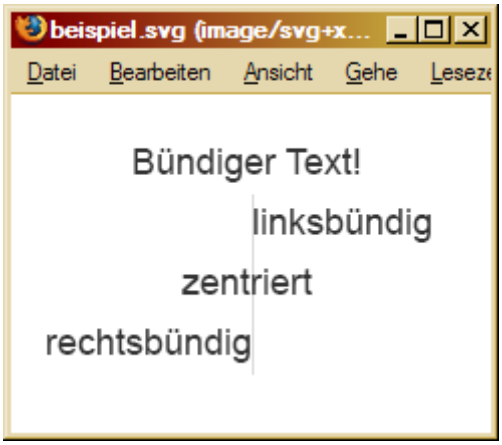

Abbildung 3.2: Text mit verschiedenen Ausrichtungen

3.6.3 Koordinatensystem, Transformation

Die Grundelemente werden über Koordinaten im karthesischen Koordinatensystem des SVG-Dokuments positioniert, das seinen Ursprung in der linken, oberen Ecke hat. Zusätzlich kann man einzelne Elemente oder eine Gruppe von Elementen durch affine Transformationen (Skalierung, Translation, Scherung und Rotation) verschieben. Dadurch werden für transformierte Elemente jeweils eigene Koordinatensysteme erzeugt. Die Positionierung der Grafikobjekte ist nicht auf den Anzeigebereich beschränkt. So können Objekte außerhalb des sichtbaren Bereichs definiert werden und später während der Anzeige etwa durch Interaktion mit dem Benutzer in den Anzeigebereich verschoben werden.

Beispiel:

```
<rect x="10" y="10"
  width="100" height="100" style="fill:#FFC; stroke:#F00;"/>
<rect transform="translate(180,0) rotate(30)" x="10" y="10"
  width="100" height="100" style="fill:#FFC; stroke:#F00;"/>
<rect transform="translate(300,0) skewY(30)" x="10" y="10"
  width="100" height="100" style="fill:#FFC; stroke:#F00;"/>
```

Das Beispiel enthält ein Rechteck, auf welches zwei verschiedene Transformationen angewendet wurden. Das Original ist in der Abbildung 3.3 links abgebildet. Das mittlere Rechteck entspricht dem linken nach Rotation, und das rechte stellt das Ursprungsrechteck in y-Richtung geschert dar. Zur besseren Übersicht sind die Rechtecke noch translatiert.

Nicht nur durch die Transformation von Elementen ändert sich das Koordinatensystem. Man kann im `svg`-Element auch durch das Attribut `viewbox` explizit

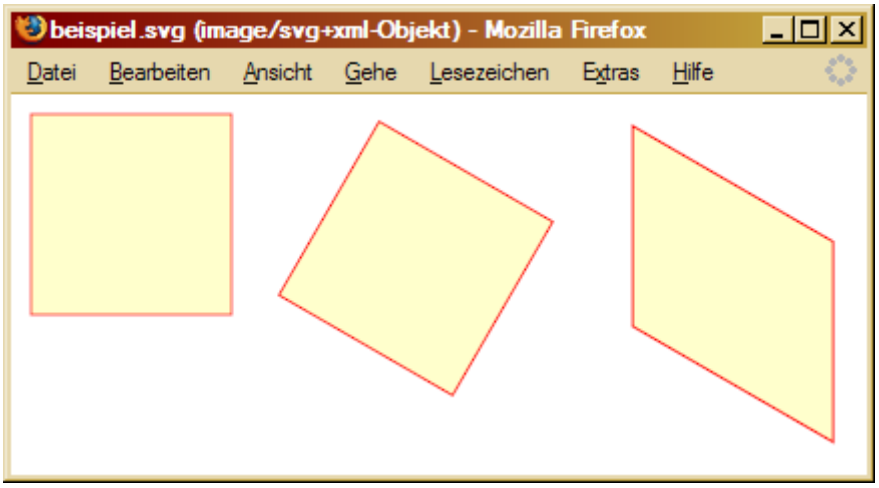

Abbildung 3.3: Transformation

ein anderes Koordinatensystem definieren. Beispielsweise erzeugt das folgende `svg`-Element

```
<svg width="640" height="480" viewBox="0 0 1 1"/>
```

eine Grafik von 640 x 480 Pixel Größe. Darin wird das Einheitsquadrat auf die gesamte Bildgröße skaliert dargestellt. Die Richtung auf den Koordinatenachsen zu vertauschen ist mit dem `viewBox`-Attribut allerdings nicht möglich. Es stellt aber eine interessante Anwendung zur Verfügung: Durch Animation der Werte lässt sich ein animiertes Zooming erzeugen.

3.6.4 Formatierung

Die Formatierungen der Elemente können auf zwei verschiedene Arten gesteuert werden. Zum Einen besitzen die Elemente spezifische Attribute, über die man wesentliche Eigenschaften der Formatierung angeben kann. Zum Anderen kann man das Aussehen der Elemente aber auch über *Cascading Style Sheets* (CSS) [W3C2005e] definieren, wie man es bei HTML über CSS-Definitionen gewohnt ist.

Stylesheet-Angaben können entweder im Element direkt über das `style`-Attribut oder indirekt über das `class`-Attribut angegeben werden. Die CSS-Definitionen werden dabei in das `defs`-Element geschrieben oder in eine externe Datei ausgelagert, auf die im SVG-Dokument referenziert wird.

Die Elemente können sehr variabel formatiert werden. Die einfachste Möglichkeit besteht in einer Farbangabe für die Umrandung oder Füllung eines Objekts. Die Farbangabe erfolgt im sRGB-Standard über Hexadezimalwerte oder über

festgelegte Farbworte (black, white, green, ...). Darüberhinaus lassen sich alle erdenklichen Eigenschaften festlegen. Der Umrandung oder der Füllung lässt sich etwa ein Transparenzwert zuweisen, Linien können in der Dicke, Strichlierungsart, Darstellung in den Stützpunkten etc. variiert werden.

Beispiel:

```
<rect x="0" y="0" width="100" height="100"
  fill="#00FF00" stroke="#0000FF" stroke-width="2"
  stroke-linecap="round" />

<rect x="0" y="0" width="100" height="100"
  style="fill:#00FF00; stroke:#0000FF; stroke-width:2;
  stroke-linecap:round; />
```

Im Beispiel wird zweimal ein Rechteck definiert, das gelb gefüllt ist und eine zwei Pixel starke blaue Umrandung hat. Mit dem Attribut `stroke-linecap` kann angegeben werden, ob die Ecken der Umrandung abgerundet oder abgeschnitten werden sollen. Die beiden Rechtecke sind vollkommen gleich, weil sie die selben Attribute besitzen. Im ersten Fall werden sie über SVG-Atribute angegeben, im zweiten Fall mit dem `style`-Attribut über CSS.

Es werden in SVG lineare und radiale Farbverläufe unterstützt, die mit beliebigen Zwischenwerten versehen werden können. SVG verfügt zudem über reichhaltige Filtereffekte, die auf SVG-Elemente und Objekte angewendet werden können wie z.B. Unschärfe, Farbveränderungen oder Beleuchtung. Da allerdings die Implementation der Filter sehr schwierig ist, gibt es noch keinen SVG-Viewer, der alle Filter unterstützt, die in der SVG-Spezifikation definiert sind.

Als Beispiel ist in Abbildung 3.4 ein Button abgebildet, der durch einen Kreis definiert ist. Der obere weiße Bereich wird mit einer Ellipse erstellt. Die restlichen Effekte sind durch Farbverläufe und Filter realisiert, die dem Button ein sehr plastisches Erscheinungsbild geben.

Zusätzlich zu den Formatierungen lassen sich Elemente und Elementgruppen beschneiden und maskieren. Als Masken können dabei beliebige andere Elemente definiert werden.

3.6.5 Animation

Über CSS und ECMAScript können in HTML Elemente animiert werden. Diese Möglicheit besteht bei SVG ebenfalls. Allerdings müssen für eine solche Animation die Zwischenbilder einzeln berechnet werden, etwa durch eine Funktion, die in

Abbildung 3.4: Filter und Verläufe

kurzen Intervallen aufgerufen wird. SVG bietet eine weitere Möglichkeit der Animation, bei der die Zwischenbilder vom SVG-Viewer selbst durch Interpolation erzeugt werden.

In Flash werden Animationen über sogenannte “Key Frames” gesteuert, die zu bestimmten Zeitpunkten feste Werte vorgeben. In SVG besteht eine solche Möglichkeit nicht. Jedes animierte Objekt hat seinen eigenen zeitlichen Ablauf, so dass die Zwischenbildberechnung unabhängig von anderen Objekten erfolgen kann.

Eine Animation wird definiert über Animationsstart, Animationsdauer oder -ende, Anzahl der Wiederholungen und Animatonsart. Je nach Objekttyp können unterschiedliche Parameter animiert werden, z.B.: Farbwert, Position, Größe, Transformation, u.v.m. Im Allgemeinen sind das alle Eigenschaften eines Elements, die sich über Attributwerte oder Stylesheet-Angaben setzen lassen, solange sie sich sinnvoll animieren lassen. Dazu wird das `animate`-Element verwendet.

Beispiel:

```
<circle id="c1" cx="100" cy="100" r="100" fill="red" />
<animate xlink:href="c1" begin="2s" dur="4s"
   attributeName="r" from="100" to="200" fill="freeze" />
```

Das Beispiel definiert einen rot gefüllten Kreis mit dem Radius 100 Pixel. Nach zwei Sekunden beginnt eine Animation, die den Radius auf 200 Pixel vergößert. Diese läuft vier Sekunden lang. Wegen des Attributs `fill="freeze"` behält der Kreis hinterher die neue Größe bei.

Für den Fall, dass ein Attributwert verändert werden soll, ohne den Übergang zu animieren, bietet SVG das `set`-Element. Es benötigt neben dem Attributnamen und neuen Attributwert einen Zeitpunkt und die Dauer der Änderung.

Beispiel:

```
<circle id="c1" cx="100" cy="100" r="100" fill="red" />
<set xlink:href="c1" begin="2s" dur="4s"
   attributeName="fill" to="blue" />
```

Das `circle`-Element definiert wieder einen roten Kreis. Dieser wird durch das `set`-Element nach zwei Sekunden für vier Sekunden blau eingefärbt. Danach wird wieder die ursprüngliche rote Füllfarbe gesetzt.

Für Bewegungsanimationen stehen im `animateMotion`-Element verschiedene Animationsarten zur Verfügung, die über das Attribut `calcMode` angegeben werden. SVG kennt vier verschiedene Attributwerte:

- `discrete`
  Bei der diskreten Animation werden nur Zwischenbilder in den Punkten berechnet, die den Animationspfad beschreiben. Das bedeutet, dass das animierte Objekt zwischen den Pfadpunkten springt.

- `paced`
  Eine Animation, die mit dem Attributwert `paced` erstellt wird, erzeugt eine Bewegung, die über den ganzen Pfad eine konstante Geschwindigkeit besitzt. Im Gegensatz zur diskreten Animation werden die Positionen der Zwischenbilder auf dem Pfad interpoliert.

- `linear`
  Eine lineare Animation verhält sich ähnlich wie die vorherige, unterscheidet sich aber von ihr dadurch, dass die Animation auf den Teilpfaden jeweils eine konstante Geschwindigkeit besitzt, die Geschwindigkeiten sich zwischen den Pfadstücken aber unterscheiden können.

- `spline`
  Bei dieser Art der Interpolation der Zwischenbilder wird die Geschwindigkeit der Bewegungsanimation über ein zusätzliches Attribut gesteuert. Über das `keySplines`-Attribut kann eine Bézierkurve 3. Grades angegeben werden, die den zurückgelegten Weg des animierten Objekts in Abhängigkeit der Zeit angibt. Wie eine Bézierkurve im `keySplines`-Attribut definiert wird, wird im Kapitel 8.8.4 beschrieben, das die Erzeugung der Zuganimationen erläutert.

3.6.6 Interaktion mit ECMAScript

Eine SVG-Grafik kann durch Skriptsprachen wie ECMAScript während der Anzeige manipuliert werden. Es lassen sich Attribute verändern, oder ganze Elemente erzeugen oder löschen. Das Adobe SVG-Plugin, welches in der Applikation

verwendet wird, unterstützt beispielsweise unter Windows auch VBScript. In der Applikation wurde ECMAScript verwendet, um plattformunabhängig zu bleiben.

ECMAScript wird entweder, wie oben beschrieben, im `defs`-Element eingefügt, oder aber in einer externen Textdatei definiert, auf die im `defs`-Bereich referenziert wird. Um in einer SVG-Grafik Interaktionen mit dem Benutzer zu ermöglichen, können ECMAScript-Funktionen ereignisgesteuert aufgerufen werden. Jede Benutzereingabe, beispielsweise eine Bewegung oder das Klicken mit der Maus, erzeugt ein Event. Damit ein SVG-Objekt auf ein Event reagieren kann, muss es einen entsprechenden Event-Handler enthalten, der eine ECMAScript-Funktion aufruft.

Beispiel:

Im folgenden SVG-Element wird ein Kreis definiert, der beim Anklicken die ECMAScript-Funktion `move(evt)` ausführt. Dazu wird im `circle`-Element der Event-Handler `onclick` als Attribut definiert.

```
<circle id="c1" cx="100" cy="100" r="100" fill="red"
  onclick="move(evt);" />

  ...
  <defs>
    <script type="text/ecmascript">
    <![CDATA[
...

function move(evt)
{
  var circle = document.getDocumentElement().getElementById('c1');
  circle.setAttribute('cx', 'evt.clientX');
  circle.setAttribute('cy', 'evt.clientY');
}

...
    ]]>
    </script>
    ...
  </defs>
  ...
```

Die `move(evt)`-Methode liest die Koordinaten des angeklickten Punktes aus und verschiebt den Mittelpunkt des Kreises dorthin, indem die Koordinaten in den Attributen des `circle`-Elements neu gesetzt werden.

Das Wurzelelement `svg` einer SVG-Grafik kann als einziges den Event-Handler `onload` definieren. Wenn ein SVG-Dokument vollständig geladen wurde, wird einmalig ein Event ausgelöst, welches dieses Ereignis mitteilt. Üblicherweise wird mit dem `onload`-Handler eine Initialisierungsfunktion aufgerufen, die das `document`-Element des DOM in einer globalen Variable speichert, um spätere Zugriffe auf Teile des DOM zu erleichtern.

3.6.7 SVG-Viewer

SVG ist ein relativ junges Format, daher stehen viele Entwicklungen, die sich mit der Anzeige von SVG-Dokumenten befinden, noch am Anfang. Inzwischen exisitieren allerdings eine Reihe von Projekten, die das Ziel verfolgen, SVG anzuzeigen. Sie sind entweder als Plugins für Browser oder als eigenständige Viewer realisiert. Plugins gibt es derzeit von Abobe und Corel. Unter den Viewern hat das CSIRO SVG Toolkit den größten Funktionsumfang. Es wird allerdings nicht mehr weiterentwickelt und ist in Bezug auf Animation und Interaktion auf Grund der verwendeten Programmiersprache Java nicht sehr performant. Ein weiterer SVG-Viewer in Java ist im Batik SVG Toolkit von Apache enthalten [Apac2004b]. Batik befindet sich noch in der Entwicklung, derzeit wird in der Version 1.5.1 weder Animation noch Interaktion unterstützt. Die statischen SVG-Elemente werden aber nahezu vollständig unterstützt.

Bei vielen anderen SVG-Viewern haben die Entwickler ebenfalls noch vielfach Probleme, Animationen und Interaktion über Skriptsprachen in ihren Projekten umzusetzen. Auch sind beispielsweise die Filtermöglichkeiten nur sehr schwer umzusetzen. Eine Übersicht über SVG-Viewer und ihren Entwicklungsstand gibt [W3C2005d].

Möchte man aber viele der Features von SVG nutzen, empfiehlt sich daher für die Darstellung im Browser derzeit das SVG-Plugin von Adobe, erhältlich unter [Ado2005]. Im Vergleich zu anderen SVG-Viewern unterstützt das Plugin die meisten Eigenschaften von SVG und ist zudem sehr performant. Es liegt derzeit für alle gängigen Plattfomen in Version 3 vor, für Windows existiert bereits eine beta-Version mit Versionsnummer 6, die sich aber nur in Details von der Version 3 unterscheidet.

Unter den Web-Browsern kann zur Zeit noch keiner SVG nativ anzeigen. Derzeit ist zwar vom Mozilla-Browser eine SVG-Version erhältlich, sie befindet sich allerdings noch in Entwicklung und unterstützt derzeit nur eine Teilmenge der SVG-Elemente.

Microsoft plant, für 2006 den Nachfolger seines Betriebssystems Windows XP mit der internen Bezeichnung Longhorn zu veröffentlichen. Für eine skalierbare Darstellung der grafischen Benutzerberfläche entwickelt Microsoft derzeit mit *Windows Vector Graphics* (WVG) ein Format, das fast identisch mit SVG ist

und sich nur in Details von SVG unterscheiden soll. Dabei soll es sich um kleine Unterschiede im DOM handeln, um besser mit dem Betriebssytem kommunizieren zu können. Es ist aber auch davon auszugehen, dass der Microsoft Internet Explorer, mit Abstand der weitverbreitetste Webbrowser, WVG nativ darstellen kann. Ob der Internet Explorer dann auch SVG ohne Plugin anzeigen kann, bleibt abzuwarten, ebenso wie die Auswirkungen auf die allgemeine Akzeptanz von Vektorgrafikformaten im Internet [Hei2003].

4 Bézierkurven

Eine Bézierkurve n-ten Grades wird auf einem Intervall $[t_1, t_2]$ durch $n+1$ Stützpunkte P_i mit $i \in \{0, \ldots, n\}$ definiert:

$$P(t) = \sum_{i=0}^{n} B_{i,n} \cdot P_i, \;\; t \in [t_1, t_2] \tag{1}$$

Der Punkt P_i wird mit Hilfe eines Bernstein-Polynoms gewichtet.

$$B_{i,n}(t) = \binom{n}{i} \cdot \frac{1}{(t_2 - t_1)^3} \cdot (t - t_1)^i \cdot (t_2 - t)^{3-i}, \;\; i = 0, \ldots, n \tag{2}$$

Einige Eigenschaften der Bernsteinpolynome:

- Alle Bernsteinpolynome sind positiv auf $[t_1, t_2]$:
  $B_{i,n}(t) > 0 \;\forall t \in [t_1, t_2]$
- Die Summe aller Bernsteinpolynome ist 1:
  $\sum_{i=0}^{n} B_{i,n}(t) = 1$
- Die Bernsteinpolynome sind paarweise symmetrisch:
  $B_{i,n}(t - t_1) = B_{n-i,n}(t_2 - t)$

Beispiel: $n = 3$

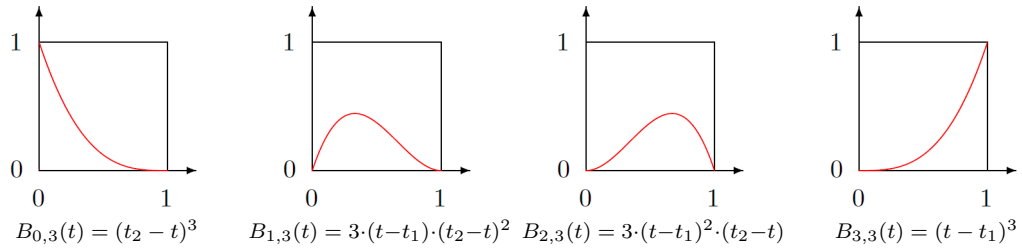

Abbildung 4.1: Verlauf der Bernsteinpolynome für $n = 3$

Eine Bézierkurve dritten Grades beginnt im ersten Stützpunkt und endet im letzten, den beiden sogenannten Ankerpunkten. Durch den zweiten und dritten Stützpunkt verläuft die Kurve nicht, ihr Verlauf wird aber durch die zwei sogenannten Kontrollpunkte beeinflusst. Die Bézierkurve beginnt im Stützpunkt P_0 tangential zur Geraden $\overline{P_0P_1}$ und endet im Punkt P_3 tangential zur Geraden $\overline{P_2P_3}$.

Diese Eigenschaft kann ausgenutzt werden, wenn man mehrere Bézierkurven aneinanderhängen möchte. Indem man den Endpunkt einer Bézierkurve P_3 als Anfangspunkt Q_0 der folgenden Bézierkurve identifiziert und die beiden angrenzenden Kontrollpunkte so wählt, dass die drei Punkte $P_2, P_3 = Q_0, Q_1$ kollinear sind, ergibt sich bei $P_3 = Q_0$ ein stetig differenzierbarer Übergang zwischen den beiden Bézierkurven (siehe Abbildung 4.2).

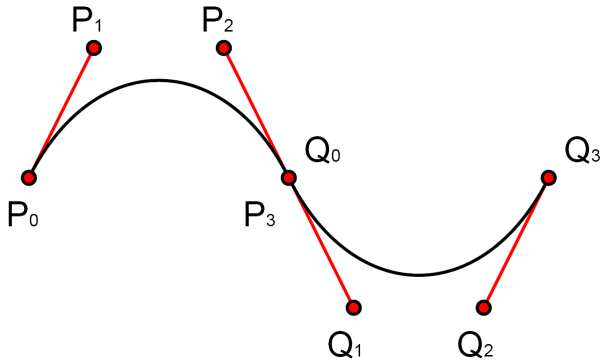

Abbildung 4.2: Bézierkurven 3. Grades mit stetig differenzierbarem Übergang

4.1 Bézierkurve als Weg-Zeit-Funktion in SVG

Eine Bézierkurve wird über einen Parameter t und mehrere Stützpunkte definiert. Parametrisiert man die y-Koordinate der Stützpunkte durch die x-Koordinate, ergibt sich eine Funktion, die die y-Koordinate in Abhängigkeit von der x-Koordinate berechnet.Identifiziert man die x-Koordinate mit t, erhält man eine Funktion in Abhängigkeit vom Parameter t.

In SVG können diese Art von Bézierkurven verwendet werden, um nicht nur auf lineare Bewegungsanimationen beschränkt zu sein (siehe Kapitel 3.6.5). Man definiert mit einer parametrisierten Bézierkurve eine Weg-Zeit-Funktion $s(t)$ und kann damit den zum Zeitpunkt t zwischen Startpunkt t_{start} und Endpunkt t_{end} zurückgelegten Weg angeben, an dem sich ein animiertes Objekt befindet. Der Gradient dieser Weg-Zeit-Funktion ist die Geschwindigkeit zum Zeitpunkt t.

Um eine Bézierkurve 3. Grades zu definieren, werden vier Stützpunkte benötigt. Start- und Endpunkt definieren ein Rechteck, in dem auch die gesamte Kurve liegen muss. Läge die Kurve beispielsweise kurzzeitig oben ausserhalb des Rechtecks, würde das bedeuten, dass das animierte Objekt beim Austritt aus dem Rechteck den Endpunkt des Animationspfades erreicht hätte und über das Ziel hinausschießen würde. Da hinter dem Endpunkt des Pfades aber keine Pfadinformationen mehr verfügbar sind, ist eine Animation für den Kurvenbereich außerhalb des Rechtecks nicht möglich. Damit die Kurve innerhalb des Rechtecks bleibt, ist

es daher erforderlich, dass auch die Kontrollpunkte im Rechteck liegen. Da eine Bézierkurve innerhalb der konvexen Hülle des Linienzuges der Stützpunkte verläuft, reicht dieses Kriterium aus.

In Abbildung 4.3 sind als Beispiel drei verschiedene Bézierkurven mit ihren Kontrollpunkten angegeben. Bei der ersten Kurve handelt es sich zum Vergleich um eine lineare Bewegung, die durch einen gradlinigen Kurvenverlauf dargestellt wird. Die zweite Kurve gibt eine Bewegung wieder, die im Vergleich zum Ende mit der dreifachen Geschwindigkeit startet. Das lässt sich an der dreifachen Steigung in den Endpunkten ablesen. Das rechte Beispiel startet mit einer Geschwindigkeit von 0 und beschleunigt dann beständig auf eine Geschwindigkeit, die ihr Maximum am Ende annimmt.

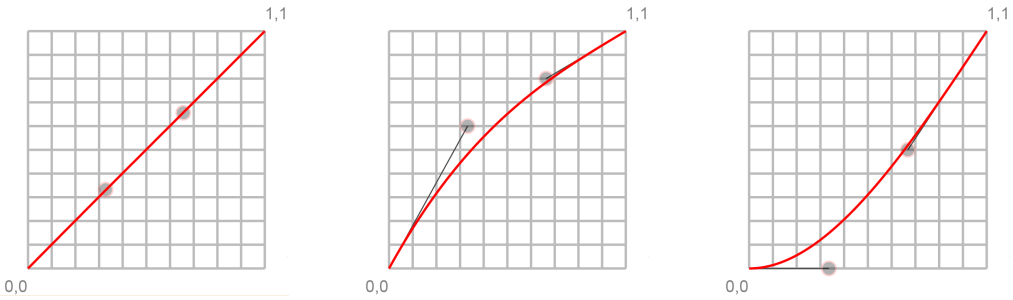

Abbildung 4.3: Beispiele für Bézierkurven

Da mit der Geschwindigkeit die Steigung benachbarter Bézierkurven in den Berührungspunkten immer gleich ist, stimmt auch die Tangente beider Bézierkurven überein. Damit ist die oben beschriebene Kollinearität erfüllt. Also ergibt sich zwischen zwei Bézierkurven ein stetig differenzierbarer Übergang, der anschaulich bedeutet, dass die Geschwindigkeit stetig ist, also keinen Sprung macht.

5 Standard Widget Toolkit

Das *Standard Window Toolkit* (SWT) ist ein Toolkit, das von IBM zur Gestaltung von Oberflächen in Java-Programmen entwickelt wurde. Es stellt damit eine Alternative zu den üblicherweise verwendeten Toolkits AWT und Swing dar, die in der Java-Laufzeitumgebung enthalten sind. SWT unterscheidet sich in vielen Aspekten von AWT und Swing, um einige Nachteile zu beseitigen.

Das *Abstract Window Toolkit* (AWT) ist von Sun mit dem Ziel entwickelt worden, die Menge von Oberflächenelementen, sie sogenannten *Widgets* in Java zur Verfügung zu stellen, die auf allen Plattformen vorhanden ist. Dazu werden die jeweiligen Widgets des Betriebssystems verwendet, die über native Methoden angesprochen werden. Allerdings sind die in AWT zur Verfügung stehenden Oberflächenelemente in ihrer Anzahl und ihrem Funktionsumfang sehr eingeschränkt, da das AWT seit 1996 im Funktionsumfang nicht mehr weiterentwickelt wurde. Statt AWT verwenden Entwickler daher häufig Swing.

Swing ist eine Weiterentwicklung von AWT, mit dem Ziel, diese Beschränkungen aufzuheben. Vielfach sind die Klassen von AWT wiederverwendet und abgeleitet worden. Ein Hauptunterschied zu AWT liegt allerdings darin, dass Swing ohne native Betriebssystemaufrufe arbeitet und die Oberflächenelemente selbst zeichnet. Dadurch können viele plattformspezifische Oberflächenelemente hinzugefügt werden, da Swing nicht mehr von den Elementen der Plattform abhängig ist und diese Elemente selbst erzeugt. Der Funktionsumfang von Swing ist dadurch viel größer als bei AWT. Allerdings hat dieses Vorgehen auch Nachteile. Dadurch, dass Swing sich nicht der Oberflächenelemente des Betriebssystems bedient, müssen für jede Plattform die Elemente in Swing nachgebaut werden, was nicht immer zufriedenstellend gelingt. Obwohl Swing zum Beispiel für Windows sowohl an die klassische Oberfläche als auch an die neue XP-Oberfläche Luna angepasst ist, merkt man einer Swing-Applikation immernoch ihre Herkunft an. Ein Hauptproblem von Swing ist aber seine Geschwindigkeit. Da alle Widgets von Swing durch Java-Befehle gezeichnet werden, wird eine mit Swing realisierte Oberfläche sehr langsam aufgebaut und reagiert sehr viel träger im Vergleich zu einer Oberfläche, die mit nativen Zeichenoperationen gezeichnet wird.

SWT verfolgt einen ähnlichen Ansatz wie AWT, die Widgets des jeweiligen Betriebssystems zur Darstellung der Oberfläche zu verwenden. Wenn ein Widget nicht vorhanden ist, wird es aus anderen Teilen zusammengesetzt oder selbst erzeugt. Im Gegensatz zu AWT enthält SWT die gesamte Palette von Widgets, die in modernen Oberflächen verwendet werden. Die Benutzung der Betriebssystemelemente reduziert einerseits den Ressourcenverbrauch einer Oberfläche, andererseits ist die Geschwindigkeit sehr hoch. Dadurch erzeugt eine in SWT gestaltete Oberfläche auf allen Plattformen den Eindruck, dass es sich um eine native Applikation handelt. Einzig der Java-typische hohe Speicherverbrauch lässt noch erkennen, dass es sich um ein Java-Programm handelt.

Um die Unterschiede der verschiedenen Plattformen zu berücksichtigen, muss die SWT-API für jedes Betriebssystem einzeln implementiert werden. Die Implementationen selbst sind in Java erstellt. Lediglich die Betriebssystemaufrufe an die nativen Oberflächenelemente, die meist in C erfolgen müssen, werden über das *Java Native Interface* (JNI) realisiert und werden in eine plattformabhängige Wrapper-Bibliothek ausgelagert. Das ermöglicht, ein Programm, welches beispielsweise unter Windows läuft, ohne Änderung am Quellcode unter Linux laufen zu lassen, da die SWT-API Java-seitig unverändert bleibt. Man muss lediglich die SWT-Implementation von Windows, die aus dem Archiv "swt.jar" und der Bibliothek "swt-win32-3116.dll" besteht, gegen die Linux-Implementation, der Linux-Version des Archivs "swt.jar" und der Bibliothek "libswt-3116.so", austauschen. Derzeit wird SWT auf Windows, Linux mit Motif (Gnome) und GtK (KDE), MacOS und wichtigen Unix-Varianten unterstützt.

SWT hat seinerseits auch einige Nachteile. Im Gegensatz zu Swing, das mit dem Ziel der einfachen Implementierbarkeit entwickelt wurde, ist die Benutzung von SWT um einiges aufwändiger. SWT unterstützt beispielsweise keine Garbage Collection, so dass sich der Entwickler um die Freigabe nicht mehr benötigter Ressourcen kümmern muss. Dies wurde von den Entwicklern mit dem vereinfachten Debugging und der Betriebssystemnähe begründet. In der Praxis macht sich dieser Nachteil auf Grund der Architektur von SWT kaum bemerkbar, da z.B. alle Elemente in einem Unterfenster freigegeben werden, sobald das Unterfenster selbst freigegeben wird. Die Nähe zum Betriebssystem findet sich zum Beispiel darin wieder, dass der Entwickler im Gegensatz zu Swing die Haupt-Eventschleife selbst erzeugt und startet.

Das SWT ist in der Entwicklung schon sehr weit fortgeschritten. Das beste Beispiel dafür ist die integrierte Entwicklungsumgebung Eclipse, bei der die Oberfläche mit SWT realisiert wurde. Allerdings befindet sich SWT noch in stetiger Weiterentwicklung, was sich auch daran erkennen lässt, dass einem Entwickler kaum die Möglichkeit gegeben wird, Widget-Klassen abzuleiten, um spätere Änderungen an den Klassen nicht unmöglich zu machen.

5.1 Das Browser-Widget

Ein wesentlicher Grund für die Benutzung des SWT ist das Browser-Widget. Es stellt dem Entwickler den jeweiligen System-Webbrowser als Widget in seiner Applikation zur Verfügung. Unter Windows ist das der Internet Explorer, unter MacOS X Safari. Unter Linux muss eine spezielle Mozilla-Version installiert sein.

Das Browser-Widget reagiert allerdings auf unterschiedlichen Plattformen nicht einheitlich, weil auf den jeweiligen Plattformen unterschiedliche System-Webbrowser zur Verfügung stehen. Ein weiterer Unterschied, der auch auf verschiedenen Installationen der gleichen Plattform auftreten kann, ist der Umfang der

installierten Plugins im Webbrowser. Über Plugins kann man einem Browser-Widget die Fähigkeit hinzufügen, SVG-Dateien anzuzeigen, indem man das SVG-Plugin von Adobe oder Corel installiert. Durch andere zusätzliche Plugins kann man das Browser-Widget in seinen multimedialen Anzeigemöglichkeiten stark erweitern und erhält so ein sehr mächtiges Viewer-Objekt für seiner Java-Applikation.

5.2 Plattformunabhängigkeit

Die Oberflächenelemente der unterschiedlichen Plattformen sind in SWT sehr gut integriert worden, abgesehen von den plattformspezifischen Eigenheiten hat die Applikation ein weitgehend einheitliches Erscheinungsbild. Unterschiede gab es nur im Browser-Widget, die sich vor Allem im ECMAScript bemerkbar machen. Adobe verwendet standardmäßig die ECMAScript-Implementation des Webbrowsers. Steht diese nicht zur Verfügung, wird das ECMA-Script verwendet, das vom Plugin selbst mitgebracht wird. Es enthält die JavaScript-Implementation aus dem Mozilla-Projekt, die in den Browsern Mozilla und Firefox enthalten ist. Der Adobe SVG-Viewer verfügt über die *Adobe-SVG-Extensions*, über die man ein Attribut in das `svg`-Element einfügen kann, mit dem angegeben werden kann, welche ECMAScript-Implementation das Plugin verwenden soll.

Die Applikation wurde unter Microsoft Windows entwickelt und läuft dort fehlerfrei. Die SVG-Datei lässt sich mit Internet Explorer 6.0, Mozilla Firefox 1.0 und Opera 7.51 betrachten, wobei jeder Browser seine eigene ECMAScript-Implementation verwendet. Unter Linux läuft die Applikation ebenfalls, allerdings kann die SVG-Datei auf Grund von Fehlern in der ECMAScript-Behandlung nicht korrekt angezeigt werden. Das ECMAScript des zu Grunde liegenden Mozilla-Browsers ignoriert eine Transformationsanweisung, die nachträglich über einen ECMAScript-Befehl eingefügt wird. Das Zooming versagt dadurch seinen Dienst.

6 Eingabedaten

Die zur Darstellung des Modells benötigten Daten lassen sich in zwei Teile aufteilen. Der erste Teil besteht aus dem Gleisplan des darzustellenden Bahnhofs mit den Koordinaten der Gleise, Weichen und Signale, sowie den Knoten des Modells. Der zweite Teil umfasst die Fahrplaninformationen der Züge. Die Aufteilung ergibt sich aber auch ganz natürlich aus den vorliegenden Daten des Optimierungsprogramms. Der erste Teil ist dabei ebenfalls Bestandteil der Eingabedateien des Optimierungsprogramms und wird zur Erzeugung der Ausgabedatei benötigt, die den Fahrplanablauf mit allen Signaländerungen enthält und damit den zweiten Teil abdeckt. Sie beinhaltet zusätzlich auch die Zuglängen.

6.1 Gleisplandatei

Die Eingabedaten des Optimierungsprogramms enthalten die Abstände der Knoten nur in einer Dimension, da die Abstände der Gleise vernachlässigbar klein sind und somit kaum Einfluss auf die Geschwindigkeit eines Zuges nehmen, wenn über eine Weiche das Gleis gewechselt wird. Für eine Visualisierung ist eine eindimensionale Betrachtung aber nicht ausreichend, da der Applikation dadurch wichtige Informationen über die Lage der Gleise und Weichen fehlen. Daher handelt es sich bei der Gleisplandatei um eine Datei, die unabhängig vom Optimierungsprogramm für die Applikation für jeden darzustellenden Bahnhof erstellt werden muss.

Als Quelle dient hauptsächlich eine Eingabedatei des Optimierungsprogramms, eine Textdatei, die eine Liste aller Knoten mit Nachfolgern im Gleisplan und den Abstand zu den Nachfolgern enthält. Die Abstände wurden in x-Richtung abgetragen. Die fehlenden Informationen der Knoten in y-Richtung und die genaue Lage der Abbiegepunkte zwischen den Knoten konnten aus zwei abstrakten Lageplänen des Gleisplans entnommen werden. Da die Geschwindigkeitsberechnungen nur in einer Dimension, der x-Richtung, erfolgen, konnte der Abstand der Gleise für eine sinnvolle Visualisierung frei gewählt werden, für Amsterdam-Schiphol beträgt er umgerechnet 50m.

Als Format für die Gleisplandatei fiel die Wahl auf XML, wie in Kapitel 3.1 begründet. Die XML-Datei muss der DTD `station.dtd` genügen, die den Aufbau der Gleisplandatei definiert. Dieser Aufbau wird im Folgenden näher erläutert.

Der Gleisplan besteht aus einer Liste aller Knoten des Optimierungsmodells.

```
<!ELEMENT nodes (node+)>
```

Ein Knoten hat als Attribute eine eindeutige ID, die Koordinaten in der Visualisierung und eine Typnummer, die in der Visualisierung durch eine Farbe

wiedergegeben wird. Handelt es sich um einen Knoten, über den Züge ein- oder ausfahren, enthält ein Knoten zusätzlich das Attribut `end`. Ein Knoten kann zudem beliebig viele Nachfolger-Elemente besitzen.

```
<!ELEMENT node (successor*)>
<!ATTLIST node
  nr ID #REQUIRED
  type (0|1|2|3|4) #REQUIRED
  xcoord CDATA #REQUIRED
  ycoord CDATA #REQUIRED
  end (-1|1) #IMPLIED
>
```

Um einen Nachfolger zu definieren, wird dem Nachfolger-Element über den Attributtyp `IDREF` ein existierender Knoten zugewiesen. Dabei handelt es sich zwangsweise um ein Knoten-Element, da nur diese Elemente eine `ID` besitzen. Falls ein Gleisabschnitt über eine Weiche das Gleis wechselt, können dem Nachfolger-Element noch beliebig viele Abbiegepunkte hinzugefügt werden.

```
<!ELEMENT successor (turn*)>
<!ATTLIST successor
  nr IDREF #REQUIRED
>
```

Ein Abbiegepunkt wird durch seine Koordinaten definiert.

```
<!ELEMENT turn EMPTY>
<!ATTLIST turn
  xcoord CDATA #REQUIRED
  ycoord CDATA #REQUIRED
>
```

Die Gleisplandatei wird in der Applikation von einem Parser eingelesen und in Form eines DOM-Trees als Objekt zur Verarbeitung bereitgestellt.

Beispiel:

Der Quellcode 6.1 enthält als Beipsiel einen einfachen Gleisplan mit drei Knoten, der in Abbildung 6.1 dargestellt ist. Knoten 1 und Knoten 3 sind Nachfolger von Knoten 2. Daraus folgt, dass auf diesem Gleisplan nur Zugbewegungen von rechts nach links stattfinden. Auf dem Pfad nach Knoten 1 liegen zusätzlich zwei Abbiegepunkte. Alle drei Knoten sind zudem Endknoten.

```
<?xml version="1.0" standalone="no" ?>
<!DOCTYPE nodes SYSTEM "station.dtd">

<nodes>
  <node nr="n2" type="1" xcoord="-500" ycoord="50" end="-1">
    <successor nr="n1">
      <turn xcoord="-640" ycoord="50" />
      <turn xcoord="-690" ycoord="100" />
    </successor>
    <successor nr="n3" dist="300" />
  </node>

  <node nr="n1" type="1" xcoord="-800" ycoord="100" end="1" />

  <node nr="n3" type="1" xcoord="-800" ycoord="50" end="1" />
</nodes>
```

Quellcode 6.1: Gleisplandatei mit drei Knoten

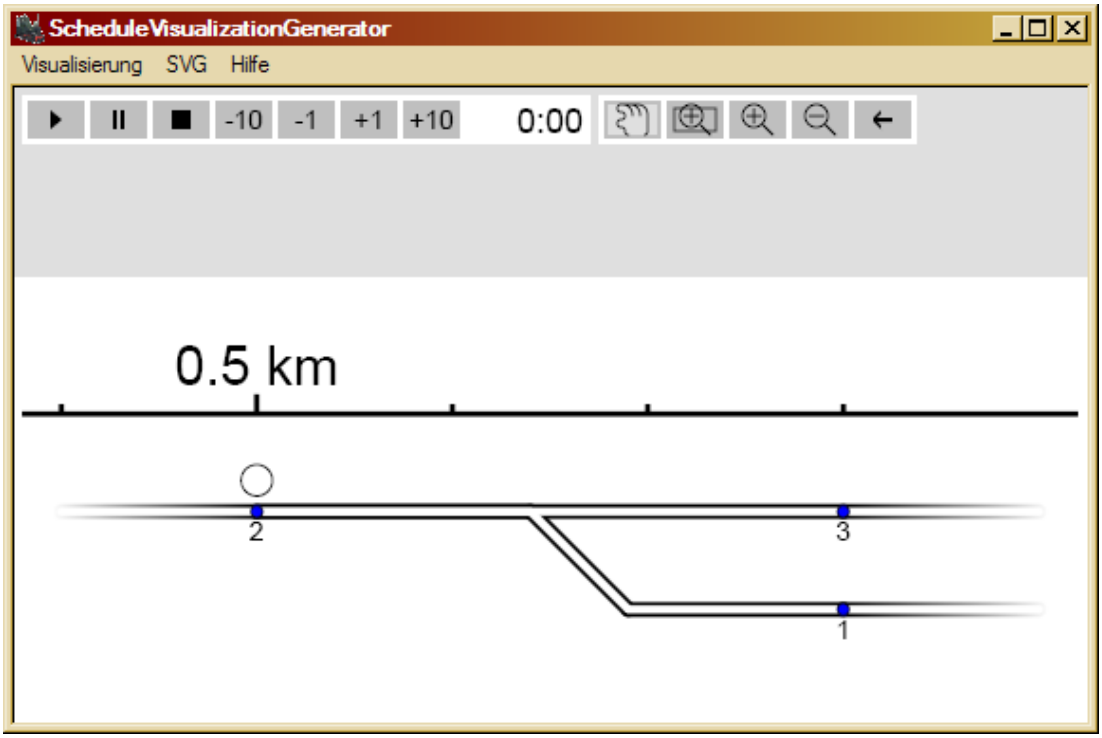

Abbildung 6.1: Gleisplan mit drei Knoten

6.2 Fahrplandatei

Die Fahrplandatei beschreibt den Fahrplan aller Züge. Dazu enthält sie für jeden Zug die Zeitpunkte und die Geschwindigkeit aller durchfahrenen Knoten. Außerdem enthält sie die Zeitpunkte, in denen Signale umgeschaltet werden und jeweils die Länge eines Zuges.

Die Fahrplandatei wird direkt vom Optimierungsprogramm erzeugt und liegt als einfache Textdatei vor. Sie enthält zeilenweise zu einem bestimmten Zeitpunkt entweder Zuginformationen oder nur Informationen zu Signaländerungen. Zusätzlich sind Zeilen enthalten, die für einen Zug seine Länge angeben. Die Einträge sind chronologisch geordnet und die Werte in einer Zeile durch Leerzeichen getrennt. Eine Unterscheidung der Zeilenart erfolgt über die erste Ziffer einer Zeile:

- Zeilentyp 0:

  `0 Zug# Knoten# Zeit Geschw. [Signalfarbe Knoten# [Standzeit]]`

  Zeilentyp 0 entält eine Zuginformation. Dabei wird wie oben bereits erwähnt, einem Zug ein Zeitpunkt und eine Geschwindigkeit in einem Knoten zugeordnet. Zusätzlich kann zum selben Zeitpunkt optional noch eine Signaländerung in einem anderen Knoten angegeben werden. Der letzte Parameter kann gesetzt werden, wenn der Zug in dem Knoten steht, d.h. die Geschwindigkeit beträgt 0 $\frac{km}{h}$, um die Standzeit des Zuges in dem Punkt anzugeben.

- Zeilentyp 1:

  `1 Zeit Knoten#`

  Da das Optimierungsprogramm für die Signalknoten nicht durchgängig Informationen über die Signalfarbe vorhält, wird in Zeilentyp 1 der Zeitpunkt angegeben, in welchem das Signal im entsprechenden Knoten auf weiß zurückgesetzt wird.

- Zeilentyp 2:

  `2 Zug# Zuglaenge`

  Um die Länge der Züge durch die Anzahl seiner Waggons modellieren zu können, werden sie in der Fahrplandatei angegeben. Für jeden Zug ist eine Zeile enthalten, die der Zugnummer seine Länge zuordnet.

Die Daten über die Züge und Signale werden in der Applikation zeilenweise gelesen und im `scheduleReader` gekapselt.

```
2  0 100
2  1 150
0  0 2  0.5 80.0  0 2
0  0 1  1.0 20.0  2 2
0  1 2  1.5 80.0  0 2
1  2.0 2
0  1 3  2.0 60.0
```

Quellcode 6.2: Fahrplan mit zwei Zügen

Beispiel:

In Quellcode 6.2 ist ein Beispielfahrplan für den Gleisplan aus Kapitel 6.1 angegeben. In den ersten beiden Zeilen steht die Zuglänge für zwei Züge. Zeile drei und vier enthalten den Fahrplan für den Zug mit der Nummer null. Er ist ist zum Zeitpunkt 0,5 Minuten mit der Geschwindigkeit 80 $\frac{km}{h}$ bei Knoten zwei und 0,5 Minuten später mit nur noch 20 $\frac{km}{h}$ bei Knoten eins. Der andere Zug mit der Nummer eins fährt nach 1,5 Minuten über Knoten zwei mit der Geschwindigkeit 80 $\frac{km}{h}$, wechselt dann das Gleis und hat zum Zeitpunkt 2 Minuten noch eine Geschwindigkeit von 60 $\frac{km}{h}$, als er Knoten drei passiert. Knoten zwei ist zudem ein Signalknoten. Nach 0,5 und 1,5 Minuten wird er jeweils für 0,5 Minuten auf grün gesetzt. Dazwiwschen zeigt er die Farbe rot. Die vorletzte Zeile löscht die Signalfarbe nach der zweiten Grünphase zum Zeitpunkt 2 Minuten.

7 Aufbau der Java-Applikation

Ein Überblick über die Klassenhierarchie der Applikation gibt das UML-Diagramm in Abbildung 7.1.

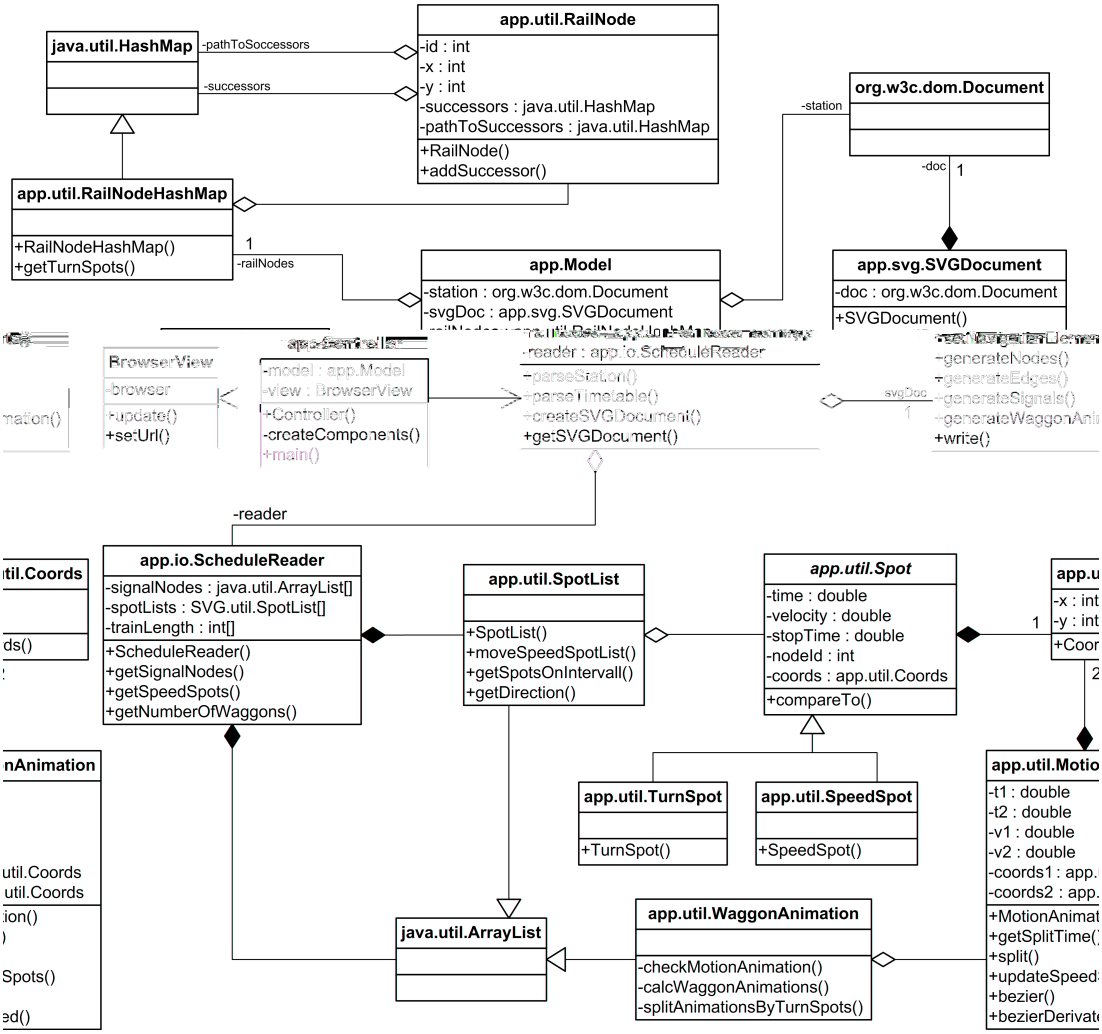

Abbildung 7.1: Klassendiagramm der Applikation

7.1 Model-View-Controller

Bei der Realisierung der Applikation wurde das Model-View-Controller Konzept angewandt. Der Hauptgrund hierfür liegt in der Trennung des Datenmodells von der Programmsteuerung und Anzeige der Daten. Beim klassischen MVC Konzept sind diese drei Schichten klar voneinander getrennt. Der Controller "kennt" die View und das Modell. Er steuert die Benutzereingaben und löst dadurch Änderungen der Daten im Model aus. Des Weiteren fragt er das Model nach seinen Daten und übergibt sie der View zur Anzeige. Die View und das Model benötigen dabei keine Kenntnis über die anderen Schichten (siehe Abbildung 7.2).

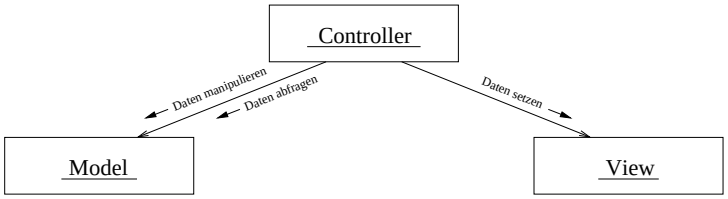

Abbildung 7.2: Model-View-Controller

Bei der vorliegenden Applikation wurde das MVC Konzept der völligen Trennung der Schichten ein wenig aufgeweicht, um das Designpattern Observer/Observable zu verwenden. Dabei wird ein Observer beim Observable angemeldet, um Änderungen im Observable verfolgen zu können. Eine Änderung im Observable wird den angemeldeten Observern über einen Methodenaufruf angezeigt. Auf diese Weise kann einer View, die das Observer-Interface implementiert, eine Zustandsänderung in der Datenbasis des Models, das dem Observable entspricht, mitgeteilt werden, um sie zu aktualisieren. Dabei wird der View auch eine Referenz des Observable-Objekts, also des Models übergeben, um für die Anzeige auf die Daten des Models zugreifen zu können (siehe Abbildung 7.3).

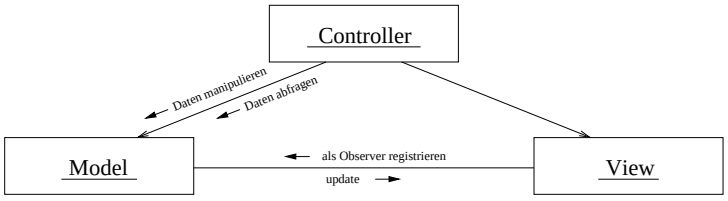

Abbildung 7.3: Model-View-Controller mit Observer-Pattern

Das *Model* enthält alle anfallenen Daten der Applikation. Dazu gehören neben den eingelesenen Eingabedaten wie `RailNodeHashMap` und `ScheduleReader` auch das erzeugte `SVGDocument`. Neben der Verwaltung der Daten führt das Model auch alle Operationen auf diesen aus. Dafür enthält das Model die Methoden um die Eingabedaten einzulesen, zu verarbeiten und das SVG-Dokument zu erstellen.

Die *View* besteht aus der `BrowserView`, die als innere Klasse im Controller realisiert ist. Sie ist als Observer beim Model registriert. Dadurch wird ihr mitgeteilt, wenn im Model ein neues SVG-Dokument erzeugt wird. In diesem Fall ruft das Model bei der BrowserView die Methode `update()` auf und übergibt sich dabei selbst in Form eines Parameters. Auf diese Weise erhält die BrowserView einen Verweis auf das Model und kann es mit der Methode `getUri()` nach der URI[3] der erzeugten SVG-Datei fragen, um sie anzuzeigen.

Der *Controller* enthält die `main`-Methode und ist zunächst für den Aufbau der Applikation und die Erzeugung der Benutzeroberfläche zuständig. Er definiert die Steuerung der Applikation. Dazu besitzt er die Kontrolle über die Eventschleife und enthält für die einzelnen Menüpunkte eine Methode, um die entsprechenden Aktionen auszuführen und ruft dabei auch das Models mit seinen Methoden auf.

Die Trennung der Schichten ist der gesamte Aufbau der Oberfläche im Controller gekapselt, die Funktionalität im Model. Durch Auswechseln des Controllers wäre es daher leicht möglich, ein kommandozeilenorientiertes Programm zu schreiben, welches das Model unverändert zur Erzeugung einer SVG-Datei verwenden könnte.

[3] *Uniform Resource Identifier*

8 Erzeugung des SVG-Dokuments

Neben der Anzeige ist die Erzeugung des SVG-Dokuments die zentrale Aufgabe der Applikation. Sie lässt sich in folgende Schritte einteilen:

1. Einlesen der Eingabedateien und Abfrage der Parameter
2. Erstellung der Knotenliste `nodeList` des Gleisplans
3. Erstellung des SVG-Grundgerüstes
4. Einfügen des ECMAScripts und der CSS-Definitionen
5. Generierung der Navigationsleiste
6. Generierung des Gleisplans aus der `nodeList`
7. Generierung der Signale mit ihren Signalabfolgen
8. Generierung der Animationen mit Hilfe des `scheduleReader` über Bézierkurven 3. Grades
9. Anzeige des SVG-Dokuments

Die Erzeugung des SVG-Dokuments findet im Model statt. Es enthält dafür drei Methoden. Die Methode `parseStation()` liest den Gleisplan ein, die Methode `parseTimetable()` den Fahrplan. Die dritte Methode `createSVGDocument()` enthält dann alle weiteren Schritte, die oben aufgeführt sind. Zur besseren Übersicht ist für diese Methode in Abbildung 8.1 ein Sequenzdiagramm abgebildet.

8.1 Einlesen der Eingabedateien und Abfrage der Parameter

Die Erzeugung des SVG-Dokuments wird in der Applikation über den Menüpunkt "SVG erzeugen..." ausgelöst. Zu Beginn werden jeweils über einen File-Dialog die beiden Pfade der einzulesenden Dateien abgefragt.

Das erste Dialogfenster verlangt nach dem Pfad der Gleisplandatei, die im XML-Format vorliegt. Sie wird mit Hilfe eines `DocumentBuilder` eingelesen. Dabei wird das XML-Dokument gegen die Gleisplan-DTD `station.dtd` validiert, um sicherzustellen, dass das Dokument im weiteren Programmablauf weiterverarbeitet werden kann.

Zur Fehlerbehandlung wird ein `StationErrorHandler` beim `DocumentBuilder` angemeldet. Der `StationErrorHandler` definiert für die drei Fehlermöglichkeiten

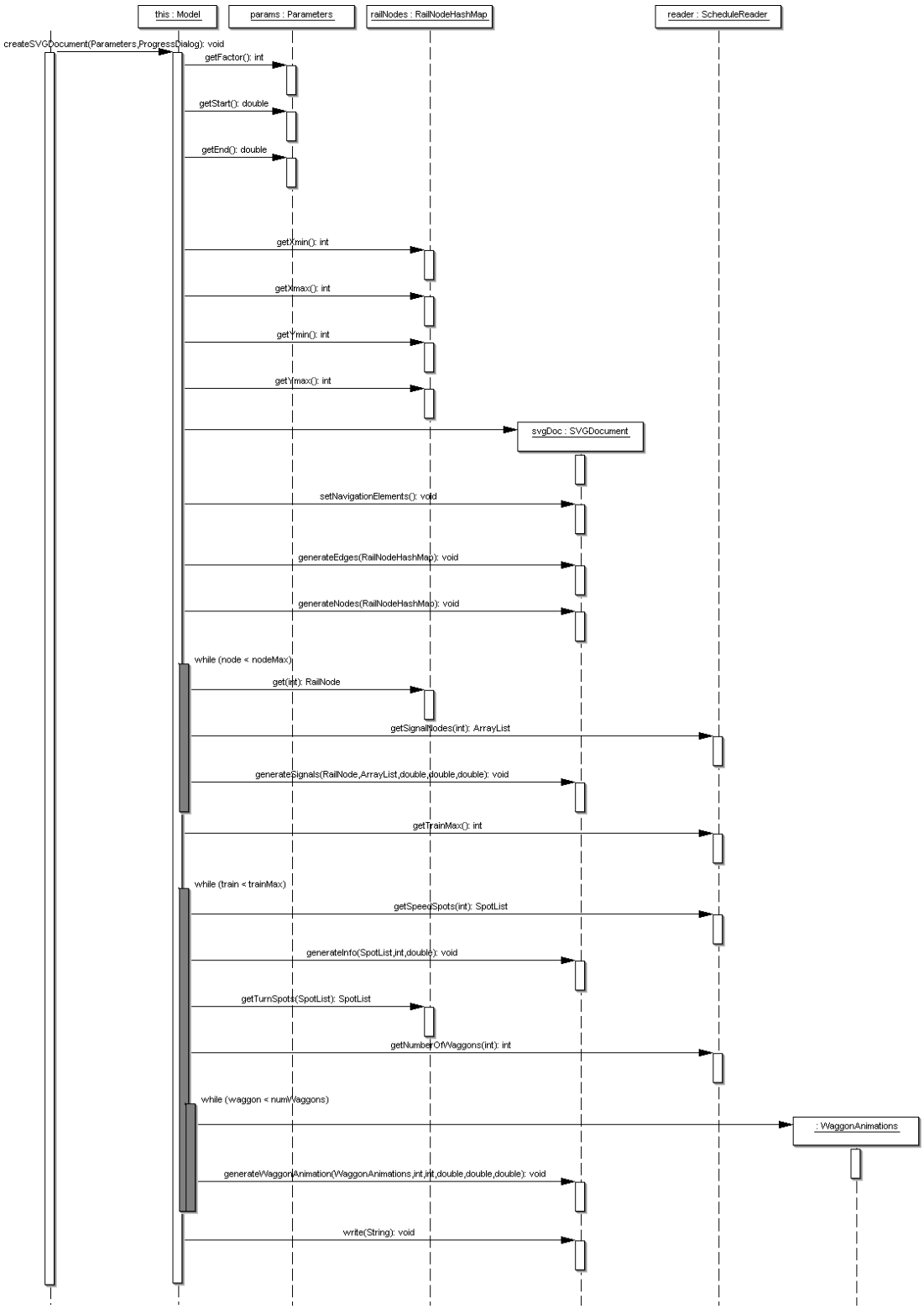

Abbildung 8.1: Sequenzdiagramm der Methode `createSVGDocument()`

```
public class StationErrorHandler extends DefaultHandler
{
  public static final class SAXWarningException
                                        extends SAXException
  {
    public SAXWarningException(String message)
    {
      super(message);
    }
  }
  ...
  public void warning(SAXParseException exception)
                                        throws SAXException
  {
    throw new SAXWarningException("Warnung!");
  }
  ...
}
```

Quellcode 8.1: Klasse `StationErrorHandler`

jeweils eine `Exception`, die eine Unterklasse von `SAXException` ist und die beim Aufruf der Methode geworfen wird. Der Codeausschnitt 8.1 zeigt dies am Beispiel einer `Warning`.

Das hat den Vorteil, dass der Fehler nicht innerhalb der Methode behandelt werden muss, sondern als Exception über die Kette der aufrufenden Methoden weitergereicht werden kann. Im `Controller`, der die Oberfläche erzeugt und die Eventschleife kontrolliert, wird als Reaktion auf eine `SAXException` ein Dialogfenster erstellt, das dem Benutzer den Fehler im XML-Dokument mitteilt.

Das zweite Dialogfenster fragt den Pfad der Datei ab, die den Fahrplan enthält. Sie wird zeilenweise gelesen. Eine Zeile wird über einen `StringTokenizer` in seine Bestandteile zerlegt. Die Knoten- und Zugnummern werden in `int`-Werte, Geschwindigkeitswerte und Zeitpunkte in `double`-Werte umgewandelt.

Die Zuginformationen in Zeilentyp 1 werden beim Einlesen für jede Zugnummer in einer eigenen `ArrayList` gespeichert. Dazu werden die Informationen für jeden Knoten in einem Objekt der Klasse `SpeedSpot` gekapselt. Da der chronologisch geordnete Fahrplan sequentiell gelesen wird, sind diese Listen, die die Fahrpläne für die einzelne Züge enthalten, ebenfalls chronologisch geordnet. Die Signalinformationen aus beiden Zeilentypen, in einem Objekt der Klasse `Signal` gekapselt, werden ebenfalls für jeden Signalknoten in einer `ArrayList` gespeichert. Auch diese sind chronologisch sortiert. Die in Zeilentyp 2 angegebenen Zuglängen werden über ein `int`-Array der Zugnummer zugeordnet. Die Listen der Zug- und

Signalinformationen sowie die Zuglängen sind im `ScheduleReader` enthalten und können über Methoden abgefragt werden.

In einem weiteren Dialogfenster werden weitere Parameter abgefragt, die für die Erzeugung der SVG-Datei nötig sind. Man kann sich zum Einen auf einen zeitlichen Ausschnitt der Animation beschränken, indem man einen Start- und Endzeitpunkt der Animation setzt. Dadurch wird der Umfang der SVG-Datei verringert, was sich positiv auf die Performanz der Anzeige auswirkt. Außerdem verringert es den Navigationsaufwand, wenn man nur an einem Ausschnitt der Animation interressiert ist. Zum Anderen kann man einen Faktor angeben, der die Geschwindigkeit der Animation steuert, um den Simulationsablauf in einer Art Zeitraffer darstellen zu können. Wenn man den Faktor so wählt, dass für einen Fahrplan mit einer Dauer von einer Stunde eine Animation erstellt wird, die fünf bis acht Minuten dauert, erhält man einen guten Kompromiss aus Übersichtlichkeit und Schnelligkeit der Animation. Der Faktor kann während der Anzeige nicht mehr verändert werden. Das liegt daran, dass SVG keine Möglichkeit bietet, etwa über ECMA-Script die Zeitachse zu stauchen oder zu strecken. Man könnte theoretisch die Zeitangaben in den `animation`-Elementen über ECMA-Script manipulieren, in der Praxis scheitert dies aber an der Menge der Animationen und damit am zu hohen Rechenaufwand.

8.2 Erstellen der Knotenliste

Nach dem Einlesen der Dateien liegt der DOM-Tree des Gleisplans vor. Daraus werden die Knoten ausgelesen und jeweils in einem Objekt der Klasse `RailNode` abgelegt. Ein `RailNode` enthält neben einer eindeutigen ID und seinen Koordinaten zusätzlich seine Nachfolgeknoten, sowie die Abbiegepunkte des Pfades dorthin. Die RailNodes sind in einer `HashMap` unter ihrer ID abgelegt.

Bereits bei der Erzeugung der `HashMap` werden die minimalen und maximalen Koordinaten der Knoten ermittelt - diese Werte werden später für die Voreinstellung des Zoomfaktors bei der initialen Darstellung des Gleisplans benötigt.

8.3 Erstellung des SVG-Grundgerüstes

Das SVG-Dokument wird im Folgenden nach und nach als DOM-Tree aufgebaut. Dazu wird die in Java enthaltene DOMImplementation des Xerces2-Parsers der Apache Foundation verwendet.

```
DOMImplementation impl =
              DOMImplementationImpl.getDOMImplementation();
```

Um ein SVG-Dokument zu erstellen, muss zunächst die DTD definiert werden.

```
DocumentType docType = impl.createDocumentType("svg",
        "-//W3C//DTD SVG 1.1//EN",
        "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd");
```

Mit dem Namen des Wurzelelements und dem `docType` wird nun das Dokument erstellt, das außer dem leeren Wurzelelement noch nichts enthält.

```
Document doc = impl.createDocument(null, "svg", docType);
```

Da das SVG-Dokument in einen Navigationsbereich am oberen Rand und in einen Anzeigebereich auf der übrigen Fläche eingeteilt ist, werden unterhalb des Wurzelelements zwei `g`-Elemente eingefügt und mit den IDs `"navigation"` und `"display"` versehen. Auf diese Weise können die Elemente, die später im Anzeigebereich die Visualisierung darstellen werden, von den für die Navigation benötigten Elementen getrennt werden. Über die `g`-Elemente werden die Elemente zusätzlich gruppiert. Dieses hat später beim Zooming einen großen Vorteil, da sich alle zu zoomenden Elemente über dieses einzelne Containerelement ansprechen lassen.

Das Einfügen von Elementen in das SVG-Dokument erfolgt immer nach gleichem Muster über die DOM-Schnittstelle. Im folgenden Beispiel wird gezeigt, wie das `g`-Element mit der ID `"navigation"` erstellt und ins SVG-DOM eingefügt wird.

Elemente zu erzeugen ist Aufgabe des `Document`-Objektes.

```
Element g = doc.createElement("g");
```

Dem Element können eine beliebige Anzahl von Attributwerten hinzugefügt werden.

```
g.setAttribute("id","navigation");
```

Anschließend wird es an ein anderes Element, in diesem Fall an das Wurzelelement, als letztes Kindelement angehängt.

```
svg.appendChild(g);
```

Unterhalb der beiden Gruppenelemente `"navigation"` und `"display"` werden im Weiteren ebenfalls `g`-Elemente mit IDs eingefügt, um die verschiedenen einzufügenden Elementgruppen voneinander zu trennen. Dies dient zum Einen der Übersichtlichkeit, zum Anderen lässt sich die Reihenfolge der Elementegruppen festlegen und auch leicht ändern. Die Reihenfolge der Elemente in einem SVG-Dokument definiert auch die Reihenfolge der Elemente bei der Anzeige, im Dokument zuletzt stehende Elemente liegen über Elementen, die vorher definiert wurden und können diese verdecken.

8.4 Einfügen des ECMAScript und der CSS-Definitionen

Das ECMAScript wird in einem `script`-Element eingeschlossen. Es befindet sich unterhalb des `defs`-Elements. Da es sich aus Sicht des SVG-Documents um Text handelt, der nicht von einem XML-Parser geparst werden muss, wird der Skript-Text von einer `CDATA-Sektion` eingeschlossen.

```
defs = doc.createElement("defs");
svg.appendChild(defs);

Element script = doc.createElement("script");
script.setAttribute("type", "text/ecmascript");
defs.appendChild(script);

Element cdata = doc.createCDATASection("\n");
script.appendChild(cdata);
```

Das ECMA-Script bleibt für jedes SVG-Dokument gleich und liegt bereits in Form einer Textdatei vor. Bevor diese aber in den `CDATA`-Bereich kopiert werden kann, werden neben dem Faktor der Animationsgeschwindigkeit noch die minimalen und maximalen Koordinaten des Gleisplans als Variablen hinzugefügt, um ein Anfangszooming definieren zu können, das den ganzen Gleisplan anzeigt. Anschließend wird die Datei zeilenweise gelesen und eingefügt.

```
cdata.appendData("var animationSpeed = "
                 + Model.ANIMATION_SPEED + "\n\n");

int margin = 120;

cdata.appendData("var xMin = " + (xMin - margin) + "\n");
cdata.appendData("var xMax = " + (xMax + margin) + "\n");
cdata.appendData("var yMin = " + (yMin - margin) + "\n");
cdata.appendData("var yMax = " + (yMax + margin) + "\n");

buffer = new BufferedReader(new FileReader(JAVASCRIPT_URI));
while ((line = buffer.readLine()) != null)
  cdata.appendData(line + "\n");
```

Das ECMA-Script enthält eine Initialisierungfunktion, die nach dem Laden des SVG-Dokuments ausgeführt wird. Damit sie aufgerufen wird, muss sie im Event-Handler `onload` des Wurzelelements eingetragen werden.

```
<svg onload="init(evt);">
```

Die Textdatei mit den CSS-Angaben für die Formatierung der Elemente wird analog in einem `style`-Element in das `defs`-Element eingehängt.

8.5 Generierung der Navigationsleiste

Die Navigationsleiste besteht aus mehreren Buttons und einem Bereich, in dem die Fahrplaninformationen eines Zuges angezeigt werden können. Die Buttons bestehen aus mehreren grafischen Grundelementen, die über den Event-Handler `onclick` mit den ECMA-Script-Funktionen verknüpft sind. Die Tabellen mit den Fahrplaninformationen der einzelnen Züge werden bereits bei der Erzeugung des SVG-Dokuments erstellt, allerdings befinden sie sich außerhalb des Anzeigebereichs der SVG-Grafik. Eine Tabelle, deren Elemente durch ein `g`-Element gruppiert werden, wird bei Bedarf eingeblendet, indem die Koordinaten des Gruppen-Elements tanslatiert werden. Da die Tabellen mit Fahrplaninformationen gefüllt werden, können sie erst während der Animationsberechnung erzeugt werden.

8.6 Generierung des Gleisplans aus der `nodeList`

Der Gleisplan besteht aus den Gleisen und den Signal- und Kontrollpunkten. Die Gleise setzen sich wiederum aus den Gleisstücken zwischen den Knoten zusammen. Da die HashMap `nodeList` neben den Knoten auch die Pfade zu den Nachfolgern enthält, können die Gleisstücke als `line`-Elemente aus den Pfadangaben generiert werden. Die Pfadangaben enthalten auch die Koordinaten der Abbiegepunkte an den Weichen, daher ergibt sich aus den Gleisstücken bereits das komplette Gleisbild.

```
<g id="blocks">
  <line x1="1342" y1="250" x2="1442" y2="250" class="black" />
  ...
  <line x1="1342" y1="250" x2="1442" y2="250" class="white" />
  ...
</g>
```

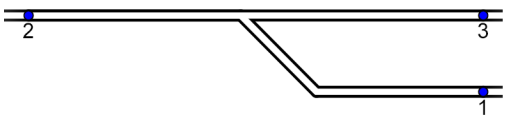

Abbildung 8.2: Gleisdarstellung durch zwei übereinanderliegende Linien

Ein Gleisstück besteht aus einer dickeren schwarzen Linie, vor der an gleicher Position eine dünnere weiße Linie liegt. So ergibt sich ein Gleisbild in Form einer

doppelten dünnen Linie, die ein Schiene andeutet (siehe Abb. 8.2). Der Pfad der Animation, der durch die selben Punkte definiert wird, verläuft daher genau in der Mitte der beiden dünnen Linien. Für die Bewegungsanimation eines Zuges entsteht dadurch der Eindruck, dass die Waggons auf diesen Schienen fahren würden.

Die Signal- und Kontrollpunkte werden als `circle`-Elemente eingefügt und durch ein `text`-Element mit ihrer ID versehen.

```
<g id="nodes">
  <circle cx="1342" cy="250" r="3" class="blueNode" />
  <text x="1342" cy="265" class="nodeIdText">1</text>
  ...
</g>
```

8.7 Generierung der Signale mit ihren Signalabfolgen

Signalinformationen sind in Objekten der Klasse `Signal` enthalten. Ein `Signal` enthält einen Zeitpunkt und eine Signalfarbe, die zum zugehörigen Zeitpunkt gesetzt wird. Die Signalinformationen eines Signalknotens, die beim Erzeugen des `ScheduleReader` eingelesen wurden, können von diesem mit der Methode `getSignalNodes(node)` in einer `ArrayList` zurückgeliefert werden. Für jedes Signal wird ein `circle`-Element in das SVG-Dokument eingefügt. Die Farbe der Signale ist zunächst hellgrau. Für jedes `Signal` wird in dem Kreiselement ein `set`-Element als Kindelement eingehängt, welches zum angegebenen Zeitpunkt die Füllfarbe des Kreises auf die Signalfarbe setzt.

```
<circle cx="1342" cy="220" r="8" class="signal">
  <set attributeName="fill" to="green"
       begin="8s" dur="indefinite" />
  <set attributeName="fill" to="red"
       begin="12s" dur="indefinite" />
  ...
</circle>
```

8.8 Generierung der Waggonanimationen

Der `scheduleReader` kapselt neben den Signalinformationen auch die Zuginformationen. Mit der Methode `getSpeedSpots(train)` lässt sich für den Zug mit der Nummer `train` der Fahrplan abfragen. Der Fahrplan eines Zuges besteht aus einer Liste, die alle passierten Knoten des Zuges enthält. In einem Knoten sind neben den Koordinaten auch der Durchfahrtszeitpunkt und die Geschwindigkeit

des Zuges bekannt. Die Daten eines Knotens werden daher in einem Objekt der Klasse `SpeedSpot` gekapselt. Ein SpeedSpot wird im Folgenden als *Geschwindigkeitspunkt* bezeichnet. Die Geschwindigkeitspunkte sind chronologisch sortiert und werden vom `scheduleReader` in einer `SpotList` zurückgeliefert, die eine Unterklasse von `ArrayList` ist. Sie implementiert zusätzlich Methoden um beispielsweise die Richtung des Zuges abfragen zu können.

Die Waggons eines Zuges werden in der Visualisierung durch Rechtecke dargestellt. Im `defs`-Element befindet sich ein Rechteck, das als Prototyp dient.

```
<rect height="10" width="22" x="-5" y="-11"
    id="waggon" class="waggon" />
```

Es ist wichtig, dass der Ursprung im Mittelpunkt des Rechtecks liegt. Dieser ist später der Bezugspunkt der Animation entlang eines Pfades, das heißt das Rechteck wird so bewegt, dass sich dieser Mittelpunkt und nicht etwa eine Ecke jederzeit genau auf dem Pfad befindet.

Um ein Rechteck aus dem Prototyp zu erzeugen, wird das `use`-Element verwendet.

```
<use xlink:href="#waggon" id="waggon1.1" x="0" y="0" />
```

Die Rechtecke werden im nichtsichtbaren Anzeigebereich positioniert.

Fasst man die Rechtecke eines Zuges als Gruppe zusammen und animiert anstatt der Rechtecke das `g`-Element, stellt sich ein Problem. Die Gruppierung der Rechtecke hätte zur Folge, dass die Rechtecke gradlinig hintereinander angeordnet würden und als statisches Objekt erscheinen, was an einem Abbiegepunkt augenscheinlich Schwierigkeit macht. Die Abbildung 8.3 zeigt eine Animation der Waggongruppe kurz vor und kurz nach der Durchfahrt der Lok durch einen Abbiegepunkt. Bezugspunkt jeder Animation ist der Ursprung des animierten Objekts, in diesem Fall ist das der Mittelpunkt des Rechtecks, das die Lok darstellt.

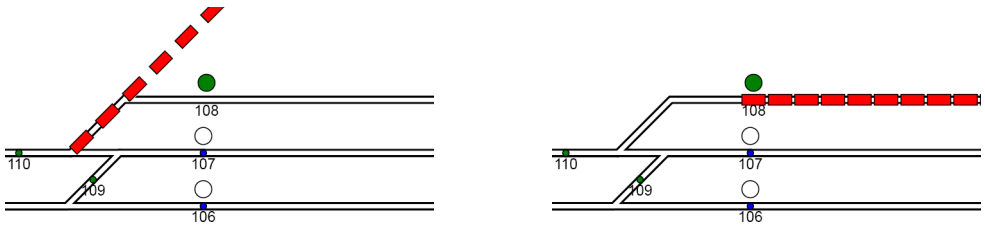

Abbildung 8.3: Zwei Zwischenbilder der Animation einer Gruppe von Waggons

Die Animationen der Waggons müssen daher einzeln und unabhängig voneinander generiert werden, was allerdings die Anzahl der `animation`-Elemente stark erhöht. Die Anzahl der Waggons eines Zuges hängt von der Zuglänge ab, die für jeden Zug aus dem `scheduleReader` ausgelesen wird.

```
scheduleReader.getTrainLength(train);
```

Die gesamte Animation eines Waggons spaltet sich an den Geschwindigkeitspunkten in einzelne Bewegungsanimationen auf. Ein Objekt der Klasse `WaggonAnimations`, eine Unterklasse von `ArrayList`, enthält eine Liste der Einzelanimationen für einen Waggon. Eine Einzelanimation wird durch die Klasse `MotionAnimation` repräsentiert und enthält die Koordinaten des Start- und Endpunkts und die Geschwindigkeiten und Durchfahrtszeiten in den Punkten. Sie enthält weiterhin eine Methode, die aus diesen Daten das `keySplines`-Attribut berechnen kann, das die Weg-Zeit-Funktion der Animation definiert.

Der Konstruktor der Klasse `WaggonAnimations` benötigt neben der Liste der Geschwindigkeitspunkte und der Position des Waggons im Zug noch eine Liste der Abbiegepunkte, die zwischen den Geschwindigkeitspunkten liegt. Diese wird aus dem Gleisplan generiert.

```
SpotList turnSpotList = railNodes.getTurnSpots(speedSpotList);
```

Um die Animation eines Waggons zu realisieren sind im Folgenden die `MotionAnimations` eines `WaggonAnimations`-Objektes zu erstellen. Ein `MotionAnimation`-Objekt kann direkt aus zwei Geschwindigkeitspunkten erzeugt werden, da alle benötigten Werte wie Zeitpunkt, Geschwindigkeit und Koordinaten in einem Geschwindigkeitspunkt enthalten sind. Allerdings ergeben sich an dieser Stelle noch einige Probleme.

Das erste Problem ist, dass die Geschwindigkeitspunkte nur für die Lok vorliegen. Die Waggons können zwar die Geschwindigkeitspunkte der Lok verwenden, sie liegen aber um den Abstand des Waggons zur Lok gegen die Fahrtrichtung verschoben. So haben die Waggons immer den gleichen Abstand untereinander.

In Abbildung 8.4 ist die Verschiebung der Geschwindigkeitsknoten für den ersten Waggon dargestellt. Seien Knoten (1) und (2) die Geschwindigkeitspunkte der Lok. Dieser Abschnitt enthält die Abbiegepunkte (a) und (b). Verschiebt man die Geschwindigkeitspunkte der Lok entgegen der Fahrtrichtung entlang dem Pfad um eine Waggonlänge (25m), ergeben sich für den ersten Waggon die Geschwindigkeitsknoten (1') und (2'). Der verschobene Abschnitt enthält nur noch den Abbiegepunkt (a). Für jeden Waggon ergibt sich daher auch eine individuelle Aufteilung der Abschnitte mit teilweise unterschiedlich vielen Abbiegepunkten.

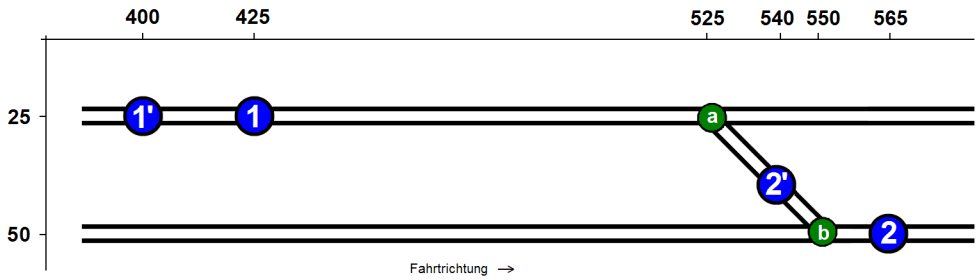

Abbildung 8.4: Verschiebung der Geschwindigleitsknoten für den ersten Waggon

Wenn einer der Geschwindigkeitspunkte ein Haltepunkt des Zuges ist, kann sich ein weiteres Problem ergeben. In einem Haltepunkt liegt eine Geschwindigkeit von 0 $\frac{km}{h}$ vor, d.h. die Tangente in diesem Punkt, auf der ein Kontrollpunkt liegt, ist waagerecht. Bei der Berechnung der Weg-Zeit-Funktion in Form einer Bézierkurve kann es in diesem Fall auf Grund der Beschaffenheit der vorliegenden Daten passieren, dass ein Kontrollpunkt außerhalb des Rechtecks liegt, welches durch Start- und Endpunkt der Bézierkurve definiert ist. Wie in Abbildung 8.5 im ersten Bild dargestellt, verlässt dann die Bézierkurve selbst auch das Rechteck. Sie kann in diesem Fall nicht zur Animation verwendet werden. Das SVG-Plugin von Adobe reagiert darauf, indem es die Animation des Waggons mit einer linearen Interpolation des zurückgelegten Weges berechnet. Das führt aber dazu, dass die Abstände der Waggons nicht mehr konstant bleiben. Die einzelnen Animationen müssen daher korrigiert werden. Der Ansatz zur Korrektur ist dabei, den Zeitraum, in dem die Kurve außerhalb des Rechtecks liegt, als zusätzliche Standzeit anzusehen, weil das animierte Objekt, wenn es beispielsweise oben aus dem Rechteck austritt, zum Zeitpunkt des Austritts bereits den Zielpunkt erreicht hat.

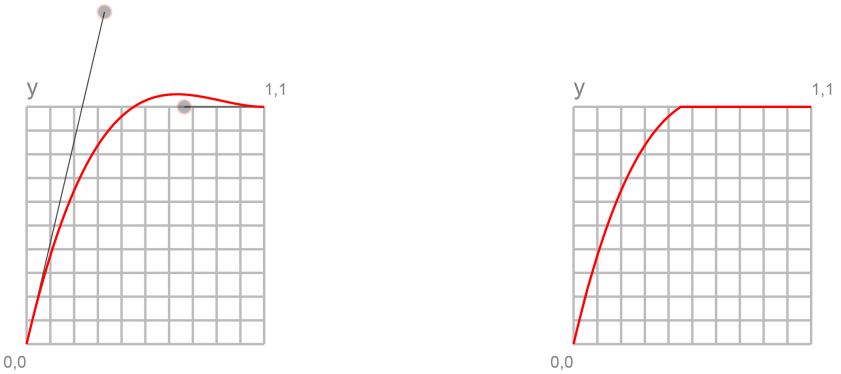

Abbildung 8.5: Eine ungültige Weg-Zeit-Funktion vor und nach Korrektur

Wie man in Abbildung 8.5 im rechten Bild erkennen kann, erzeugt die Korrektur einen Knick in der Weg-Zeit-Kurve beim Erreichen oder Verlassen des Haltepunktes, was gleichbedeutend ist mit einer Unstetigkeitsstelle in der Ge-

schwindigkeitsfunktion. Anschaulich bedeutet das für die Visualisierung, das ein Zug beispielsweise beim Erreichen eines Haltepunktes noch eine Geschwindigkeit besitzt, die der Steigung der Kurve im Knickpunkt entspricht. Im Haltepunkt wird dieser Zug dann abrupt gestoppt. Da der Geschwindigkeitsunterschied aber nicht sehr groß ist, fällt das dem Betrachter nicht auf. Er hat immernoch den Eindruck, eine Animation mit stetiger Geschwindigkeitskurve zu sehen.

Das dritte Problem ist die individuelle Form der Pfade auf den Abschnitten zwischen zwei Geschwindigkeitspunkten. Bei einem gradlinigen Pfad zwischen zwei Geschwindigkeitspunkten könnte man direkt das `MotionAnimation`-Objekt erzeugen, mit dem sich ein `animateMotion`-Element für den Abschnitt erstellen und ins SVG-Dokument einfügen ließe. Allerdings muss hier noch auf die Abbiegepunkte geachtet werden. Denn wie bereits in Kapitel 3.6.5 erwähnt, kann ein `animateMotion`-Element nur auf gradlinige Pfade angewendet werden. Um dieses zweite Problem zu lösen, muss daher an jedem Abbiegepunkt ein Geschwindigkeitspunkt eingefügt werden.

In einem Abbiegepunkt sind nur die Koordinaten bekannt, jedoch weder der Durchfahrtszeitpunkt noch die Geschwindigkeit eines Waggons. Diese beiden fehlenden Daten können erst aus der Weg-Zeit-Funktion berechnet werden, die ein `MotionAnimation`-Objekt aus seinen Eingabedaten erstellt und mit der der Waggon in der SVG-Grafik animiert wird. Daher muss für einen Abschnitt zwischen zwei Geschwindigkeitspunkten zunächst die `MotionAnimation` erzeugt werden, die im darauf folgenden Schritt an den Abbiegepunkten aufgespalten wird.

```
public WaggonAnimation(SpotList speedSpotList,
    SpotList turnSpotList, int waggon)
{
  // verschiebe die SpeedSpots an Hand der Waggonnr.
  SpotList movedSpeedSpotList =
      speedSpotList.moveSpeedSpotList(turnSpotList, waggon);

  // erzeuge fuer je zwei SpeedSpots die MotionAnimations
  calcWaggonAnimations(movedSpeedSpotList);

  // korrigiere ungueltige Animationskurven
  checkMotionAnimations();

  // MotionAnimations an den TurnSpots aufspalten
  splitAnimationsByTurnSpots(turnSpotList);
}
```

Quellcode 8.2: Konstruktor der Klasse `WaggonAnimations`

Wie am Konstruktor der Klasse `WaggonAnimations` in Quellcode 8.2 abzulesen ist, lässt sich die Erzeugung der Bewegungsanimationen eines Waggons in vier Schritte aufteilen:

1. Verschieben der Geschwindigkeitspunkte
2. Berechnung der einzelnen `MotionAnimation`s
3. Korrektur ungültiger `MotionAnimation`s
4. Aufspalten der Einzelanimationen an den Abbiegepunkten

8.8.1 Verschieben der Geschwindigkeitspunkte

Das Verschieben der Geschwindigkeitspunkte ist nicht trivial. Das liegt daran, dass ein Zug während der Durchfahrt durch den Gleisplan mehrfach das Gleis wechseln kann und daher auf die Abbiegepunkte geachtet werden muss. Mit Hilfe der Liste der Abbiegepunkte und der Position des Waggons im Zug werden die Koordinaten der verschobenen Geschwindigkeitspunkte berechnet. Die Methode zur Verschiebung der Geschwindigkeitspunkte ist in Quellcode 8.3 gelistet.

Die x-Koordinate des verschobenen Geschwindigkeitspunktes ergibt sich aus der Position des Waggons im Zug. Die Schwierigkeit bei der Bestimmung der neuen Koordinaten liegt daher in der Berechnung der y-Koordinate. Der neue Punkt liegt auf einer Strecke, die durch zwei Geschwindigkeits- und/oder Abbiegepunkte definiert ist.

Zunächst wird der letzte Geschwindigkeitspunkt vor der aktuellen Stelle ermittelt. Dann wird die Liste der Abbiegepunkte nach dem letzten Abbiegepunkt vor der aktuellen Stelle abgefragt. Dafür wird in beiden Fällen von der Klasse `SpotList` die Methode `getLastSpotBefore()` verwendet. Der nähergelegende dieser beiden Punkte ist der eine Streckenendpunkt. Mit der Methode `getFirstSpotAfter()` wird analog der nächstliegende Abbiege- bzw. Geschwindigkeitspunkt hinter der Stelle x ermittelt, der den anderen Streckenendpunkt darstellt. Mit Hilfe der Geradengleichung der Strecke kann dann über den x-Wert die y-Koordinate berechnet werden.

Mit diesen Koordinaten wird ein neuer Geschwindigkeitspunkt erzeugt, der in die Liste der verschobenen Geschwindigkeitspunkte eingefügt wird.

```
public SpotList moveSpeedSpotList(SpotList turnSpots,
                                  int waggon) {
  SpotList movedSpots = new SpotList(size());

  for (Iterator iter = iterator(); iter.hasNext();) {
    Spot spot = (Spot) iter.next();
    int y, x = spot.getX() - getDirection()
        * Model.WAGGON_OFFSET * waggon;

    Spot lastSpotBefore = getLastSpotBefore(x);

    if (lastSpotBefore == null)
      y = spot.getY();
    else {
      Spot lastTurnSpotBefore = turnSpots.getLastSpotBefore(x);

      if (lastTurnSpotBefore != null
          && lastTurnSpotBefore.getX() * getDirection()
             > lastSpotBefore.getX() * getDirection())
        lastSpotBefore = lastTurnSpotBefore;

      Spot firstSpotAfter = getFirstSpotAfter(x);
      Spot firstTurnSpotAfter =  turnSpots.getFirstSpotAfter(x);

      if (firstTurnSpotAfter != null
          && firstTurnSpotAfter.getX() * getDirection()
             < firstSpotAfter.getX() * getDirection())
        firstSpotAfter = firstTurnSpotAfter;

      int x1 = lastSpotBefore.getX();
      int y1 = lastSpotBefore.getY();
      int x2 = firstSpotAfter.getX();
      int y2 = firstSpotAfter.getY();

      y = (x1 == x2) ? y1 : (x-x1) * (y2-y1) / (x2-x1) + y1;
    }
    movedSpots.add(new SpeedSpot(spot.getNodeId(),
        spot.getTime(), spot.getVelocity(),
        new Coords(x, y), spot.getStopTime()));
  }
  return movedSpots;
  }
}
```

Quellcode 8.3: Methode zur Verschiebung der Geschwindigkeitspunkte

8.8.2 Berechnung der Einzelanimationen

Die Methode `calcMotionAnimations(movedSpeedSpotList)` (siehe Quellcode 8.4) durchläuft danach die Liste der Geschwindigkeitspunkte eines Waggons, die bereits des Abstands zur Lok entsprechend verschoben wurden. Aus jeweils zwei benachbarten `SpeedSpot`s werden die benötigten Werte ausgelesen und aus ihnen eine `MotionAnimation` erzeugt.

```
private void calcWaggonAnimation(SpotList speedSpotList)
{
  SpeedSpot startSpot, endSpot = null;

  Iterator speedSpotListIter = speedSpotList.iterator();

  if (speedSpotListIter.hasNext())
  {
    startSpot = (SpeedSpot) speedSpotListIter.next();

    ... // Animation fuer Einfahrt

    while (speedSpotListIter.hasNext())
    {
      endSpot = (SpeedSpot) speedSpotListIter.next();

      add(new MotionAnimation(startSpot.getTime() +
          startSpot.getStopTime(), endSpot.getTime(),
          startSpot.getVelocity() * speedSpotList.
          getDirection(), endSpot.getVelocity()
          * speedSpotList.getDirection(), startSpot.
          getCoords(), endSpot.getCoords()));

      // neuer Startpunkt ist alter Endpunkt
      startSpot = endSpot;
    }
    ... // MotionAnimation fuer Ausfahrt erstellen
  }
}
```

Quellcode 8.4: Methode zur Berechnung der Waggonanimationen

Als erste und letzte `MotionAnimation` wird für jeden Waggon jeweils eine zusätzliche Animation eingefügt. Diese Animationen werden benötigt, um einen Zug flüssig im Gleisbild ein- und ausfahren zu lassen. Andernfalls würden die Züge im ersten Knoten ihres Pfades wie aus dem Nichts erscheinen und im letzten Kno-

ten abrupt stehen bleiben. Die beiden Pfade der beiden zusätzlichen Animationen werden durch ein Rechteck überdeckt, das mit einem Transparenzgradienten gefüllt ist, um so den Eindruck zu erwecken, dass ein Zug vor seinem ersten Knoten aus völliger Transparenz eingeblendet und bei der Ausfahrt ebenso wieder ausgeblendet wird.

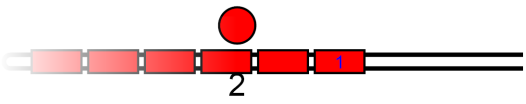

Abbildung 8.6: Transparenzeffekt bei Ein- und Ausfahrt

8.8.3 Korrektur ungültiger `MotionAnimations`

Der Zeitpunkt, zu dem ein Waggon einen bestimmten Punkt erreicht, wird mit einem Intervallschachtelungsverfahren berechnet, das durch die rekursive Methode `getSplitTime()` (siehe Quellcode 8.5) realisiert wird.

```
public double getSplitTime(double t1, double t2, double s1,
                           double s2, double s_split)
{
  double t_mitte = t1 + (t2 - t1) / 2;
  double s_mitte = bezier(t_mitte);

  if (Math.abs(t2 - t1) > 0.0001)
  {
      if ((s_mitte - s_split) * (s2 - s1) > 0)
      t = getSplitTime(t1, t_mitte, s1, s_mitte, s_split);
    else
      t = getSplitTime(t_mitte, t2, s_mitte, s2, s_split);
  }

  return t;
}
```

Quellcode 8.5: Methode zur Berechnung des Zeitpunktes der Aufspaltung

In einem Methodenaufruf wird zunächst der Wert des zurückgelegten Weges $s(t_{mitte})$ in der Mitte des Intervalls ermittelt. An Hand des Vorzeichens der Differenz $d = s(t_{mitte}) - s(t_{split})$ lässt sich ablesen, in welchem Halbintervall t_{split} liegt und mit welchem rekursiven Aufruf weitergesucht werden muss. Die Rekursion bricht ab, sobald das Intervall eine festgelegte Größe unterschreitet.

Mit der Methode `getSplitTime()` wird im Folgenden ermittelt, ob eine ungültige Weg-Zeit-Funktion vorliegt, also der Start- oder Endpunkt zusätzlich noch einmal innerhalb des Zeitintervalls der Animation durchlaufen wird. Liegt dieser Fall vor, liefert die Methode den zugehörigen Zeitpunkt t_{split} zurück. In der Methode `checkMotionAnimations()` wird dieser Zeitpunkt verarbeitet, um eine gültige `MotionAnimation` zu errechen.

```
public void updateSpeedSpots()
{
  double tSplit =
      this.getSplitTime(t1, t2, coords1.getX(),
                        coords2.getX(), coords1.getX());

  if (Math.abs(t1 - t1New) > EPS)
  {
    t1 = tSplit;
    v1 = bezierDerivated(tSplit);
  }
  else
  {
    tSplit = this.getSplitTime(t1, t2, coords1.getX(),
                               coords2.getX(), coords2.getX());

    if (Math.abs(t2 - t2New) > EPS)
    {
      t2 = tSplit;
      v2 = bezierDerivated(tSplit);
    }
  }
}
```

Quellcode 8.6: Methode zur Korrektur einer ungültigen `MotionAnimation`

Mit der Zeit t_{split} lässt sich die Geschwindigkeit $v(t_{split})$ der eigentlich ungültigen Animation berechnen. Die Berechnung der Abbleitung der Bézierkurve und der Bézierkurve selbst, die in den Formeln 9 und 10 weiter unten beschrieben wird, liegen als Methoden in der Klasse `MotionAnimation` vor. Sie berechnen den zurückgelegten Weg bzw. die Geschwindigkeit direkt aus dem Anfangs- und Endgeschwindigkeitspunkt der `MotionAnimation`.

Je nachdem, ob die Kurve das Rechteck oben oder unten verlässt, ersetzen die berechnete Zeit und die Geschwindigkeit die Originalwerte zum Start- oder Endzeitpunkt. Auf diese Weise wird die Animation um die Zeit verkürzt, die die Kurve außerhalb des Rechtecks verläuft. Als neuer Kurvenverlauf der Weg-Zeit-Funktion ergibt sich das Kurvenstück, das innerhalb des Rechtecks verläuft. Für

```
public double bezier(double t)
{
  return (coords2.getX() * (t - t1) * (t - t1) * (t - t1)
      + coords1.getX() * (t2 - t) * (t2 - t) * (t2 - t)
      - (t - t1) * (t - t2) * (t - t2) * (-3 * coords1.getX()
      + (t1 - t2) * v1) - (t - t1) * (t - t1) * (t - t2)
      * (3 * coords2.getX() + (t1 - t2) * v2)) / ((t2 - t1)
      * (t2 - t1) * (t2 - t1));
}
```

Quellcode 8.7: Methode zur Berechnung von $s(t)$

```
private double bezierDerivated(double t)
{
  return (-6 * coords1.getX() * (t - t1) * (t - t2) + 6
      * coords2.getX() * (t - t1) * (t - t2) + (t1 - t2)
      * ((t - t2) * (3 * t - 2 * t1 - t2) * v1 + (t - t1)
      * (3 * t - t1 - 2 * t2) * v2)) / ((t1 - t2) * (t1 - t2)
      * (t1 - t2));
}
```

Quellcode 8.8: Methode zur Berechnung von $v(t) = s'(t)$

die übrige Zeit existiert keine Animation mehr. Sie verlängert daher die Standzeit des Waggons im Haltepunkt.

8.8.4 Aufspalten der Einzelanimationen an den Abbiegepunkten

Die Einzelanimationen eines Waggons werden in der Methode `splitAnimationsByTurnSpots()`, die in Quellcode 8.9 angegeben ist, der Reihe nach aufgespalten.

Zunächst werden die Abbiegepunkte `intervalSpots` abgefragt, die auf dem Animationsabschnitt liegen. Diese `SpotList` enthält die Abbiegepunkte des Abschnitts in umgekehrter Reihenfolge des Besuchs. An jedem Abbiegepunkt wird die `MotionAnimation` in zwei verschiedene `MotionAnimation`s aufgeteilt. Da der letztbesuchte Abbiegepunkt zuerst betrachtet wird, enthält der zweite Teil der ursprünglichen `MotionAnimation` keine Abbiegepunkte mehr. Er wird daher an der passenden Stelle in die `WaggonAnimations` eingefügt. Solange noch Abbiegepunkte vorhanden sind, wird der erste Teil der ursprünglichen `MotionAnimation` weiter am letzten Abbiegepunkt aufgeteilt.

Das Aufspalten einer `MotionAnimation` an einem Abbiegepunkt wird in der Methode `split()` (siehe Quellcode 8.10) realisiert.

Als Parameter bekommt die Methode die Koordinaten des Abbiegepunkts übergeben, die die zurückgelegte Strecke $s(t_{split})$ eines Waggons enthalten. Der Zeitpunkt

```
private void splitAnimationsByTurnSpots(SpotList turnSpotList)
{
  MotionAnimation animation;

  for (int i = 0; i < size(); i++)
  {
    animation = (MotionAnimation) get(i);

    // TurnSpots eines Intervalls holen
    SpotList intervalSpots = turnSpotList.getSpotsOnInterval(
        animation.getS1(), animation.getS2());

    if (!intervalSpots.isEmpty())
    {
      int turns = 0;

      for (Iterator intervalSpotsIter = intervalSpots.
               iterator(); intervalSpotsIter.hasNext();)
      {
        Coords turn = ((TurnSpot) intervalSpotsIter.next()).
                                                  getCoords();
        // Bezierkurve in zwei teile aufteilen
        MotionAnimation animation2 = animation.split(turn);

        add(i + 1, animation2);
        turns++;
      }
      i += turns;
    }
  }
  // y-Koordinate der Ausfahrtanimation korrigieren
  animation = (MotionAnimation) get(size() - 1);
  animation.setY2(animation.getCoords1().getY());
}
```

Quellcode 8.9: Methode zum Aufspalten der Animationen eines Waggons

t_{split}, an dem der Waggon den Abbiegepunkt $s(t_{split})$ erreicht, wird wieder über die Methode `getSplitTime()` (siehe Quellcode 8.5) berechnet. Die Geschwindigkeit $v(t_{split})$ erhält man ebenfalls wieder durch Einsetzen in die Ableitung der Weg-Zeit-Funktion `bezierDerivated()`.

Nachdem der Durchfahrtszeitpunkt und die Geschwindigkeit im Abbiegepunkt berechnet wurde, werden diese Werte mit der zurückgelegten Strecke als neue

```
public MotionAnimation split(Coords turn)
{
  // alte Werte merken
  double t2End = t2, v2End = v2;
  Coords coords2End = coords2;

  // Split-Zeitpunkt berechnen
  double t2Turn =
      getSplitTime(t1, t2, coords1.getX(), turn.getX());

  // Geschwindigkeit zum Split-Zeitpunkt berechnen
  v2 = bezierDerivated(t2Turn);

  // t2 und coord2 der ersten MotionAnimation uebernehmen
  t2 = t2Turn;
  coords2 = turn;

  // zweite MotionAnimation erzeugen
  return new MotionAnimation(t2, t2End, v2,
                             v2End, coords2, coords2End);
}
```

Quellcode 8.10: Methode zum Aufspalten einer `MotionAnimation`

Endwerte in der aktuellen `MotionAnimation` gesetzt. Für den Abschnitt hinter dem Abbiegepunkt wird eine neue `MotionAnimation` zurückgeliefert, die die neuberechneten Werte als Startwerte und die Endwerte der ursprünglichen `MotionAnimation` als neue Endwerte erhält.

In einem Objekt der Klasse `WaggonAnimations` liegen nun alle Einzelanimationen eines Waggons vor. Für jede `MotionAnimation` wird ein `animateMotion`-Element ins SVG-Dokument eingefügt.

```
<animateMotion xlink:href="#waggon1.1"
  begin="2.233s" dur="1.431s"
  rotate="auto" fill="freeze"
  calcMode="spline"
  keySplines="0.333 0.452 0.666 0.724"
  path="1342,250 L 1442,250" />
```

Die Werte der Attribute `begin` und `dur` können direkt aus der `MotionAnimation` ermittelt werden. Ebenso direkt lassen sich die Koordinaten des Anfangs- und Endpunktes der `MotionAnimation` im `path`-Attribut auslesen. Die folgenden beiden Attribute `rotate` und `fill` sind für alle `animateMotion`-Elemente gleich.

`rotate="auto"` richtet das zu animierende Objekt, also das Rechteck, immer entlang des Pfades aus (siehe Abb. 8.5). `fill="freeze"` bewirkt, dass das Rechteck nach dem Ende der Animation am Endpunkt verbleibt und nicht aus der Anzeige entfernt wird. Andernfalls würden Züge verschwinden, die an einem Haltepunkt wie z.B. einem Bahnsteig oder einem roten Signal halten, sobald sie dort gestoppt hätten und sie würden während ihrer Standzeit nicht angezeigt werden. Mit dem Attribut `calcMode="spline"` wird angegeben, dass die Interpolation über eine Bézierkurve erfolgt, deren Kontrollpunkte im Attribut `keySplines` enthalten sind.

Die beiden Ankerpunkte liegen in $(t_1, s(t_1))$ und $(t_2, s(t_2))$. Auf Grund der allgemeinen Eigenschaften von Bézierkurven 3. Grades liegen die beiden Kontrollpunkte jeweils auf einer Tangente, die in den Knotenpunkten an der Bézierkurve anliegt. Die Geschwindigkeit in den Ankerpunkten $v(t_1)$ und $v(t_2)$ gibt die Steigung der Tangenten an. Aus der Steigung und einem Ankerpunkt lässt sich jeweils die Tangentengleichung in t_1 und t_2 aufstellen:

$$T_1(t) = v(t_1) \cdot t + s(t_1) - v(t_1) \cdot t_1 \tag{3}$$

$$T_2(t) = v(t_2) \cdot t + s(t_2) - v(t_2) \cdot t_2 \tag{4}$$

Es ist nicht eindeutig festgelegt, wo sich die Kontrollpunkte auf den Tangenten befinden. Das bedeutet auf das Modell übertragen, dass die Geschwindigkeit eines Zuges nur in den Geschwindigkeitspunkten, nicht aber dazwischen eindeutig definiert ist. Tatsächlich hat ein Lokführer eine gewisse Freiheit, wie er seine Lok beispielsweise vor einem roten Signal zum Stehen bringt. In der Applikation wurden die Kontrollpunkte nach $\frac{1}{3}$ und $\frac{2}{3}$ des Intervalls gesetzt, sie berechnen sich zu

$$t_{k1} = \frac{2}{3}t_1 + \frac{1}{3}t_2 \tag{5}$$

$$t_{k2} = \frac{1}{3}t_1 + \frac{2}{3}t_2 \tag{6}$$

Daraus ergeben sich für die beiden Kontrollpunkte

$$s(t_{k1}) = tan_1(t_{k1}) = s(t_1) + \frac{1}{3}(t_2 - t_1)v(t_1) \tag{7}$$

$$s(t_{k2}) = tan_2(t_{k2}) = s(t_2) + \frac{1}{3}(t_1 - t_2)v(t_2) \tag{8}$$

Aus dem Anfangs- und Endpunkt, sowie aus den beiden berechneten Kontrollpunkten ergibt sich für den zurückgelegten Weg eines Waggons die Bézierkurve

$$
\begin{aligned}
s(t) \quad = \quad & \frac{s(t_2)\cdot(t-t_1)^3 + s(t_1)\cdot(t_2-t)^3}{(t_2-t_1)^3} - \\
& \frac{(t-t_1)(t-t_2)^2(-3s(t_1)+(t_1-t_2)\cdot v(t_1))}{(t_2-t_1)^3} - \\
& \frac{(t-t_1)^2(t-t_2)(3s(t_2)+(t_1-t_2)\cdot v(t_2))}{(t_2-t_1)^3}
\end{aligned} \tag{9}
$$

und für die Geschwindigkeit die Ableitung der Bézierkurve nach t

$$
\begin{aligned}
v(t) = s'(t) \quad = \quad & \frac{6(s(t_2)-s(t_1))(t-t_1)(t-t_2)}{(t_1-t_2)^3} - \\
& \frac{(t-t_2)(3t-2t_1-t_2)\cdot v(t_1)}{(t_1-t_2)^2} - \\
& \frac{(t-t_1)(3t-t_1-2t_2)\cdot v(t_2)}{(t_1-t_2)^2}
\end{aligned} \tag{10}
$$

Der Startpunkt und der Endpunkt der Bézierkurve definieren ein Rechteck, in dem die Bézierkurve verläuft. Um die Bézierkurve als Weg-Zeit-Funktion im `animateMotion`-Element angeben zu können, muss dieses Rechteck in das Einheitsquadrat transformiert werden. Dazu wird es erst wie in Abbildung 8.7 gezeigt um $S = (-t_1, -s(t_1))$ translatiert und danach mit $T = (\frac{1}{(t_2-t_1)}, \frac{1}{s(t_2)-s(t_1)})$ skaliert.

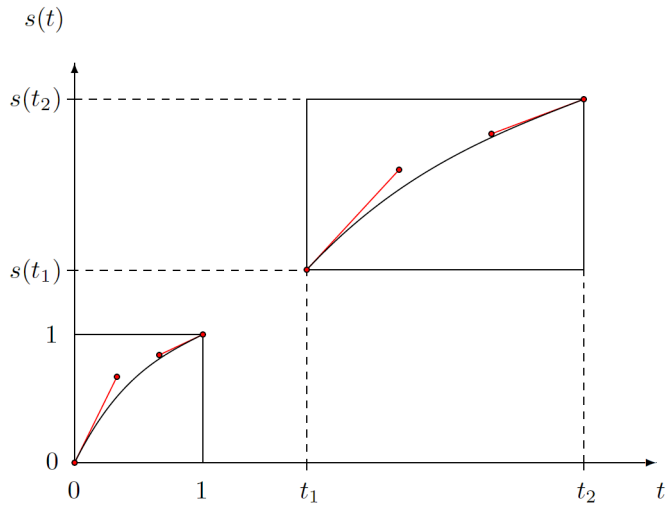

Abbildung 8.7: Die Bézierkurve vor und nach Transformation ins Einheitsquadrat

Die transformierten Zeitpunkte der Kontrollpunkte ergeben sich wie gewünscht zu $\bar{t}_{k1} = \frac{1}{3}$ und $\bar{t}_{k2} = \frac{2}{3}$. Für den zurückgelegten Weg in $\bar{s}(\bar{t}_{k1})$ und $\bar{s}(\bar{t}_{k2})$ ergibt sich nach Transformation:

$$\bar{s}(\bar{t}_{k1}) = \frac{(t_2 - t_1) \cdot v(t_1)}{3(s(t_2) - s(t_1))} \quad (11)$$

$$\bar{s}(\bar{t}_{k2}) = 1 - \frac{(t_2 - t_1) \cdot v(t_2)}{3(s(t_2) - s(t_1))} \quad (12)$$

Die Berechnung von $\bar{s}(\bar{t}_{k1})$ und $\bar{s}(\bar{t}_{k2})$ ist in den Methoden `getK1()` und `getK2()` implementiert, die in Quellcode 8.11 gelistet sind. Mit ihnen wird der Attributwert des `keySplines` erzeugt und eingesetzt.

```
public double getK1()
{
  return = (v1 * (t2 - t1))
      / (3 * (coords2.getX() - coords1.getX()));
}

public double getK2()
{
  return = 1 - (v2 * (t2 - t1))
      / (3 * (coords2.getX() - coords1.getX()));
}
```

Quellcode 8.11: Methoden zur Berechnung der Kontrollpunktkoordinaten

8.9 Anzeige des SVG-Dokuments

Nach dem Einfügen der `animationMotion`-Elemente aller Züge ist der DOM-Tree der SVG-Grafik vollständig. Um an das Browser-Widget übergeben zu werden, muss der DOM-Tree serialisiert werden. Dies geschieht mit Hilfe eines `XMLSerializer`s, der das SVG-Dokument über einen `FileWriter` in eine temporäre Datei schreibt.

```
public void write(String uri) throws IOException
{
  FileWriter writer = new FileWriter(uri);

  XMLSerializer out = new XMLSerializer(writer,
            new OutputFormat(doc, "utf-8", true));
  out.serialize(doc);
}
```

Quellcode 8.12: Methode zum Abspeichern des DOM

Das Model wird dem Browser-Widget in der `update()`-Methode übergeben. Es liest daraus die temporäre URI aus, um die neu erstellte SVG-Grafik anzeigen zu können.

```
public void update(Observable o, Object arg)
{
  browser.setUrl(
      new File(((Model)o).getURI()).getAbsolutePath());
  browser.refresh();
}
```

Quellcode 8.13: `update()`-Methode des Browser-Widgets

8.10 Die SVG-Grafik

In Abbildung 8.8 ist die Applikation abgebildet, nachdem die SVG-Grafik für den Bahnhof Amsterdam-Schiphol erstellt wurde. Sie teilt sich in zwei Bereiche. Am oberen Bildrand befindet sich eine Leiste, die durch einen grauen Hintergrund abgesetzt ist. Sie ermöglicht über Buttons die Navigation im Gleisplan sowie die Navigation auf der Zeitachse und kann Detailinformationen über einen Zug anzeigen. Den Rest der Grafik nimmt der Anzeigebereich des Gleisplans ein, der zu Beginn den Gleisplan in seiner gesamten Ausdehnung darstellt.

Abbildung 8.8: Screenshot (1) der Applikation

Das Zooming und die Animationssteuerung wird wie in Kapitel 3.6.6 beschrieben über das eingebundene ECMAScript realisisert. Nach dem vollständigen Laden der SVG-Grafik im Viewer wird zu Beginn der Anzeige über den Eventhandler `onload` die Initialisierungsfunktion `init()` aufgerufen.

```
function init(evt)
{
  doc = evt.target.ownerDocument;
  doc.documentElement.pauseAnimations();
  setZooming(xMin, xMax, yMin, yMax);
  getContextMenu().removeChild(getContextMenu().firstChild);
  window.setTimeout('init2()', 20);
}

function init2()
{
  doc.documentElement.setCurrentTime(minTime);
  setTime();
}
```

Zunächst wird das `ownerDocument` abgefragt und in der Variablen `doc` gespeichert. Über das `ownerDocument` erfolgen die Aufrufe an das DOM. Die Zeit wird zu Beginn angehalten. Mit minimalen und maximalen Koordinaten des Gleisplans wird der anfängliche Zoomfaktor berechnet, um den gesamten Gleisplan anzuzeigen. Anschließend wird das Kontextmenü, das im Allgemeinen über die rechte Maustaste geöffnet wird, unterdrückt. Der letzte Befehl in der `init()`-Methode ruft nach einer Verzögerung von 20 Millisekunden die Methode `init2()` auf, die als letzte Initialisierungsschritte den Startzeitpunkt setzt, den der Benutzer bei der Erzeugung angegeben hat, und die Methode `setTime()` startet, die in regelmäßigen Intervallen die Zeitanzeige aktualisiert.

Das Aufteilen der Initialisierungen in zwei Methoden wurde erforderlich, da sich der Adobe SVG-Viewer fehlerhaft verhält. Nachdem ein SVG-Dokument in den Viewer geladen wurde, wird das `onclick`-Event ausgelöst, welches die erste Initialisierungsmethode `init(evt)` aufruft. Diese Methode ist aber schon beendet, bevor der SVG-Viewer die gesamte Grafik anzeigt. Wenn die Anzeige fertig ist, wird im Viewer fälschlicherweise noch einmal die Zeitachse auf 0 zurückgesetzt. Damit würde aber der Aufruf `setCurrentTime(minTime)` überschrieben, wenn er in `init(evt)`-Methode stände. Um das Überschreiben zu verhindern, reicht eine Wartezeit von 20 Millisekunden, um auf die vollständige Anzeige der Grafik zu warten, bevor die Startzeit gesetzt wird.

8.10.1 Zooming

Abbildung 8.9: Buttonleiste für die Zoomfunktionen

Für das Zooming stehen dem Benutzer in der Buttonleiste (siehe Abbildung 8.9) folgende Funktionen zur Verfügung:

1. Der Button (1) ermöglicht, den Gleisplan bei gedrückter Maustaste entsprechend der Mausbewegung zu verschieben. Der Zoomfaktor wird dabei nicht verändert.
2. Der Button (2) liefert die Funktion, mit der Maus im Gleisplan ein Rechteck aufzuziehen, in welches hineingezoomt werden soll (siehe Abbildung 8.10).
3. Über die beiden Buttons (3) und (4) kann der Benutzer ein- und auszoomen. Der Zoomfaktor beträgt dabei 2 bzw. 0,5.
4. Über den Button (5) kann der letzte Schritt des Zooming rückgängig gemacht werden. Dies ist besonders sinnvoll, wenn ein sehr kleines Zoomingrechteck gewählt wurde.

Alle SVG-Elemente, die zur Darstellung des Gleisplans benötigt werden, sind in einem `g`-Element mit der ID `"display"` gruppiert. Dazu zählen nicht nur die Elemente zur Darstellung des Gleisplans und der Waggons, sondern auch die `animation`-Elemente der Waggons. Sie enthalten die Pfadangaben der Züge.

Um die Gruppe der Elemente entsprechend des gewünschten Zoomings und Ausschnitts darzustellen, muss sie, wie in Kapitel 3.6.3 beschrieben, über das `transform`-Element transformiert werden. Die Transformation setzt sich zusammen aus einer Skalierung und einer Translation. Dadurch ist es möglich, die Elemente zunächst mit ihren ursprünglichen Koordinaten im Gleisbild zu positionieren.

Das Zooming übernimmt eine Funktion. Sie bekommt als Parameter ein Rechteck übergeben, das die minimalen und maximalen absoluten Koordinaten des neu anzuzeigenden Ausschnitts enthält. Dieses Rechteck wird für jede Zoomingart in einer separaten Funktion berechnet, die jeweils durch das passende Maus-Event ausgelöst wird. Für fast alle Zoomingarten ist es wichtig, dass die aktuellen Minimal- und Maximalkoordinaten des angezeigten Gleisplans in Variablen vorgehalten werden.

Für das Zooming mit festen Zoomfaktoren über die Buttons (3) und (4) lassen sich die neuen Koordinaten einfach berechnen. Über die Klick-Events beim

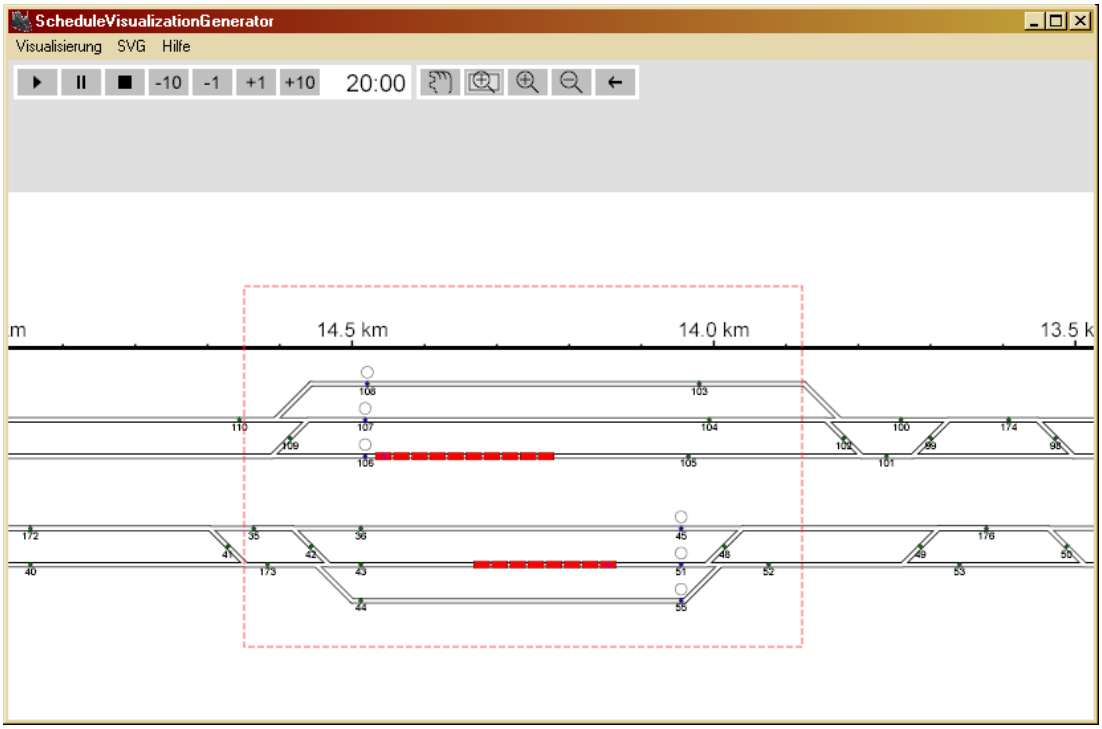

Abbildung 8.10: Screenshot (2) mit Zoomingrechteck

Erzeugen des Zoomingrechtecks (Button (2)) können die Koordinaten des Mauszeigers beim Klick ermitteln werden. Diese können ebenfalls über die aktuellen Minimal- und Maximalkoordinaten in das Koordinatensystem des Gleisplans überführt werden. Um einen Zoomingschritt über Button (5) rückgängig zu machen, ist es erforderlich, neben den aktuellen Minimal- und Maximalkoordinaten auch die alten Koordinaten zu speichern.

Der Aufruf der Zooming-Funktion führt dazu, dass zunächst die aktuellen Koordinaten des angezeigten Rechtecks gespeichert werden.

```
oldXMin = currentXMin;
oldXMax = currentXMax;
oldYMin = currentYMin;
oldYMax = currentYMax;
```

Über die Fenstergröße werden die Skalierungsfaktoren in Höhe und Breite berechnet, mit denen der Gleisplan aus den Ursprungskoordinatensystem in die gewünschte Anzeigegröße skaliert werden kann. Als Skalierungsfaktoren können sich für beide Dimensionen unterschiedliche Werte ergeben.

```
var winWidth = window.innerWidth;
var winHeight = window.innerHeight-naviHeight;
var scaleX = winWidth/(xMax-xMin);
var scaleY = winHeight/(yMax-yMin);
```

Um in beide Dimensionen einheitlich zu skalieren und den ganzen Ausschnitt anzeigen zu können, wird das Minimum der beiden Skalierungsfaktoren verwendet. Dabei werden die Koordinaten in der jeweils anderen Dimension so angepasst, dass sich das Seitenverhältnis des Anzeigerechtecks nicht verändert.

```
if (scaleX < scaleY)
{
  var scale = scaleX;
  var diff = Math.round(((xMax-xMin)*winHeight/winWidth
                         - (yMax-yMin))/2);
  currentXMin = xMin;
  currentXMax = xMax;
  currentYMin = yMin - diff;
  currentYMax = yMax + diff;
}
else
{
  var scale = scaleY;
  var diff = Math.round(((yMax-yMin)*winWidth/winHeight
                         - (xMax-xMin))/2);
  currentYMin = yMin;
  currentYMax = yMax;
  currentXMin = xMin - diff;
  currentXMax = xMax + diff;
}
```

Neben der Skalierung ergibt sich als Translation anschließend eine Verschiebung des Ausschnitts in die linke obere Ecke des Anzeigebereichs der SVG-Grafik. Die Transformationen werden als Attributwert in die `"display"`-Gruppe eingefügt:

```
var translateStr1 = 'translate(0, '+naviHeight+') ';
var scaleStr      = 'scale(-'+scale+', '+scale+') ';
var translateStr2 = 'translate('+(-currentXMax)+',
                                '+(-currentYMin)+')';

doc.documentElement.getElementById('display').setAttribute(
    'transform', translateStr1 + scaleStr + translateStr2);
```

8.10.2 Zeitsteuerung

Animationen in SVG erfolgen zeitbasiert (siehe Kapitel 3.6.5). Über die Funktion `getCurrentTime()` kann jederzeit die in der SVG-Grafik vergangene Zeit abgefragt werden. Diese wird in der folgenden Funktion verwendet:

```
function setTime()
{
  var time = document.documentElement
                .getCurrentTime() / animationSpeed;
  var seconds = Math.round((time-Math.floor(time))*60);
  if (seconds < 10)
    seconds = '0' + seconds;
  var timeStr = Math.floor(time)+':'+seconds;
  document.documentElement.getElementById('time')
      .getFirstChild().setNodeValue(timeStr);
  window.setTimeout('setTime()', 1000);
}
```

Abbildung 8.11: Buttonleiste der Zeitsteuerung

Mit `window.setTimeout('setTime', 1000)` wird diese Funktion alle 1000 Millisekunden erneut aufgerufen. Sie setzt jeweils die vergangene Zeit, indem sie den Inhalt eines `text`-Elements in der Navigationsleiste (8) überschreibt (siehe Abbildung 8.11). Bei der angezeigten Zeit handelt es sich um die verstrichene Zeit des Fahrplans in Minuten, die sich aus dem Animationsfaktor berechnet, der beim Erzeugen der SVG-Grafik angegeben werden kann.

Der zeitliche Ablauf innerhalb einer SVG-Grafik, also die Summe der Animationen, lässt sich über die Methoden `pauseAnimations()` und `unpauseAnimations()` anhalten und fortsetzen. Diese Methoden werden durch die Start- und Pause-Buttons (1) und (2) aufgerufen.

In SVG ist es auch möglich, sich auf der Zeitachse zu bewegen. Dadurch hat der Benutzer die Möglichkeit, an bestimmte Stellen der Animation zu springen, oder kurze Abschnitte mehrfach zu wiederholen. Über die Funktion `setCurrentTime(time)` lässt sich auf der Zeitachse zum Zeitpunkt `time` springen. Die Buttons der Zeitsteuerung (4)-(7) rufen diese Funktion auf.

8.10.3 Anzeige der Zuginformationen

Befindet sich der Benutzer nicht in einer der Zoomingfunktionen, hat er die Möglichkeit, sich die Fahrplaninformation eines Zuges anzeigen zu lassen (siehe Abbildung 8.12). Die Waggons der Züge reagieren dabei auf das Überfahren mit dem Mauszeiger. Der Eventhandler `onmouseover` löst eine Funktion aus, die eine Tabelle mit allen Zuginformationen in der Navigationsleiste anzeigt. Die Tabellen sind schon zu Beginn der Animation vorhanden, sind aber durch das Attribut `visibility="invisible"` unsichtbar. Um eine Tabelle anzuzeigen, wird das Attributwert per ECMAScript auf `visible` gesetzt. Die Tabellen enthalten jeweils die Geschwindigkeitspunkte eines Zuges, d.h. die Zeitpunkte und Geschwindigkeit aller passierten Knoten.

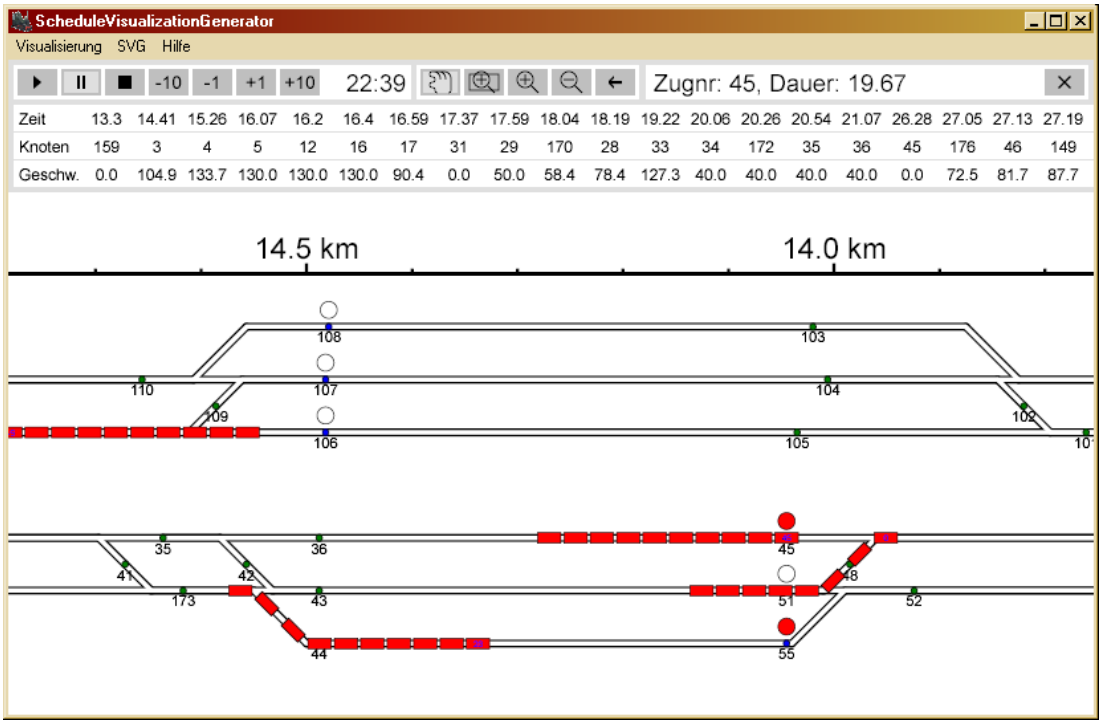

Abbildung 8.12: Screenshot (3) der Applikation

In der Abbildung 8.12 ist ein weiterer Screenshot der Applikation dargestellt. Der Zoomausschnitt zeigt die Haltestelle des Bahnhofs Amsterdam-Schiphol. Der Zug mit Nummer 45 steht an einem roten Signal im Bahnhof. Zu diesem Zug gehören auch die Fahrplaninformationen in der Navigationsleiste. Der Zug mit Nummer 0 hat die Freigabe durch ein grünes Signal bekommen und verlässt sein Haltegleis. Der Zug mit Nummer 23 fährt gerade in sein Haltegleis ein.

9 Fazit

Die Applikation erstellt eine Visualisierung für einen Gleisplan und dem zugehörigen Fahrplan. Als Grafikformat wurde für die Visualisierung SVG gewählt, um die Vorteile eines Vektorformats verwenden zu können, das außerdem sehr interessante Möglichkeiten zur Animation bietet.

9.1 Analysemöglichkeiten der Visualisierung

Die Applikation verwendet als Beispiel die Daten des Bahnhofs Amsterdam-Schiphol, da das Optimierungsprogramm, das den Fahrplan erzeugt, speziell für diesen Bahnhof entwickelt wurde. Die Applikation kann aber auch andere Gleispläne verarbeiten, sofern die Eingabedaten zur Verfügung stehen.

Die Visualisierung für den Bahnhof Amsterdam-Schiphol erreicht eine hohe Qualität und stellt die Bewegung der Züge sehr realistisch dar. Das resultiert vor allem daraus, dass die Abstände der Waggons eines Zuges während der gesamten Animation konstant bleiben, was Ziel der aufwändigen Berechnung der Bézierkurven war.

Die Unübersichtlichkeit der breiten Ausdehnung des Gleisplans kann mit der stufenlosen Zoomingfunktion recht gut ausgeglichen werden. Durch die Interaktionsmöglichkeiten stehen dem Betrachter weitreichende Mittel zur Verfügung, um einzelne Zeit- oder Gleisabschnitte näher zu analysieren.

Ein Zugfahrplan ist ohne eine geeignete Visualisierung nur sehr schwer daraufhin zu analysieren, ob er ein gültiger Fahrplan ist, d.h. ob sich Zugkollisionen im Gleisbild ereignen. Um eine Kollision zu entdecken, wäre es notwendig, die einzelnen Fahrpläne aller Züge paarweise miteinander zu vergleichen. Das Optimierungsprogramm muss natürlich alle Restriktionen berücksichtigen, die im Modell enthalten sind und die dazu führen können, dass ein ungültiger Fahrplan erzeugt wird. Fehler in der Visualisierung deuten daher daraufhin, dass nicht alle Restriktionen berücksichtigt wurden. In einem frühen Stadium der Entwicklung der Visualisierungsapplikation stellte sich heraus, dass das Optimierungsprogramm einen ungültigen Fahrplan generierte. Zwei Züge durchfuhren zeitgleich denselben Block auf dem gleichen Gleis (siehe Abbildung 9.1).

Dieses Problem hätte theoretisch auch an wenigen weiteren Stellen auftreten können. Es wurde gelöst, indem an diesen kritischen Stellen weitere Kontrollknoten in den Graphen des Gleisbildes eingefügt wurden. Kollisionen sind seitdem nicht mehr aufgetreten.

Allerdings kann die Visualisierung nur die Probleme offenlegen, die vom Betrachter in der Visualisierung auch erkannt werden können. Dazu zählt die o.g. Kollision. Theoretisch wäre es nach wie vor möglich, dass ein Zug in einen gesperrten

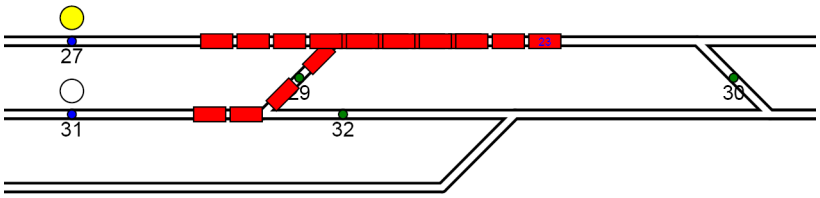

Abbildung 9.1: Zugkollision in Fahrplan

Block einfährt, ohne dass eine Kollision auftritt. Da dem Betrachter ein solches Ereignis nicht sofort auffällt, kann es unentdeckt bleiben. Die Visualisierung ist daher eine Hilfe bei der Analyse der Fahrpläne, nicht aber eine Kontrollinstanz, da sie die zu Grunde liegenden Daten nicht auf Gültigkeit analysiert. Dazu wären weitere Algorithmen nötig.

9.2 Performanz von SVG

Da die SVG-Datei einer Visualisierung auf Grund des XML-Formats in Textform vorliegt, wird sie relativ groß. Die Größe einer SVG-Datei, die die gesamte Animation von einer Fahrplandauer von über zwei Stunden enthält, beträgt ca. 5,6 MB (komprimiert 340 KB). Im Vergleich zu einer leeren SVG-Grafik belegt das DOM dieser SVG-Datei im Adobe SVG-Viewer etwa die Größe von ca. 60 MB.

Die SVG-Grafik enthält zu einem Zeitpunkt ein Animationselement für jeden fahrenden Waggon. Das sind bis zu 80 Einzelanimationen, die gleichzeitig ablaufen. In jedem Zwischenbild muss die Position der Waggons berechnet werden. Die Position eines Waggons wird dadurch angegeben, dass die Bézierkurve ausgewertet wird, die den zurückgelegten Weg definiert. Die Bézierkurve muss aber aus den angegebenen Stützpunkten berechnet werden. Dadurch entsteht für jedes Zwischenbild ein hoher Rechenaufwand. Darüber hinaus werden mit Hilfe des ECMAScripts laufend die Mausaktionen überwacht, mit denen der Benutzer die Visualisierung steuert.

Das Adobe-Plugin, immerhin der momentan leistungsfähigste SVG-Viewer, kann auf weniger leistungsstarken Rechnern die Menge der Rechenoperationen nicht mehr bewältigen, um einen absolut flüssigen Animationsablauf darstellen zu können. Auf einem Rechner mit Intel Celeron Prozessor mit 450 MHz Taktfrequenz kann man zum Beispiel eine deutlich stockende Animation erkennen. Je schneller die Gesamtanimation ist, desto auffälliger wird auch das Stocken. Der Aufruf einer Zoomfunktion führt sogar zu einem längeren Aussetzer. Die Animation hält dabei nicht an, sondern es wird nur eine große Menge von Zwischenbildern ausgelassen, so dass es bei diesen Aussetzern zu Sprüngen in der Bewegung kommt. Die

Performanz ist aber auch von anderen Hardwarefaktoren wie Grafikleistung und Größe des Arbeitsspeichers abhängig. Auf einem Rechner mit schnellerem AMD Athlon 2100+ Prozessor ließ sich aufgrund einer betagten Grafikkarte sogar eine Performanzverschlechterung gegenüber dem Celeron-Rechner feststellen. Auf heute erhältlichen, leistungsstarken Rechnern lässt sich aber eine Visualisierung erreichen, die nahezu nicht mehr stockt, sondern sehr flüssig abläuft.

A Inhalt der CD-Rom

arbeit	**Diplomarbeit**
arbeit/latex	Latex-Quellen und Grafiken
arbeit/pdf	PDF-Version
daten	**Anwendungsdaten der Applikation**
daten/gleisplaene	Gleispläne und DTD
daten/fahrplaene	Fahrpläne
daten/svg	erstellte SVG-Dateien
classes	**Java-Klassen**
classes/app	Klassenhierarchie der Applikation `scheduleVisualizationGenerator`
libs	**Bibliotheken**
libs/app	Klassen der Applikation mit Startskript
libs/swt	SWT-Bibliotheken
libs/xerces	Xerces2-Bibliotheken
docs	**Dokumentation**
docs/app	API der Applikation
docs/java	Java 2 API
docs/eclipse	Eclipse API (enthält API des SWT)
docs/xerces	Xerces2 API
software	**benötigte Software**
software/java	Java 2 SDK 1.4.2
software/adobe	Adobe SVG Viewer 3.02 und 6 beta

B Literaturverzeichnis

Aufgrund der Aktualität der in dieser Diplomarbeit eingesetzten Techniken sind viele Informationen noch nicht in Büchern zu finden. Daher beziehen sich die meisten Literaturverweise auf Websites, deren Erreichbarkeit und Inhalt sich schnell ändern können. Zum Zeitpunkt der Erstellung dieser Diplomarbeit waren alle Quellen unter den hier aufgeführten Links erreichbar.

Literatur

[Ado2005] **Adobe Systems**
Adobe SVG Zone
http://www.adobe.com/svg

[Apac2004] **Apache Foundation**
Apache Xerces2 Java Parser
http://xml.apache.org/xerces2-j

[Apac2004b] **Apache Foundation**
Batik SVG Toolkit
http://xml.apache.org/batik

[Apti2004] **Aptico GmbH**
SVG Tutorial
http://svg.tutorial.aptico.de

[BrKaSt2003] **Brucker, Peter; Kampmeyer, Thomas; Strotmann, Christian**
Schiphol Tunnel 2007 - A theortical model and first versions of solution methods
Universität Osnabrück 2003

[Daum2004] **Daum, Berthold; et al.**
Web-Entwicklung mit Eclipse
Dpunkt Verlag, Heidelberg 2004

[Ecma2004] **ECMA International**
Standard ECMA-262: ECMAScript Language Specification
http://www.ecma-international.org/publications/standards/Ecma-262.html

[Eise2002] **J. David Eisenberg**
SVG Essentials
O'Reilly, Köln 2002

[HaMe2004] **Harold, Elliotte Rusty; Means, W. Scot**
XML in a nutshell, 3rd Edition
O'Reilly, Sebastopol 2004

[Hat2004] **Hatton, Tim**
SWT - A Developer's Guide
O'Reilly, Sebastopol 2004

[Hei2003] **Heise Newsticker**
Microsoft, das Grafikformat SVG und Longhorn
http://www.heise.de/newsticker/meldung/42378

[Holz2004] **Holzner, Steve**
Eclipse
O'Reilly, Sebastopol 2004

[Kart2003] **Kartographen im Netz**
Kartographie im Internet auf Vektorbasis, nur mit Hilfe von SVG möglich
http://www.carto.net/papers/svg/index_d.html

[Mei2005] **Meinike, Thomas**
SVG - Learning by Coding
http://www.datenverdrahten.de/svglbc

[Moz2004] **Mozilla Organisation**
Mozilla SVG Project
http://www.mozilla.org/projects/svg

[SAX2004] **Mailingliste** `xml-dev`
Simple API for XML
http://sax.sourceforge.net

[Schw2004] **Schwarz, Hans Rudolf**
Numerische Mathematik
B.G. Teubner, Stuttgart 2004

[Vorn2004] **Vornberger, Oliver**
Vorlesung Computergrafik
http://www-lehre.inf.uos.de/ cg

[W3C2005] **World Wide Web Consortium**
World Wide Web Consortium
http://www.w3c.org

[W3C2005a] **W3C Recommendation**
Extensible Markup Language (XML)
http://www.w3.org/TR/REC-xml

[W3C2005b] **W3C Recommendation**
Scalable Vector Graphics (SVG) 1.1 Specification
http://www.w3c.org/TR/SVG11

[W3C2005c] **W3C Recommendation**
Mobile SVG Profiles: SVG Tiny and SVG Basic
http://www.w3c.org/TR/SVGMobile

[W3C2005d] **W3C**
SVG Implementations
http://www.w3.org/Graphics/SVG/SVG-Implementations.htm8#viewer

[W3C2005e] **W3C Recommendation**
Cascading Style Sheets (CSS), level 2
http://www.w3c.org/TR/REC-CSS2

[W3C2005f] **W3C Recommendation**
Document Object Model (DOM), level 3
http://www.w3.org/TR/DOM-Level-3-Core

[W3C2005g] **W3C**
XML Schema
http://www.w3.org/XML/schema

[WaLi2004] **Watt, Andrew H.; Lilley, Chris; et al.**
SVG Unleashed
SAMS, kkk 2004

Erklärung

Hiermit erkläre ich, dass ich die Diplomarbeit selbstständig angefertigt und keine anderen Quellen und Hilfsmittel außer den in der Arbeit angegebenen benutzt habe.

Osnabrück, den 27.01.2005 .....

Patrick Fox