Ein Multi-Selection-Tool für GEF

Christian Wilms

6053770

Projekt
Animationswerkzeug zur Visualisierung
sozialen Handelns

Wintersemester 2010/11

Dozenten:
Dr. Carola Eschenbach
Felix Lindner

Arbeitsbereich WSV
Department Informatik
Universität Hamburg

Inhaltsverzeichnis

1 Einleitung

Mit dem Fortschreiten der Mensch-Roboter-Interaktion und dem verstärkten Einsatz von Robotern im Alltag des Menschen, wird die möglichst realistische Bewegung von Robotern immer wichtiger. Hierbei geht es nicht nur um möglichst geschmeidige Bewegungen von Armen und Beinen oder eine möglichst natürliche Art der Fortbewegung, sondern auch um sozial angemessenes räumliches Verhalten, das auch bei nicht humanoiden Robotern wichtig ist. Zu diesem räumlichen Verhalten gehören unter anderem das Zugehen auf Menschen oder das Ausweichen in engen Fluren [Pacchierotti u. a., 2005]. Sozial angemessenes Verhalten ist besonders bei humanoiden Robotern wichtig, da diese sonst ins Uncanny Valley stürzen, weil sie zwar menschenähnlich sind, sich aber nicht vertraut verhalten [Gee u. a., 2005].

Um das räumliche Verhalten von Robotern zu testen, werden vielfach aufwendige Experimente mit echten Robotern in realen Umgebungen durchgeführt. Da diese Experimente zumeist sehr teuer sind, ist eine Alternative zu suchen. Diese Alternative könnten kurze Animationsfilme sein, die die Situation auf eine zweidimensionalen Sicht projizieren.

Diese Art von Animationen, wie sie z.B. auch Heider und Simmel [Heider u. Simmel, 1944] konstruiert haben (siehe Abbildung 1), lassen sich nutzen, um soziales Verhalten ihrer Akteure durch Menschen einzuschätzen. Dabei gibt es neben den Animationen von Heider und Simmel weitere Beispiele, die zeigen, dass selbst einfachen geometrischen Figuren Absichten und Lebendigkeit zugeschrieben werden oder Interaktion zwischen den Figuren wie z.B. beim "launching effect" von Michotte [Scholl u. Tremoulet, 2000] wahrgenommen werden.

Zur Erzeugung von Animationssequenzen gibt es mehrere Formate und Möglichkeiten, die aber meist höheres technisches Verständnis jener Formate erfordern. Deshalb ist es Ziel dieses Projekts, ein Programm zu entwickeln, mit dessen Hilfe es auch Laien auf dem Gebiet der Animationserzeugung möglich ist, kleine Animationen in der Art von Heider und Simmel zu erstellen.

Dieses Programm besteht dabei aus drei durch jeweils eigene Tabs klar voneinander getrennten Teilen. Zum einen gibt es den Grafikeditor, der im Fokus dieser Arbeit stehen soll, daneben gibt es die Komponenten Animationseditor und Player. Der Animationseditor dient der Erzeugung von Animationen zu vorher erstellten Objekten, wobei es verschiedene Animationen gibt, wie z.B. Bewegungsanimationen oder Farbanimationen. Näheres zu Animationen und dem Animationseditor ist in der Arbeit von Stefan Schäfermeier [Schäfermeier, 2011] zu lesen. Als dritte Komponente ist ein Player in das Programm integriert, der zum Abspielen der Animation und zum Exportieren in ein gängiges Videoformat genutzt werden kann, was in der Arbeit von Christian Donocik [Donocik, 2011] näher erläutert wird.

Diese Arbeit stellt einen Teil des im Projekt erarbeiteten Programms vor mit dem Fokus auf dem Grafikeditor. Zunächst werden dafür einige Grundlagen gelegt und Teile des benutzten Frameworks GEF (Graphical Editing Framework) erläutert. Danach wird auf den Grafikeditor fokussiert, um einige allgemeine Details dieses Editors zu erläutern. Eine weitere Fokussierung erfolgt dann in den

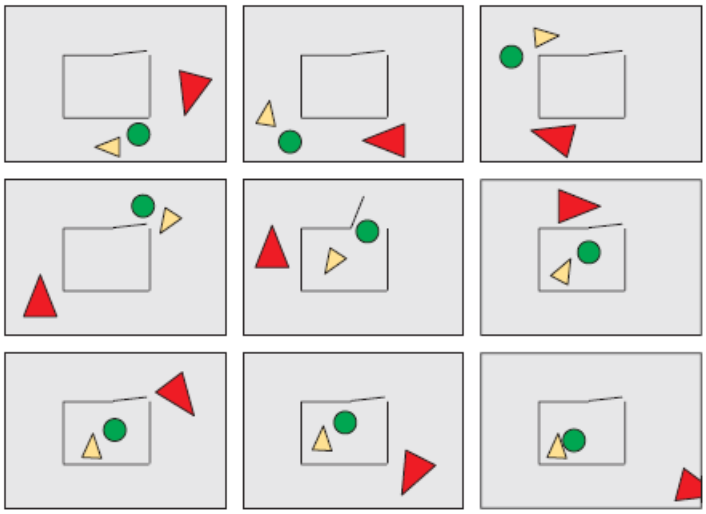

Abb. 1. Filmausschnitte aus einer Animation ähnlich denen von Heider und Simmel. Einige Beobachter nehmen wahr, dass das große Dreieck das kleine verfolgt und schreiben den Objekten Ziele sowie Absichten zu, Quelle [Scholl u. Tremoulet, 2000]

folgenden zwei Kapiteln, in denen zum einen ein neues Tool zur Erzeugung von speziellen Formen, unter anderem Polygonen und Pfaden, vorgestellt wird und zum anderen eine spezielle Form von Pfaden, runde Pfade, detailliert beschrieben wird. Abschließend gibt es eine kurze Evaluation.

2 Grundlagen

Zunächst werden nun einige für die Erstellung des Programms grundlegende Techniken näher erläutert. Im Zentrum dieser Grundlagen steht zum einen SVG mit SMIL als bereits bestehende, relativ einfache Datenformate zur Erzeugung und Speicherung von Animationen. Zum anderen liegt der Fokus auf dem Framework GEF, das als Grundlage für das spätere Programm dient und dessen Architektur maßgeblich vorgibt.

2.1 SVG und SMIL

Scalable Vector Graphics (SVG) [SVG, 2010] ist ein XML basiertes Datenformat, das zum Erzeugen von zweidimensionalen Vektorgrafiken genutzt werden kann. Es bietet vielfältige und sehr flexible Möglichkeiten verschiedene, einfache geometrische Formen, wie sie in den in der Einleitung beschriebenen Animationen zu finden sind, auf eine einfache Weise zu erzeugen. Dies wird gewährleistet durch die hierarchische XML-Notation mit Elementen/Tags und Attributen, die auch für Menschen recht gut lesbar ist. Hierzu ein Beispiel für die Erstellung

eines roten (RGB-Wert 255,0,0) Kreises mit Radius 30 und dem Mittelpunkt an der Koordinate $(100, 50)$:

```
<circle cx="100" cy="50" r="30" fill="rgb(255, 0, 0)" />
```

Zudem sind bereits in SVG Animationen wie Transformationen verankert und können für die Animationen genutzt werden. Die Synchronized Multimedia Integration Language (SMIL) [SMI, 2008], eine ebenfalls XML-basierte Sprache zur Animation von Objekten, liefert weitere Spezifikationen für ausgefeiltere Animationen. Diese beinhalten unter anderem Animationen von Farben oder Bewegungen entlang von Pfaden mit Timing und Synchronisation. Besonders interessant ist dabei die Bestimmung des Timing mittels eines Splines, so dass die Geschwindigkeit auch langsam erhöht oder gesenkt werden kann, um realistische Beschleunigungen zu simulieren.

Die Entscheidung für SVG mit SMIL als Datenformat für die Animationen ist durch mehrere Faktoren begründet. Zum einen ist die Anzeigbarkeit von SVG-Dokumenten in den meisten Standardbrowser, besonders Opera, ein sehr wichtiges Kriterium, da hierdurch eine sehr einfache Wiedergabe gewährleistet ist. Zum anderen ist das Erstellen von SVG-Dokumenten in Kombination mit SMIL relativ einfach, da keine neue Sprache erlernt werden muss, sondern nur die zum Teil vorhandenen XML-Kenntnisse erweitert werden müssen. Außerdem gibt es mit dem Batik-Toolkit [Bat] bereits eine bestehende Bibliothek für die Anbindung von SVG an Java. Eine andere Möglichkeit Animationen zu erzeugen wäre z.B. die Benutzung eines Raytracer [Hansmann, 2010] wie POV-Ray [POV], was allerdings wesentlich komplizierter gewesen wäre, da dieser viel komplexer zu handhaben ist und auch keine einfache Anbindung an Java vorhanden ist.

2.2 GEF

Als Grundlage des während des Projekts entwickelten Programms dient das Graphical Editing Framework (GEF) [GEF] [Ares, 2008] [Mammana u. a., 2007], eine Eclipse basierte Umgebung zur Erstellung von grafischen Editoren. Hierzu bietet es eine grundlegende Architektur und viele hilfreiche Details, die die Erstellung eines gut benutzbaren Editors erheblich vereinfachen.

Die durch GEF vorgegebene Grundarchitektur basiert maßgeblich auf dem Model-View-Controller-Pattern, der Trennung des Modells von der Repräsentation und der Steuerung. Weitere wichtige Patterns, auf denen die Architektur von GEF basiert und die in das MVC-Pattern integriert sind, sind das Command-Pattern, das Chain-of-responsibility-Pattern und das Observer-Pattern. Mehr zu den hier erwähnten Design Patterns findet sich auch im Buch “Head First Design Patterns” [Freeman u. a., 2004].

Als weitere Bibliotheken nutzt GEF unter anderem Draw2d [Dra], eine auf SWT aufsetzende, leichtgewichtige Bibliothek zum Rendern und Anordnen von Elementen. Draw2d unterstützt dabei das einfache Erzeugen und Positionieren von Figuren in GEF auf einer SWT-Canvas.

Grundlegende Elemente
Die Basis von GEF bilden drei Elemente, die durch das erwähnte MVC-Pattern vorgegeben sind. Zum einen sind dies die Modellklassen, die ausschließlich für das Halten des Zustands eines fachlichen Elements zuständig sind. Es ist wichtig dabei zu beachten, dass die Modelle, ähnlich wie in SVG, hierarchisch wie in einem Baum geordnet sind. Jedes Modell, daher jeder Knoten in diesem Baum, hat einen Elternknoten, mit Ausnahme der Wurzel des Baums, und eine Menge von Kindern, die auch leer sein kann. Ein großer Vorteil hierbei ist, dass GEF keine Oberklassen vorgibt oder Schnittstellen, was den Entwurf einer ganz individuelle Vererbungshierarchie möglich macht.

Ein weiteres Basiselement aus dem MVC-Pattern und auch in GEF ist die Figur. Figurobjekte sind die Repräsentationen genau eines Modells für die Benutzungsschnittstelle und beinhalten daher meist keine Logik. Auch hier kann man völlig individuell arbeiten oder man bedient sich der Formen der Bibliothek Draw2d. Diese bietet viele grundlegende Formen an, wie z.B. Ellipsen, Rechtecke oder Polygone. Diese können dann über eine Veränderung des Zustands individualisiert oder durch Vererbung erweitert werden, um Effekte hinzuzufügen oder bestehende Effekte individuell anzupassen.

Das dritte Basiselement ist der EditPart, der im MVC-Pattern üblicherweise Controller heißt. Dieser stellt die Kommunikation zwischen dem Modell auf der einen Seite und der Figur auf der anderen Seite sicher. Wichtig dabei ist, dass jedes Exemplar eines Modells seinen eigenen EditPart hat. Um die Kommunikation zu gewährleisten sind allerdings noch drei weitere Elemente in GEF grundlegend.

Requests spielen eine wesentliche Rolle in der Kommunikation von GEF. Sie werden bei der Benutzung eines Werkzeugs, z.B. des Auswahlwerkzeugs, durch den Benutzer erzeugt und starten eine Kette, an deren Ende meist das visuelle Feedback einer erfolgreichen Aktion auf dem Bildschirm steht. Erzeugt werden die Requests dabei im benutzten Werkzeug, das sie auch mit Informationen anreichert. Des Weiteren spielen Commands eine wichtige Rolle für die Kommunikation in GEF. Diese beschreiben, welche Operationen tatsächlich ausgeführt werden sollen, wenn ein bestimmtes Werkzeug benutzt und eine Aktion ausgeführt wurde. Erzeugt werden die Commands durch EditPolicies, dem letzten grundlegenden Baustein, der hier Erwähnung findet, auf Basis von Requests. EditPolicies werden dabei von den jeweiligen EditParts gehalten.

Kommunikation
An dieser Stelle soll einmal exemplarisch der Kommunikationsweg von der Aktion des Benutzers bis zum visuellen Feedback grob dargestellt werden. Zunächst wählt der Benutzer ein Werkzeug aus und initiiert eine Aktion an einem bestehenden Objekt. Darauf erzeugt das Werkzeug einen Request und füllt ihn mit den Information, die das Objekt nach seiner Änderung beschriebt. Außerdem wird der EditPart des Zielobjekts durch das Werkzeug ermittelt, um die Änderungen später durchführen zu können. Die gespeicherten Informationen können Größenangaben, Positionsangaben oder ähnliches sein. Die Infor-

mationen dieses Requests werden mit Hilfe von Constraints in ein Command gespeichert, das vom EditPart bzw. von einer EditPolicy erzeugt wird. Diese EditPolicy wird durch den EditPart ausgewählt und entscheidet dann selbst in Abhängigkeit des Requests welches Command benötigt wird. An dieser Delegation von Kompetenz wird auch das Chain-of-responsibility-Pattern deutlich, das, wie bereits erwähnt, in GEF umgesetzt wurde. Auch das Command-Pattern ist hier schon zu erkennen.

Der Command wird anschließend auf den Commandstack gelegt und durch das Werkzeug zur Ausführung gebracht, sofern das Command ausführbar ist. Jetzt wird eine oder mehrere Methoden an dem betreffenden Modell ausgeführt, wie im Command festgelegt. Sollte sich das Modell dadurch ändern, wird dies vom EditPart erkannt, da dieser ein Beobachter des Modells ist. Hier wird das Observer Pattern deutlich, um die Repräsentation von der Logik und den Daten zu trennen.

Der EditPart entscheidet dann, ob es sich um eine Veränderung handelt, auf die zu reagieren ist und zeichnet entsprechend dieser Entscheidung die Figur neu, wenn dies benötigt wird. Die Änderung des Modells muss nicht unbedingt durch Interaktion des Benutzer entstehen, sondern kann auch durch anderes Verhalten hervorgerufen werden, der Ablauf danach bleibt jedoch gleich.

Weiterführende Möglichkeiten

Neben der beschriebenen Architektur, bietet GEF, wie bereits erwähnt, die einfache Implementierung kleiner Details, die die Benutzung eines Editors einfacher machen. Zu diesen Details zählen unter anderem der Commandstack und die einfache Möglichkeit des Undo und Redo durch bereits vorimplementierte Buttons. Ähnlich verhält es sich mit gewohnten Details wie einem Löschen-Button, Zoom oder Property Sheets. Zudem bringt GEF bereits Werkzeuge zur Auswahl bzw. Markierung sowie zur Erzeugung von Objekten und Paletten für die Anordnung von diesen Werkzeugen mit. Auch viele weitere Details lassen sich mit GEF einfach umsetzen. Diese sollen hier aber nicht weiter ausgeführt werden, da sie für das Projekt keine maßgebliche Rolle spielen.

Die Entscheidung für GEF als Grundlage des Editors fußt primär auf zwei Gründen. Zum einen bietet es eine softwaretechnisch gute und durchdachte Architektur. Dies erspart die Erstellung einer vollständig neuen Softwarearchitektur für das Programm, was sehr viel Zeit kosten würde und vermutlich nicht zu so einem hochwertigen Ergebnis führen würde, da sich, wie bereits beschrieben, in der GEF-Architektur zahlreiche wichtige Design Patterns finden.

Zum anderen sind die vielen kleinen Details, die sich einfach implementieren lassen, ein großer Vorteil und ein gutes Argument für das Framework. Auch die Erstellung von Canvas und grundlegenden Werkzeugen hätte sonst viel Zeit in Anspruch genommen, die an anderer Stelle gefehlt hätte. Ebenso verhält es sich unter anderem mit einer reibungslosen Undo- und Redo-Funktionalität.

Insgesamt bietet GEF eine gute Grundlage, um am Ende des Projekts ein durchaus vorzeigbares Ergebnis präsentieren zu können, für das ohne das Fra-

mework sehr viel mehr Zeit benötigt worden wäre. Weitere Details zu GEF im Zusammenhang mit der Entwicklung von Editoren sowie eine Evaluation dazu findet sich in der Arbeit von Sven Röhling [Röhling, 2011].

3 Der Grafikeditor

Beim Erstellen einer Animation ist meist der erste Schritt das Erzeugen der grafischen Objekte, die in der späteren Animation entweder statisch immer am selben Punkt bleiben oder, zum Teil entlang von angelegten Pfaden, sich über den Bildschirm bewegen. Die Erzeugung von zunächst immer statischen Objekten geschieht im Grafikeditor, der in Abbildung 2 zu sehen ist. Dafür bietet der Grafikeditor zehn verschiedene Formen von Objekten an: Kreis, Ellipse, Rechteck, Quadrat, Dreieck, Polygon, Linie, Polylinie, Pfad und runder Pfad, die jeweils über Werkzeuge am linken Rand auszuwählen sind.

Diese Objekte lassen sich auf zwei Weisen unterschieden. Zum einen gibt es Formen, die direkt den Formen des SVG-Formats entsprechen, und synthetische, wie das Dreieck und das Quadrat, die aus SVG-Formen abstrahiert wurden und nur in den modellierten Java-Modellen existieren. Eine andere und für die Arbeit wesentliche Unterscheidung ist die in Formen, die über eine Punktliste definiert sind, und Formen, die über einen Punkt und weitere Eigenschaften definiert sind. Diese Unterscheidung wird in den folgenden beiden Abschnitten deutlich.

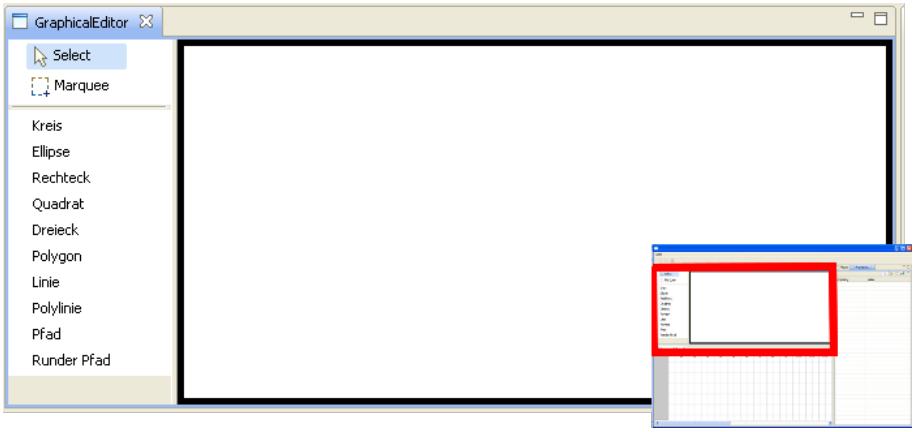

Abb. 2. Der Grafikeditor mit der Canvas rechts und den Werkzeugen für die Formen und die Auswahl links. Unten rechts befindet sich die Übersicht der Gesamtoberfläche.

3.1 Erzeugung von Basic Shapes

Basic Shapes von SVG sind die Formen Kreis, Ellipse und Rechteck. Diese Formen haben alle gemein, dass sie über einen Punkt und weitere Eigenschaften

definiert sind. Dieser Punkt ist dabei entweder die linke obere Ecke wie beim Rechteck oder der Mittelpunkt wie bei den anderen beiden Formen. Die weiteren Eigenschaften sind die Ausdehnung oder ein bzw. zwei Radien.

Eine weitere wesentliche Eigenschaft, die diese Formen gemein haben, ist, dass man sie alle aus einem Rechteck berechnen kann. Das heißt zum Erzeugen einer dieser Formen ist lediglich das Aufziehen eines Rechtecks nötig, aus dem sich alle wichtigen Informationen bezüglich der Größe berechnen lassen. Bei der Rechtecksform ist dies besonders einfach, da sie dem aufgezogenen Rechteck entspricht. Auch die Ellipse kann direkt in das Rechteck gelegt werden mit dem Mittelpunkt im Mittelpunkt des Rechtecks und den beiden Radien als halbe Höhe bzw. Breite. Beim Kreis muss man sich entscheiden, an welcher Kante man sich bei der Bestimmung des Radius orientieren will. Hier ist dies stets die kürzeste Kante.

Auf Grund dieser Eigenschaft, ist die Nutzung des von GEF vorgegebenen Standardwerkzeugs möglich, da dies ein Rechteck aufziehen und die Information daraus weiterleiten kann. Das hier benutzte Werkzeug ist das `CreationTool`, welches mit Hilfe eines EditParts ein Modell erzeugt oder verändert. Wenn noch kein EditPart besteht, erzeugt das `CreationTool` mittels der übergebenen Factory einen EditPart für das gewünschte Modell.

Die Factory ist dabei ein sehr wichtiges Element, da im `CreationTool` selbst nicht definiert ist, was erzeugt wird, es ist lediglich eine Hülle. Die Factories in unserem Projekt sind alle vom Typ `NodeCreationFactory` und erhalten einen Modell-Typ, der bestimmt, was die Factory erzeugt. Die Factory wird dem `CreationTool` über den `CreationToolEntry` übergeben. Das hier gewählte Design entspricht dem Factory-Pattern [Freeman u. a., 2004], wobei die hier verwendete Factory nur ein Objekt erzeugt und primär aus einer großen Fallunterscheidung besteht, in Kombination mit einer Form von selbst erzeugter parametrischen Polymorphie [Poetzsch-Heffter, 2009]. Eine Alternative dazu sind Javas Generics, die ab Java 5 verfügbar sind.

Von besonderer Bedeutung im `CreationTool` sind die Methoden, die das Drücken der Maustasten verarbeiten. Die Methode `handleButtonDown(int)` wird dabei gerufen, wenn eine Maustaste gedrückt wird und reagiert nur auf den Druck der linken Maustaste, da das Standardwerkzeug nur mit der linken Maustaste bedient wird. Wird das Werkzeug korrekt bedient, wird zunächst ein Punkt in den `CreateRequest` geschrieben. Sollte dieser noch nicht bestehen, wird er ebenfalls mit Hilfe der oben erwähnten Factory erzeugt. Anschließend wird der EditPart des Ziels gesperrt, um die Aktualisierung der Figur während des Aufziehens des Rechtecks zu verhindern. Das Aufziehen arbeitet dabei mit Drag&Drop und verschiedenen Zuständen für die einzelnen Phasen der Operation. Während des Ziehens werden die Größeninformationen im Request durch die Methode `updateTargetRequest()` im `CreationTool` ständig angepasst.

Wird die Operation erfolgreich abgeschlossen und die linke Maustaste wieder losgelassen, wird die Methode `handleButtonUp(int)` gerufen. Diese bewirkt, dass der EditPart wieder freigegeben wird, um eine Aktualisierung der Figur zuzulassen. Danach wird die Methode `performCreation(int)` gerufen, die das

aktuelle Command ausführt, sofern es ausführbar ist. Dieses Command ist vom Typ `CreationCommand` und ist mit einem Feld für das zu erzeugende Modell ausgestattet, das es in der Hierarchie unter dem Elternknoten einhängt. Die Größeninformationen werden hier nicht noch einmal gesetzt, da sie während des Aufziehens des Rechtecks ständig aktualisiert wurden.

Die Folge dieser Architektur ist, dass das Erzeugen des Modells nicht im `CreateCommand` geschieht, sondern schon bei der Aktivierung des Werkzeugs. Daher wird das Modell bei der Benutzung des Werkzeugs bzw. der Bewegung der Maus ständig aktualisiert. Das `CreateCommand` macht das Modell dann nur noch in der Hierarchie des Gesamtmodells bekannt. Der Name CreateCommand kann daher etwas missverständlich sein.

3.2 PointList basierte Modelle

Die bisher diskutierten Modelle, die der Basic Shapes, sind alle definiert über einen Punkt und ein oder zwei weitere Größen, beispielsweise Mittelpunkt und Radius oder zwei Radien. Eine solche Art der Definition ist bei Formen wie Polygonen oder Polylinien nicht mehr möglich. Bei einem solchen Form-Typ bietet sich bei der Modellierung die Definition über Punkte an. Dies können je nach Form Anfangspunkt, Endpunkt und Stützpunkte sein wie bei Polylinien oder Eckpunkte wie bei Polygonen.

In SVG geschieht dies so bei den Formen Line, Polyline, Polygon und Path. Dabei gibt es unterschiedliche Arten der Speicherung der Punktliste. Beim einfachsten Formtyp, den Linien, wird jeweils Anfangs- und Endpunkt als Koordinate von x und y angegeben. Die Formen Polyline, Polygon und teilweise auch Path werden über eine Liste von Punkten gespeichert, wobei dort die Koordinaten einfach konkateniert werden. Hier ein Beispiel für einen Polygon mit fünf Eckpunkten:

```
<polygon points="57,136 141,61 179,104 164,163 117,170"/>
```

Ein etwas anderes Prinzip wird bei der Speicherung von abgerundeten Pfaden angewendet, da spezielle mathematische Gegebenheiten ausgenutzt werden können. Dies wird in einem späteren Kapitel detaillierter behandelt.

Die Speicherung in unseren Modell-Klassen in Java ist bei allen eben betrachteten Formen gleich. Es wird jeweils ein Exemplar der Klasse `PointList` aus der Draw2d-Libary in einem Feld gespeichert und mit den Punkten, die die Form definieren, gefüllt. Hier liegt ein Unterschied zwischen unseren modellierten Java-Modellen und SVG. Der Vorteil der hier benutzten Definition und Speicherung ist eine einfach realisierbare Vererbung. Man kann daher einen abstrakten Obertypen und darunter die einzelnen Modell-Klassen definieren, was bisher jedoch nur zwischen `Path` und `RoundedPath` genutzt wird.

Neben den Punkten wird ein Objekt auch über weitere Eigenschaften definiert, wie z.B. die Farbe. Diese haben allerdings für das Thema dieser Arbeit keine große Bedeutung, weshalb hier auch nicht weiter darauf eingegangen wird. Näheres zur Modellstruktur ist in der Arbeit von Tobias Staron [Staron, 2011] zu lesen.

4 Multi-Selection-Tool

Im vorherigen Kapitel wurde ein deutlicher Unterschied zwischen Formen wie Kreis und Rechteck auf der einen Seite und Pfad und Polygon auf der andere Seite herausgearbeitet. Der deutliche Unterschied liegt in der Möglichkeit, Kreise oder Rechtecke mit Hilfe von Rechtecken zu definieren und zu erzeugen, Pfade, Polygone und die anderen `PointList` basierten Modelle jedoch nicht. Dieses Problem zieht sich nicht nur durch die Modellierung der Modellklassen, sondern auch durch die Erzeugung der Formen. Eine Verwendung des beschriebenen von GEF bereits mitgelieferten `CreationTool` ist hier deshalb nicht möglich. Aus diesem Grund ist die Entwicklung eines weiteren, individuellen Erzeugungstools, dem Multi-Selection-Tool, das speziell auf die Erzeugung von `PointList` basierten Formen abgestimmt ist, für das Programm nötig.

4.1 Funktionsweise

Die Anforderungen des neuen Erzeugungstool sind relativ klar abgesteckt. Es soll ein einfach zu benutzendes Werkzeug sein, das sich ähnlich dem bekannten GEF `CreationTool` verhält und benutzen lässt. Dabei soll die Erzeugung von Linien, Polylinien, Polygonen und Pfaden, eckig wie rund, unterstützt werden, wobei es für die Erzeugung von Linien eine Unterform des eigentlichen Werkzeugs gibt, die als Subklasse modelliert ist und lediglich eine Methode neu definiert. Ebenso verhält es sich mit dem Werkzeug für runde Pfade, was im nächsten Kapitel eingeführt wird.

Um eine möglichst einfache Benutzung der neuen Werkzeuge zu gewährleisten, ist die Steuerung der Werkzeuge ausschließlich über die Maus sinnvoll. Eine andere Möglichkeit wäre das Benutzen der Strg-Taste der Tastatur, allerdings wäre dies ein deutlicher Unterschied zur Benutzung der bisherigen Werkzeuge, die sich alle ebenfalls ausschließlich über die Maus steuern lassen.

Die Benutzung des Werkzeugs ist in zwei Phasen zu unterteilen. Zum einen gibt es die Phase der Erzeugung der Figur, dem Setzen der Punkte, was durch den Druck der linken Maustaste umgesetzt werden kann. Die andere Phase des Werkzeugs ist das korrekte Beenden der Erzeugung, was einen Unterschied zu den bisher vorhandenen GEF Werkzeugen darstellt. Bei dem bisher bekannten GEF `CreationTool` ist die Erzeugung beendet, wenn der Druck auf die linken Maustaste beendet wurde, dies ist bei der Erzeugung von Figuren durch mehrere Punkte jedoch nicht praktikabel, weshalb die Phase des manuellen Beendens nötig wird. Ein inkorrektes Beenden des Werkzeug, z.B. durch das direkte Klicken auf ein anderes Werkzeug, führt unter Umständen zu nicht definiertem Verhalten. Meist entsteht jedoch kein Problem, da sowohl Modell als auch SVG-Repräsentation nach dem Anlegen eines Punktes immer einen korrekten Zustand haben. Ein unter Umständen irritierendes Verhalten entsteht allerdings beim Anlegen eines runden Pfads, da hier durch die mathematischen Gegebenheiten manchmal automatische Aktionen ablaufen, die im nächsten Kapitel verdeutlicht werden. Dadurch entsteht unter Umständen ein nicht korrektes Modell in

den Java-Objekten, was jedoch keine gravierenden Auswirkungen hat. Im SVG-Dokument ist das Modell aber weiterhin korrekt. Bei der Erzeugung der Linie ist ein manuelles Beenden des Werkzeugs nicht nötig, da die Anzahl der zu setzenden Punkte feststeht.

4.2 Innerer Aufbau

Beim inneren Aufbau des Werkzeugs muss noch einmal deutlich zwischen den Varianten unterschieden werden. Zunächst wird das Werkzeug zur Erzeugung von Polylinien, Polygonen und Pfaden betrachtet, das `PLCreationTool`, dessen Name von den `PointList` basierten Modellen abgeleitet ist. Im Anschluss daran wird das `LineCreationTool`, das Werkzeug zur Erzeugung von Linien vorgestellt, wobei lediglich die Unterschiede dargestellt werden, um Redundanzen möglichst gering zu halten. Das `RoundedPLCreationTool` wird erst im nächsten Kapitel vorgestellt, da hierbei Details der runden Pfade benötigt werden.

PLCreationTool
Die Klasse `PLCreationTool` ist in der Hierarchie der GEF-Werkzeuge eingebettet und steht in dieser als eine weitere Unterklasse des `TargetingTool` neben dem `CreationTool`, das für die Erzeugung der Basic Shapes mittels eine Rechtecks zuständig ist. Die Vorteile liegen darin, dass dadurch einige nützliche Methoden aus den Oberklassen benutzt werden können und somit schon eine gewisse Basis innerhalb der GEF Umgebung für das neue Tool besteht. Zudem muss die Klasse `Tool` von GEF eine Oberklasse sein, um auf die vorhandenen Mechanismen bei den Entrys der Werkzeuge zurückgreifen zu können. Eine Ansiedlung des `PLCreationTool` unter dem `CreationTool` wäre auch möglich, jedoch weniger sinnvoll, da dann viele Methoden mit leeren Methodenrümpfen überschrieben werden müssten, um die gleiche Funktionalität zu erreichen.

In vielen dieser Methoden ist auch einer der größten Unterschiede zu erkennen zwischen den beiden Werkzeugen. Das `PLCreationTool` benötigt keine Drag&Drop-Funktionalität, im Gegensatz zum bisherigen Tool, das maßgeblich darauf aufbaut. Der Effekt daraus ist, dass die Komplexität für das neue Werkzeug sinkt, was die Implementation erleichtert.

Ein weiterer großer Unterschied besteht darin, dass das neue Werkzeug beim Klick des Benutzers entscheiden muss, ob ein bereits bestehendes Objekt geändert werden muss oder ob ein ganz neues zu erzeugen ist. Diese Logik ist in der Methode `handleButtonDown(int)` implementiert. Hier wird zunächst entschieden, ob die linke Maustaste heruntergedrückt wurde. Wenn dem so ist, kommt die wichtige Fallunterscheidung zwischen Erzeugung und Änderung, wobei dies über den Request und dem darin gespeicherten Modell entschieden wird. Ein Modell, was noch keine Punkte enthält, ist dabei ein noch nicht erzeugter Pfad und daher noch nicht in der internen Objektstruktur gespeichert. Dieser Durchgriff ist softwaretechnisch nicht ideal, aber der einzige Weg diese Frage zu klären. Eine doppelte Buchführung ist als Alternative nicht sehr sinnvoll, da sich dann Probleme beim Undo/Redo ergeben.

Je nach Ausgang der Fallunterscheidung wird der Typ des Requests über eine String-Konstante entweder auf Erzeugung oder Änderung gesetzt, was maßgeblich für weitere Behandlung des Requests in den EditPolicies ist. Die String-Konstanten für die neuen Typen eines Requests stammen aus der Erweiterung der `RequestConstants` den `ExtendendRequestConstants`. Die Differenzierung der Typen von Requests bewirkt in der Methode `createCommand(Request)` der Klasse `ChangeGraphEditorLayoutPolicy`, dass entsprechend des Status des Objekts das richtige Command erstellt und zurückgegeben wird. Das so erzeugte Command wird mit dem Modell sowie den Punkten befüllt bzw. dem Elternknoten in der Hierarchie, falls es sich um ein neues Objekt handelt. Dieses Command wird an das Werkzeug zurückgegeben, das es mit der Methode `getCommand()` über den EditPart des Objekts bei der EditPolicy angefordert hat. Diese Unterscheidung der Commands hat auch erhebliche Vorteile bei der Benutzbarkeit, vor allem beim Undo, die am Ende dieses Abschnitts ausführlicher erläutert werden.

Beim Loslassen der Maustaste wird die Methode `handleButtonUp (int)` aufgerufen, die wiederum eine wichtige Fallunterscheidung beinhaltet, nämlich zwischen rechter und linker Maustaste. Während der Druck der linken Maustaste nur die Erzeugung eines weiteren Punkts abschließt, schließt die rechte Maustaste das Tool ab. Für die Erzeugung eines Punktes, ob durch Erzeugung eines neuen Pfads oder durch Änderung eines bestehenden, ist nach der eben beschriebenen Vorarbeit nur das Ausführen des aktuellen Commands notwendig, da dieses bereits alle relevanten Informationen bekommen hat. Das Abschließen des Werkzeugs bedeutet hingegen das Auswählen der erzeugten Figur und das Aktivieren des Default-Werkzeugs, in diesem Fall des Selection Tools.

Neben den bereits beschriebenen Unterschieden zwischen dem `CreationTool` und dem neuen `PLCreationTool` gibt es noch weitere veränderte Methoden und Funktionen. Zum einen ist die Methode `handleFocusLost()` zu nennen, die das Verhalten beim Wechsel des Fokus z.B. zu einem anderen Programm regelt. Diese arbeitet im `CreationTool` so, dass während der Erstellung eines Objekts das Programm auch in den Hintergrund wechseln kann und der Benutzer danach seine Arbeit an der gleichen Stelle wieder aufnehmen kann. Dies macht beim normalen `CreationTool` wenig Sinn, kann hier aber sehr nützlich sein, da die Erzeugung meist mehr Zeit in Anspruch nimmt. Eine weitere wichtige Neuerung ist zudem beim Aufruf des Request zu erkennen. Hier wird nicht mehr der von GEF vorgegebene Weg durch die Superklassen gewählt, sondern ein neuer durch die überschriebenen Methoden `getTargetRequest()` sowie `setTargetRequest()`. Dies ist wichtig, da hier kein normaler Request der Klasse `CreateRequest` benötigt wird, sondern der spezielle Request `PLRequest`, um die neuen, weiter oben bereits erwähnten, String-Konstanten zur Typisierung des Requests einbinden zu können.

Eine weitere Möglichkeit ein solches Multi-Selection-Tool zu implementieren ist, die Punkte zunächst nur im Werkzeug zu speichern und erst bei korrekter Beendigung des Werkzeugs die Erzeugung des gesamten Pfads zu veranlassen. Diese Lösung würde dann der Funktionsweise des `CreationTool` nahe kommen, das auch erst ganz am Ende die eigentliche Erzeugung vornimmt. Jedoch hätten

sich Nachteile bei der Undo-Funktion ergeben, da dann nur die Rücknahme des gesamten Pfads möglich gewesen wäre. So aber kann jeder einzelne Punkt für sich zurückgenommen und neu gesetzt werden, was eine erhebliche Erleichterung für den Benutzer zur Folge hat, da dieser nun nicht seine gesamte Arbeit wiederholen muss, wenn er mit einem Punkt unzufrieden ist. Zudem ist so auch bei nicht korrektem Abschluss des Werkzeugs eine korrektes Verhalten sehr wahrscheinlich, da der bisherige Pfad im Modell und im SVG-Dokument bereits gespeichert ist.

LineCreationTool
Das `LineCreationTool` ist in der Vererbungshierarchie der Erzeugungswerkzeuge unter dem `PLCreationTool` als direkte Unterklasse angeordnet. Die meisten Teile der beiden Werkzeuge sind gleich, lediglich der automatische Abschluss des `LineCreationTool` nach zwei Punkten ist eine Neuerung. Um diesem Rechnung zu tragen, ist lediglich die Methode `handleButtonUp(int)` verändert. Nach der Ausführung des Commands zur Erzeugung bzw. Änderung der Linie wird jetzt eine Fallunterscheidung vorgenommen, die prüft, ob die Linie zwei Punkte hat. Sollte dies so sein, dann wird das Werkzeug an dieser Stelle automatisch abgeschlossen, die Linie ausgewählt und das Default-Werkzeug selektiert. Damit wird der hier unnötige Schritt des manuellen Beendens übergangen.

5 Runde Pfade

Ein wichtiges Element der in der Einleitung erwähnten Animationen, ist die möglichst natürliche Bewegung der Objekte. Dafür ist es von großer Wichtigkeit, Objekte nicht nur entlang von eckigen Pfaden von Punkt zu Punkt bewegen zu können, sondern auch entlang von abgerundeten Pfaden und Kurven.

SVG bietet dafür die Möglichkeit normale Pfade mit speziellen Parametern bei der Eingabe der Punkte zu runden Pfaden zu machen. Dabei gibt es drei verschiedene Möglichkeiten für die mathematische Erzeugung der Rundung, die SVG anbietet: kubische Bézierkurven, quadratische Bézierkurven und Kurven mit Hilfe von Teilen von Ellipsen. Jeweils werden Abschnitte von 3 oder 4 Punkten benötigt, um eine Kurve zu erstellen. Für das Projekt reichen kubischen Bézierkurven als einzige Möglichkeit, da es für den Benutzer kaum von Interesse sein wird, zwischen den verschiedenen Interpolationen bzw. Approximationen zu wechseln.

5.1 Mathematischer Hintergrund

Bézierkurven sind eine approximative Möglichkeit eine Kurve an eine gegebene Folge von Punkten, Kontrollpolygon genannt, anzunähern. Dabei sind sie in parametrischer Form allgemein definiert als [Hansmann, 2010]

$$R(t) = \sum_{i=0}^{n} P_i \cdot B_{n,i}(t)$$

wobei n den Grad angibt, der von der Anzahl der Punkte abhängt, die man für die Bestimmung der Kurve heranzieht. Der Grad ist dabei immer um eins kleiner als die Anzahl der Punkte. $B_{i,n}(t)$ sind die Bernsteinpolynome des entsprechenden Grads, die für die Gewichtung der Punkte P_i sorgen. Das i-te Bernsteinpolynom n-ten Grades ist dabei in parametrischer Form definert als [Hansmann, 2010]

$$B_{n,i}(u) = \binom{n}{i} u^i (1-u)^{n-i}.$$

Da im Projekt nur kubische Bézierkurven verwendet werden, daher $n = 3$, werden nur die vier Bernsteinpolynome des Grads 3, die in Abbildung 3 grafisch dargestellte sind, benötigt.

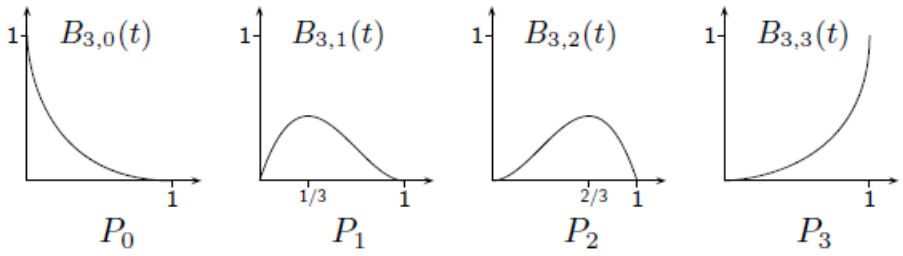

Abb. 3. Die vier Bernsteinpolynome dritten Grades mit Zuordnung zu den Stützpunkten, Quelle [Hansmann, 2010]

Anschließend kann man die beiden Definitionen zusammenführen und umformen:

$$\begin{aligned} R(t) &= \sum_{i=0}^{3} \binom{3}{i} t^i (1-t)^{3-i} P_i \\ &= (1-t)^3 P_0 + 3t(1-t)^2 P_1 + 3t^2(1-t) P_2 + t^3 P_3 \\ &= (-P_0 + 3P_1 - 3P_2 + P_3)t^3 + (3P_0 - 6P_1 + 3P_2)t^2 \\ &\quad + (-3P_0 + 3P_1)t + P_0, t \in [0,1] \end{aligned}$$

So entsteht eine Gleichung, die man nach t umstellen kann und zu dem in Abbildung 4 dargestellten, allgemeinen Ergebnis führt, wobei P_0 der Startpunkt ist, P_3 der Endpunkt und P_1 und P_2 die Kontrollpunkte.

Eine Möglichkeit zur Lösung der Gleichung bildet das Einsetzen der Punkte und eine geeignete Wahl des Parameter t. Dieser gibt an, wie der berechnete Punkt die Kurve teilt. $t = \frac{1}{3}$ wäre z.B. das Verhältnis 1 zu 2. Eine weitere Möglichkeit bildet der Algorithmus von de Casteljau, der im Projekt jedoch nicht verwendet wird. Details zu Bézierkurven, Bernsteinpolynomen und dem

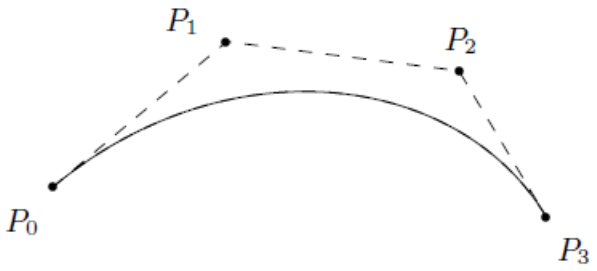

Abb. 4. Eine Bézierkurve an einem Kontrollpolygon, Quelle [Hansmann, 2010]

Algorithmus von de Casteljau sind unter anderem im Buch "Einführung in die Computergraphik" [Bungartz u. a., 2002] zu lesen.

Die nach der vorgestellten Formel berechneten kubischen Bézierkurven gelten allerdings immer nur für Abschnitte von vier Punkten. Um daraus eine Kurve zu erzeugen, die länger ist und mehr als maximal zwei Schwünge oder einen Wendepunkt hat, werden nun mehrere dieser Abschnitte konkateniert, wobei der Endpunkt des i-ten Abschnitts der Startpunkt des $i+1$-ten Abschnitts ist. Des Weiteren ist zu beachten, dass die Steigung der beiden Abschnitte in diesem Punkt identisch sein muss, damit die Abschnitte ineinander übergehen und eine Gesamtkurve ohne Sprünge entsteht. Um dies zu erreichen, ist es wichtig, dass nach der Beendigung des i-ten Abschnitts der erste Kontrollpunkt des $i+1$-ten Abschnitts automatisch gesetzt wird, als Spiegelung des zweiten Kontrollpunkts der i-ten Abschnitts am Endpunkt des i-ten Abschnitts (gleichzeitig Startpunkt des $i+1$-ten Abschnitts). Dieses wird an Abbildung 5 verdeutlicht. Durch diese Spieglung wird entsprechend die Form des nächsten Abschnitts beeinflusst.

Abb. 5. Automatische Spieglung des zweiten Kontrollpunkts des ersten Abschnitts am Endpunkt zum ersten Kontrollpunkt des zweiten Abschnitts.

Jetzt können mehrere Abschnitte miteinander verbunden werden und so beliebige Kurven geformt werden, immer unter der Einschränkung, dass eine bestimmte Anzahl von Punkten vorhanden sein muss, um vollständige kubische Bézierkurven zeichnen zu können. Diese Anzahl lässt sich leicht durch eine Glei-

chung ausdrücken:

$$\text{AnzahlPunkte} = 1 + 3 \cdot \text{AnzahlAbschnitte}$$

5.2 Speicherung im SVG

Bei der Speicherung von kubischen Bézierkurven nutzt SVG wie bei den normalen Pfaden das Attribut `d`, ähnlich dem Attribut `points` aus dem Beispiel des Grundlagenkapitels, allerdings werden bei den runden Pfaden nicht alle Punkte, Start- und Endpunkte sowie Kontrollpunkte, sondern nur ausgewählte Punkte gespeichert. Außerdem werden spezielle Kommandos gesetzt, um die einzelnen Abschnitte einzuleiten.

Die Auswahl der zu speichernden Punkte wird durch die mathematischen Gegebenheiten auf ein Minimum reduziert. Es wird lediglich der erste Abschnitt der Kurve, die ersten vier Punkte, komplett gespeichert. Dabei wird das Kommando `M` gesetzt, um den Startpunkt zu kennzeichnen und das Kommando `C`, um den Rest des ersten Abschnitts anzuzeigen. Nach dieser Deklaration der Basis wird für jeden weiteren Abschnitt lediglich der zweite Kontrollpunkt sowie der Endpunkt gespeichert und mit einem `S` eingeleitet. Ein Beispiel für einen solchen Pfad mit drei Abschnitten stellt sich dann wie folgt da:

```
<path d="M72,176 C100,143 153,143 173,171
                 S238,200 247,170
                 S299,145 322,175" />
```

Die Ausgabe ist in Abbildung 6 zu sehen.

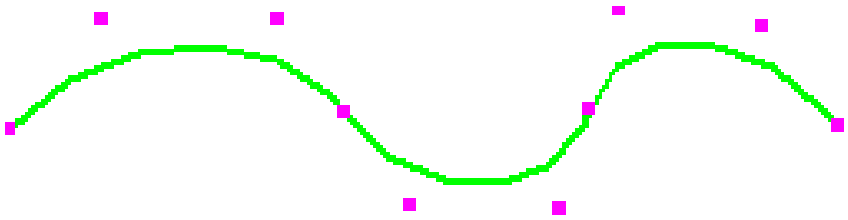

Abb. 6. Visuelles Ergebnis des Beispiels, mit Stützpunkten.

Die anderen beiden Punkte, Startpunkt und erster Kontrollpunkt, ergeben sich aus dem vorherigen Abschnitt und der eben erläuterten Mathematik zur Verbindung von mehreren Abschnitten. Aus diesen Bedingungen berechnet SVG die fehlenden Punkte selbst und schließt so nicht realisierbare Beschreibungen aus.

5.3 Umsetzung in Modell und Figur

Anders als die Umsetzung im SVG-Dokument, die die mathematischen Gegebenheiten ausnutzt, sieht die Umsetzung in den hier modellierten Java-Modellen aus. Die runden Pfade werden dabei modelliert durch die Klasse `RoundedPath`, einer Subklasse des eigentlichen Pfadmodells `Path`. Dieses Modell speichert alle Punkte, die der beim Erstellen des runden Pfads angelegt wurden, d.h. sämtliche Startpunkte, Endpunkte und Kontrollpunkte. Diese Art der Speicherung ist zwar weniger effektiv, legt aber keine Details über die Art der Rundung fest. Eine Änderung im Verfahren, z.B. das Umstellen auf quadratische Bézierkurven ist so leichter möglich. Eine weitere Konsequenz daraus ist auch, dass das Modell nicht jeden gezeichneten Pfadpunkt enthält, ähnlich wie beim normalen Pfad, der auch nur die Stützpunkte speichert, zwischen denen die Figur dann später linear interpoliert. So ist es auch bei den runden Pfaden, hier approximiert ebenfalls nur die Figur.

Im Modell sind daher keine Details zur Rundung des Pfads gespeichert, jedoch findet hier die Umsetzung auf das SVG statt und das Hinzufügen des gespiegelten ersten Kontrollpunkts in der Methode `addPointList(PointList)`. Das Spiegeln passiert automatisch bei jeder Vollendung eines Abschnitts und kann nicht vom Benutzer beeinflusst werden. Dazu wird zunächst überprüft, ob der Abschnitt vollständig ist. Danach wird gegebenenfalls eine Kopie des vorletzten Punkts um die doppelte Differenz zum letzten Punkt verschoben und als neuer letzter Punkt gespeichert. Anschließend werden alle für das SVG-Dokument relevanten Punkte vollständiger Abschnitte im SVG-Dokument gespeichert.

Des Weiteren hat das Modell die spezielle Methode `finish()`, die am Ende der Erzeugung überschüssige Punkte, den nicht vollständige letzte Abschnitt, aus dem Modell entfernt. Dies ist nötig, um den automatisch generierten ersten Kontrollpunkt wieder zu löschen.

Die eigentliche Rundung des Pfades geschieht in der Figur, die bei aktiviertem Flag `rounded` versucht den Pfad zu runden. Alle nicht vollständigen Abschnitte die während der Erzeugung des Pfads durch den Benutzer entstehen, werden linear interpoliert wie normale Pfade. Der Code zur Rundung der Pfade basiert auf dabei Code aus dem Forschungsprojekt Haystack des MIT und wurde legal aus dem Internet kopiert [Rou].

Der Code zur Rundung besteht primär aus zwei Stufen. Die erste Stufe bereitet die Liste der Punkte vor und bestimmt nacheinander die vollständigen Abschnitte, für die eine kubische Bézierkurve approximiert werden soll. Dies geschieht in der Methode `roundedOutlineShape(Graphics)` über eine Schleife, ähnlich wie im Modell bei der Speicherung in SVG. Dabei wird für jeden vollständigen Abschnitt die Methode `drawBezier(Graphics, Point, Point, Point, Point, double)` aufgerufen. Diese Methode setzt in die oben erwähnte Formel für kubisch Bézierkurven die übergebenen Punkte ein und berechnet einen Punkt auf der Kurve. Dieser wird dann mit dem Startpunkt oder seinem Vorgänger verbunden und gezeichnet. Die Werte für t, die Abstände für die Punkte, werden der Methode in Form einer Schrittweite übergeben, um die t dann von 0 ausgehend erhöht wird, bis 1 erreicht oder überschritten ist. Das

eigentliche Ausrechnen der Gleichung ist in die innere Klasse `BezierDimension` und deren Methode `getValue(double)` verlagert. So entsteht sukzessive eine Kurve aus kubischen Bézierkurven wie in Abbildung 7 zu sehen ist.

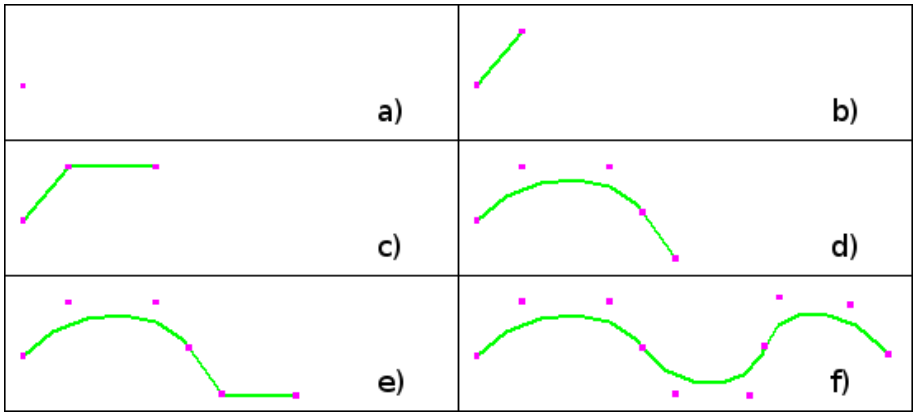

Abb. 7. Schrittweise Erstellung eines runden Pfads. **a)** Startpunkt gesetzt; **b)** erster Kontrollpunkt gesetzt, eine lineare Interpolation wird vorgenommen; **c)** zweiter Kontrollpunkt gesetzt, weitere lineare Interpolation; **d)** Endpunkt des ersten Abschnitts gesetzt, fertiger Abschnitt wird approximiert und erster Kontrollpunkt des zweiten Abschnitts wird gesetzt; **e)** zweiter Kontrollpunkt kann wieder individuell gesetzt werden; **f)** Pfad korrekt fertiggestellt.

5.4 RoundedPLCreationTool

Das `RoundedPLCreationTool` ist neben dem `LineCreationTool` eine weitere Unterklasse des `PLCreationTool` und überschreibt wiederum nur eine Methode, `handleButtonUp(int)`. Diese Methode regelt das Loslassen der Maustaste neu, wobei nur der Fall, dass die rechte Maustaste losgelassen wird, erweitert wird. In diesem Fall wird jetzt ein weiteres Command abgesetzt, ein `FinishRLLayoutCommand`, das die Methode `finish()` des erstellten Pfads aufruft und wie beschrieben überschüssige Punkte löscht. Eine nicht korrekte Beendigung des Werkzeugs hat deshalb in diesem Fall sichtbare Konsequenzen, da die überschüssigen Punkte weiter im Modell enthalten und dementsprechend auf der Canvas zu sehen sind. Im SVG-Dokument sind sie jedoch nicht zu finden, da dort nur vollständige Abschnitte gespeichert werden.

6 Evaluation

Eine detaillierte Evaluation des gesamten Programms in Bezug auf die Erstellung von kleinen, in der Einleitung erwähnten Animationen zu Roboterexperimenten,

ist in der Arbeit von Sergej Lehmann [Lehmann, 2011] zu lesen. An dieser Stelle deshalb nur eine kleine Evaluation der in dieser Arbeit beschriebenen Teile, des Multi-Selection-Tools und der runden Pfade.

Das Multi-Selection-Tool zur Erzeugung `PointList` basierter Modelle ist wie andere Komponenten grundlegend für die Erzeugung von Animationen, da ohne dieses Werkzeug das Anlegen von Pfaden, auf denen Objekte sich über die Canvas bewegen, nicht möglich ist. Selbiges gilt für die runden Pfade, ohne die nur eine Bewegung entlang von Geraden möglich wäre, was die Effekte, die bei Animationen wie von Heider und Simmel auftreten, nicht ermöglichen würde. Zudem würde durch diese wenig natürlichen Bewegungen nicht die soziale Akzeptanz von Verhalten von Robotern getestet werden können, was ein Ziel des Projekts war, da das Verhalten nicht richtig abgebildet werden kann. Die beiden hier entwickelten Teilkomponenten sind daher grundlegend zum Erreichen des Projektziels, allerdings weisen beide Komponenten noch Verbesserungs- und Erweiterungsmöglichkeiten auf.

Eine Verbesserungsmöglichkeit ist die Erhöhung der Benutzbarkeit. Beim normalen Multi-Selection-Tool zur Erzeugung von Pfaden oder Polygonen ist die Benutzbarkeit recht intuitiv und es passiert das, was auch ein ungeübter Benutzer erwarten würde. Bei der Erstellung von runden Pfaden mit unserem Werkzeug hingegen ist dies nicht unbedingt immer gegeben, besonders durch die Spiegelung und die damit verbundene automatische Erzeugung des ersten Kontrollpunkts. Dies kann bei der anfänglichen Benutzung durchaus für Verwirrung sorgen. Hier ist ein gewisses Maß an Vorwissen des Benutzers über grundlegende Strukturen und deren Mathematik vorauszusetzen. Es ist unter anderem auch nicht von außen zu erkennen, mit welcher Methode die Kurven erzeugt werden. Dass es sich um kubische Bézierkurven handelt, was wichtig ist in Bezug auf die Anzahl der Punkte, wird nicht erwähnt.

Abhilfe könnte in vielerlei Weise geschaffen werden, zum einen könnte es einen Hilfedialog geben, der bei der Auswahl des Werkzeugs für runde Pfade in einem neuen Fenster im Blickfeld des Benutzers erscheint. Dieser könnte aber auch den während der Erzeugung unwichtigen Tab des Players bzw. des Eigenschafteneditors belegen, um immer sichtbar zu sein. Zum anderen wäre auch ein kleines Tutorial sinnvoll, das der Benutzer auswählen kann, um Überraschungen zu vermeiden. Unterstützt wird aktuell allerdings schon das gefahrlose Explorieren des Werkzeugs durch eine sinnvolle Undo- und Redo-Funktionalität, die es auch erlaubt einzelne Punkte zurückzunehmen. So ist eine “Trial and Error” basierte Exploration des Werkzeugs möglich.

Des Weiteren kann man die Benutzbarkeit des Multi-Selection-Tools durch ein Werkzeug erhöhen, welches eine nachträgliche interaktive Verschiebung der Punkte zulässt. Besonders bei runden Pfaden wäre dies sehr sinnvoll, da so der Einfluss der Stützpunkte sehr gut sichtbar wird. Die aktuelle Version erlaubt zwar die Veränderung der Punkte, allerdings nur mit Hilfe eines Eigenschafteneditors in einem Tab. Dies ist jedoch wenig interaktiv und offenbart zudem noch das Problem der Zuordnung der Punkte der Figur zu denen aus der Liste des Eigenschafteneditors.

Das Ausblenden von Pfaden ist eine weitere und vor allem bei komplexeren Animationen wichtige Funktionalität, die bisher noch nicht realisiert wurde. So kann es bei größerer Zahl von Objekten oder langen Bewegungen zu starker Überlagerung der Pfade und damit der gesamten Canvas kommen.

In der internen Struktur gibt es ebenfalls Möglichkeiten zur Verbesserung, besonders in der Kategorie Refactoring gibt es noch einiges zu tun, um die Wartbarkeit und Erweiterbarkeit des Programms zu erhöhen.

Insgesamt ist festzuhalten, dass das Projektziel, ein Programm zu Erstellung von kleinen Animationen, ähnlich den in der Einleitung beschriebenen, zu entwickeln, durchaus erreicht wurde, wobei es im Bereich der Benutzbarkeit und vor allem der Hilfestellung durch kleine Zusatzfunktionalitäten noch Verbesserungsbedarf gibt.

Literaturverzeichnis

[Bat] *Batik SVG Toolkit*. http://xmlgraphics.apache.org/batik/, . – Stand: 26. Februar 2011

[Dra] *The Draw2d API Specification*. 3.3. http://help.eclipse.org/helios/index.jsp : Specification,

[GEF] *GEF (Graphical Editing Framework)*. http://www.eclipse.org/gef/, . – Stand: 26. Februar 2011

[POV] *POV-Ray*. http://www.povray.org, . – Stand: 18. März 2011

[Rou] *RoundedPolyline*. https://coloane.lip6.fr/trac/browser/trunk/application/core/src/fr/lip6/move/coloane/core/ui/figures/RoundedPolyline.java?rev=4199, . – Stand: 18. März 2011

[SMI 2008] *Synchronized Multimedia Integration Language (SMIL 3.0)*. http://www.w3.org/TR/SMIL3/ : Specification, Dezember 2008

[SVG 2010] *Scalable Vector Graphics (SVG) 1.1*. 2. http://www.w3.org/TR/SVG11/index.html : Specification, Juni 2010

[Ares 2008] Ares, Koen: *Developing an Editor for Directed Graphs with the Eclipse Graphical Editing Framework*. http://www.eclipsecon.org/2008/?page=sub/&id=102 : Vortragsfolien, 2008. – Stand: 26. Februar 2011

[Bungartz u. a. 2002] Bungartz, Hans-Joachim ; Griebel, Michael ; Zenger, Christoph: *Einführung in die Computergraphik. Grundlagen, Geometrische Modellierung, Algorithmen*. Vieweg, 2002

[Donocik 2011] Donocik, Christian: *Ein Animationswerkzeug zur Visualisierung sozialer Handlungen*. Projektbericht, 2011. – Universität Hamburg

[Freeman u. a. 2004] Freeman, Eric ; Freeman, Elisabeth ; Bates, Bert ; Sierra, Kathy: *Head First Design Patterns*. 1. O'Reilly Media, 2004

[Gee u. a. 2005] Gee, F. C. ; Browne, W. N. ; Kawamura, K.: Uncanny valley revisited. In: *ROMAN 2005. IEEE International Workshop on Robots and Human Interactive Communication*, 2005, S. 151–157

[Hansmann 2010] Hansmann, Werner: *Prüfungsunterlagen zum Modul Interactive Visual Computing (IVC)*. Skript, 2010. – Universität Hamburg

[Heider u. Simmel 1944] Heider, Fritz ; Simmel, Marianne: An Experimental Study of Apparent Behavior. In: *The American Journal of Psychology* 57 (1944), Nr. 2, S. 243–259

[Lehmann 2011] Lehmann, Sergej: *Toolevaluation: Erstellbarkeit von Filmen zu existierenden Roboterexperimenten*. Projektbericht, 2011. – Universität Hamburg

[Mammana u. a. 2007] Mammana, Jean-Charles ; Meson, Romain ; Gramain, Jonathan: *GEF (Graphical Editing Framework) Tutorial*. http://www.psykokwak.com/blog/images/gef/GEF_Tutorial.pdf : Tutorial, 2007. – Stand: 26. Februar 2011

[Pacchierotti u. a. 2005] Pacchierotti, Elena ; Christensen, Henrik I. ; Jensfelt, Patric: Human-robot embodied interaction in hallway settings: A pilot

user study. In: *Proceedings of the 2005 IEEE International Workshop on Robots and Human Interactive Communication*, 2005, S. 164–171

[Poetzsch-Heffter 2009] In: Poetzsch-Heffter, Arnd: *Konzepte objektorientierter Programmierung: Mit einer Einführung in Java (eXamen.press) (German Edition)*. 2., überarb. Aufl. Springer, 2009, S. 137

[Röhling 2011] Röhling, Sven: *GEF als Werkzeug für Editoren - Evaluation auf Basis einer Fallstudie*. Projektbericht, 2011. – Universität Hamburg

[Schäfermeier 2011] Schäfermeier, Stefan: *Animationseditor: Anforderungen & Lösungen*. Projektbericht, 2011. – Universität Hamburg

[Scholl u. Tremoulet 2000] Scholl, Brian J. ; Tremoulet, Patrice D.: Perceptual causality and animacy. In: *Trends in Cognitive Sciences* 4 (2000), Nr. 8, S. 299 – 309. – ISSN 1364–6613

[Staron 2011] Staron, Tobias: *Der Graphikeditor und GEF: die Umsetzung, das Modell und der Eigenschfteneditor*. Projektbericht, 2011. – Universität Hamburg