Der Graphikeditor und GEF: die Umsetzung, das Modell und der Eigenschafteneditor

Tobias Staron

Projekt
Animationswerkzeug zur Visualisierung
sozialen Handelns

Universität Hamburg
Wintersemester 2010/2011

Inhaltsverzeichnis

1 Motivation

Es gibt eine große Anzahl an Studien, die sich mit dem Verhalten von Menschen gegenüber Robotern befassen. In einer solchen Studie wurde eine Verhaltensweise seitens der Menschen entdeckt, welche heute als das Uncanny Valley Phänomen bekannt ist [Mori 1970][Gee u.a. 2005]. Dieses beschreibt die Beobachtung, dass Roboter bei Menschen eine umso höhere Akzeptanz haben, je menschenähnlicher sie sind. Dies gilt aber in der Regel nur bis zu einem gewissen Punkt. Sind Roboter rein äußerlich kaum noch von Menschen zu unterscheiden, verhalten sich aber nicht so wie einer, sinkt die Akzeptanz rapide. Dieses Phänomen ist zum Beispiel in der Unterhaltungsindustrie zu beachten, da hier dem Menschen nachempfundene Charaktere eine große Rolle spielen, wie beispielsweise in Filmen oder Computerspielen.

Hieran lässt sich erkennen, dass es wichtig ist, wie das Verhalten künstlicher Objekte auf Menschen wirkt, und dass sich eine Untersuchung hiervon lohnt. Bei einer anderen Studie sollten Menschen durch einen Gang gehen, in dem ihnen ein Roboter entgegenkam. Untersucht wurde dabei, wie die Probanden auf das Verhalten des Roboters reagierten. Expeimente mit solchen „echten“ Robotern können aber mitunter aufwändig und kostspielig werden.

Eine mögliche Lösung hierfür liefert eine Studie von Heider und Simmel aus dem Jahr 1944 [Heider u. Simmel 1944]. Sie haben kleine Filme erstellt, in denen nichts weiter passierte, als dass sich geometrische Figuren bewegten. Die Testpersonen sollten nur schildern, was sie sahen. Viele wollten aber nicht einfach nur Formen gesehen haben, die sich bewegten. Sie beobachten zum Beispiel, dass ein großes Dreieck ein kleineres verfolgen würde, und waren der Meinung, das größere sei aggresiv, und machten weitere Beschreibungen dieser Art.

Heider und Simmel haben damit gezeigt, dass es möglich ist, mit solchen einfachen Filmen untersuchen zu können, wie Menschen das Verhalten von künstlichen Objekten einschätzen und welche Absichten dieser sie darin erkennen. Daher wollten wir mit Unserer Anwendung, einem *Animationswerkzeug zur Visualisierung sozialen Handelns*, ein Programm zur Verfügung stellen, mit dem man Animationen im Stile der Heider-Simmel-Filme erstellen kann.

Unsere Gruppe, bestehend aus Viktor Borbus, Christian Wilms und mir, ist für die Entwicklung eines Graphikeditors zuständig gewesen. Mit diesem soll es möglich sein, verschiedene geometrische Figuren zu erstellen und diese zu verändern.

Meine Hauptaufgabe bestand darin, den Graphikeditor um neue Figuren und den zum Graphikeditor gehörenden Eigenschafteneditor um alle relevanten Eigenschaften der einzelnen Figuren zu erweitern.

2 State of the Art

Unsere Anwendung verwendet eine Reihe von bereits existierenden Techniken. Draw2D, das Graphical Editing Framework und Scalable Vector Graphics werde ich in diesem Abschnitt für ein besseres Verständis der folgenden kurz erläutern, da diese eine zentrale Rolle spielen bei der Realisierung des Graphikeditors, den unsere Gruppe entwickelt hat.

2.1 Draw2D

Für diesen Projektbericht ist von Draw2D nur von Interesse, dass hiervon eine Reihe von geometrischen Figuren zur Verfügung gestellt werden. Für die Figuren, die mit dem Graphikeditor erstellt werden können, wurde auf vier von diesen zurückgegriffen: Ellipse, Rechteck (Name der Klasse: RectangleFigure), Polygon und Polyline (s. Abb. 1). Es werden zwar mehr als nur diese vier Figuren angeboten, die anderen basieren aber auf diesen: Kreis ist ein Spezialfall der Ellipse, Quadrat des Rechtecks, Dreieck des Polygons. Linie und Pfade sind Spezialfälle der Polyline.

Wer sich näher mit Draw2D befassen möchte, für den ist sicherlich die API von Interesse [Draw2D].

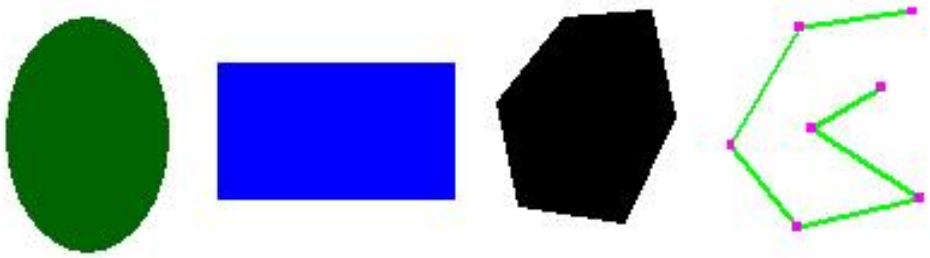

Abb. 1: Ellipse, Rechteck, Polygon, Polyline (v. l. n. r.)

2.2 GEF - Graphical Editing Framework

Bei dem Graphical Editing Framework (kurz: GEF) handelt es sich um ein Open-Source-Framework, mit dem sich Editoren erstellen lassen, die man den eigenen Vorstellungen anpassen kann. GEF verwendet eine Reihe von Design-Patterns, unter anderem das Model-View-Controller-Pattern.

Dabei beinhaltet das Modell die Klassen, von denen Exemplare erstellt werden, die dann von der View visualisiert werden. Wie das Modell konkret auszusehen hat, wird von GEF nicht vorgeschrieben. Es gibt aber einige Dinge, die zu beachten sind. So muss zum einen das Command-Pattern Berücksichtigung finden. Die Klassen müssen Operationen anbieten, um das Modell entspechend verändern zu können, wenn die/der BenutzerIn des Editors etwas verändert. Zum anderen findet das Observer-Pattern Verwendung, indem das Modell den Controller darüber informiert, wenn sich etwas an ihm verändert. Außerdem sollte eine geeignete Möglichkeit gefunden werden, das Modell abzuspeichern, damit bereits erstellte Modelle zu einem späteren Zeitpunkt wieder geladen werden können.

Die View ist die Darstellung des Modells. Für jedes Objekt einer Modellklasse gibt es ein Figure-Objekt, das das Modell-Objekt visualisiert. Für jede Modellklasse gibt es eine

Figureklasse. Bei diesen wird auf die von Draw2D bereitgestellten geometrischen Figuren zurückgegriffen. Man kann Figureklassen auch von solchen erben lassen, die man bereits für andere Modelle verwendet.

Der Controller besteht aus sogenannten EditParts. Zu jedem Objekt des Modells wird nach dessen Erstellung auch einer oder mehrere EditParts generiert. Sowohl die Modell- als auch die EditPart-Objekte werden durch sogennante Factorys erzeugt, allerdings durch verschiedene. Die EditParts erstellen dann zum jeweiligen Modell-Objekt die dazugehörige Figure. Danach wandeln sie zum einen Requests, die durch Aktionen der/des Benutzers/Benutzerins im Editor auftreten, in Commands um, die dann bearbeitet werden. Ändert die/der BenutzerIn des Programms etwas an einer Figure, verschiebt er sie beispielsweise, verändert sich durch Ausführung der entsprechenden Commands auch das Modell. Zum anderen sind die EditParts für die Aktualisierung der Figuren zuständig. Dies passiert, nachdem sich das Modell entweder auf die eben beschriebene Weise geändert hat oder es auf andere Art manipuliert worden ist. Dann werden alle zu den von den Änderungen betroffenen Modellobjekten gehörigen EditParts hierüber informiert, diese erfragen dann vom Modell die aktuellen Werte und zeichnen die betroffenen Figures entsprechend neu.

Außerdem stellt GEF autmatisch eine UNDO- und REDO-Funktion zur Verfügung.

Detailiertere Informationen zu GEF im Allgemeinen, seiner Architektur - wie beispielsweise zu weiteren verwendeten Patterns -, etc. sind im Praktikumsbericht von Sven Röhling [Röhling 2011] oder im Tutorial zu GEF [GEF 2007] zu finden.

2.3 SVG - Scalable Vector Graphics

Scalable Vector Graphics (kurz: SVG) eignet sich zur Repräsentation von graphischen Objekten und deren Animation. Auf Beispiele wird verzichtet, da es für das Verständis dieser Arbeit nicht von Nöten ist, die genaue Syntax von SVG zu kennen (für weitere Informationen über SVG s. Projektbericht von Alexej Geiger [Geiger 2011] oder SVG-Spezifikation [SVG]).

Bei SVG-Dokumenten handelt es sich um XML-Dokumente. Daher besitzen sie dieselbe Syntax und die Bearbeitung dieser durch Java-Programme beziehungsweise durch Java-Quelltexte funktioniert genauso wie bei XML-Dokumenten.

Für diesen Bericht ist vor allem von Interesse, dass SVG eine Reihe grundlegender geometrischer Figuren zur Verfügung stellt: Ellipsen, Rechtecke, Polygone, Polylines, Kreise und Linien. Die anderen Figuren, die durch den Graphikeditor erstellt werden können, werden als eine dieser von SVG angebotenen Figuren abgespeichert: Quadrate als spezielle Rechtecke und Dreiecke als Polygone. Pfade werden als gesonderte Objekte von SVG angeboten. Diese werden später für die Animation von Bewegungen von Figuren benötigt.

Für diese und weitere Animationen von Figuren greift SVG auf SMIL (Synchronized Multimedia Integration Language) zurück. Es stehen insgesamt vier verschiedene Animationen zur Verfügung: Man kann Objekte sich bewegen, sich ihre Farbe oder Form oder aber andere Eigenschaften verändern lassen.

Um sich das fertige Ergebnis anschauen zu können, gibt es verschiedene Möglichkeiten, in der Regel reicht aber ein Browser.

3 Struktur unserer Anwendung

Mit Hilfe des Animationswerkzeuges sollen letztlich SVG-Dokumente erstellen werden können, also verschiedene geometrische Objekte angelegt, ihre Eigenschaften eingestellt und ihnen die verschiedenen, von SMIL angebotenen Animationen zugewiesen werden können. Außerdem sollen diese Dokumente abgespielt werden können. Es gibt zwar noch weitere Funktionen, aber die Kernstruktur des Programmes basiert auf den genannten.

Dadurch haben sich vier Kernbereiche im Programm ergeben. Das SVG-Dokument beziehungsweise die einzelnen Objekte aus diesem werden im zentralen Szenenmodell verwaltet. Für jedes SVG-Objekt gibt es ein Java-Objekt. Wird das zentrale Szenenmodell verändert, wird auch das zugrunde liegende SVG-Dokument angepasst. Das zentrale Szenenmodell besitzt eine baumartige Struktur, genau wie auch SVG-Dokumente. Die Wurzel bildet ein Exemplar der Klasse Scene. Diese Klasse entspricht dem Wurzelknoten des SVG-Dokumentes. Die Exemplare der einzelnen Modellklassen, die jeweils zu einer von der/vom BenutzerIn erstellten geometrischen Figur gehören, werden als Kindknoten an den Wurzelknoten angehängt und die Objekte, die für die Animationen stehen, werden wiederum an die Knoten der Elemente als Kinder angehängt, die sie animieren sollen.

Mit Hilfe des Graphikeditors lassen sich geometrischen Figuren erstellen und manipulieren. Die einzelnen Animationen werden in einem seperaten Editor angelegt. So lassen sich einzelne Eigenschaften der Animationen visualisieren, wie beispielsweise die Dauer in Form von Balken unter einer Zeitleiste. Die/der BenutzerIn der Anwendung kann diese durch direkte Manipulation verändern, was die Intuitivität beim Benutzen der Software deutlich erhöht. Dies in den Graphikeditor zu integrieren wäre nur schwer möglich gewesen und hätte die Benutzeroberfläche darüber hinaus unübersichtlich gemacht. Für die Realisierung beider Editoren ist GEF verwendet worden, da es bereits viel Grundlegendes bereitstellt, was für diese benötigt wird. Wichtig ist, dass sowohl der Graphik- als auch der Animationseditor auf dasselbe Modell zugreifen, nämlich das zentrale Szenenmodell.

Außerdem gibt es einen Player, der in erster Linie für das Abspielen der erstellten SVG-Dokumente zuständig ist.

Die einzelnene Editoren und der Player interagieren nicht direkt miteinander. Alles läuft über das zentrale Szenenmodell. Alles, was die einzelnen Komponenten benötigen, holen sie sich von diesem und wenn sich am Modell etwas ändert, werden alle Komponenten, die davon betroffen sind, darüber vom zentralen Szenenmodell informiert. Dies geschieht auf Grund des Model-View-Contoller-Patterns, welches in GEF Verwendung findet (s. Abschn. 2.2).

4 Graphikeditor

In Abbildung 2 ist die Benutzeroberfläche der gesamten Anwendung zu sehen. Bei dem rot umrandete Teil handelt es sich um den Graphikeditor, den dieses Kapitel behandelt. Zuerst wird kurz auf die zur Verfügung stehenden Funktionen eingegangen, danach kommt eine detaillierte Darstellung der Klassen des Modells, die für den Graphikeditor relevant sind, und ihrer Hierarchie, dann wird gezeigt, wie der Graphikeditor relativ einfach um neue geometrische Figuren erweitert werden kann, basierend auf GEF, und worauf dabei zu achten ist, und abschließend wird es einen Einblick darin geben, wie Eigenschaften über den Eigenschafteneditor manipuliert werden und wie dies geschieht.

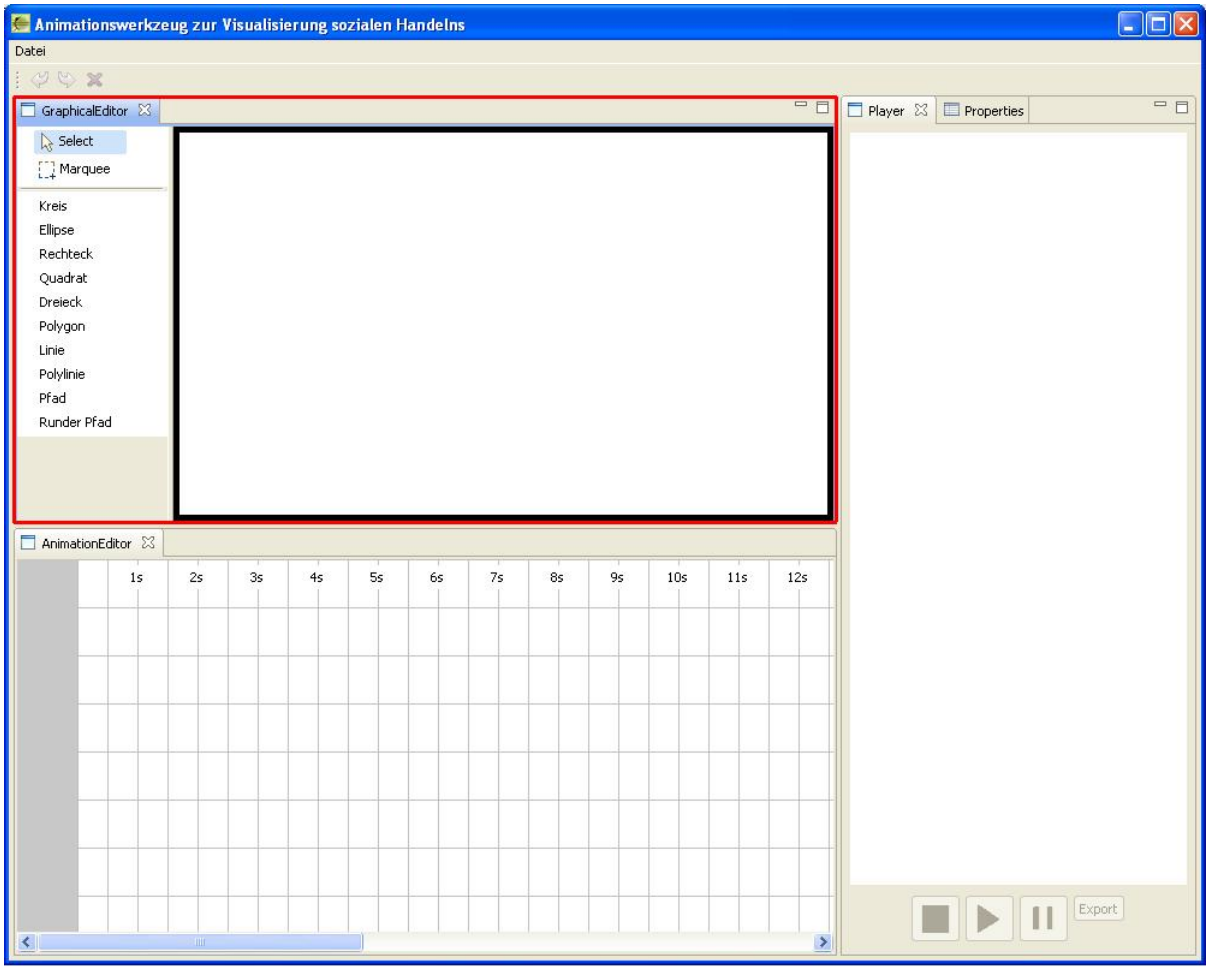

Abb. 2: Benutzeroberfläche unserer Anwendung

4.1 Funktionalität

Der Graphikeditor stellt eine Reihe von geometrischen Figuren zur Verfügung, die von der/vom BenutzerIn der Anwendung erstellt werden können: Kreise, Ellipsen, Rechtecke, Quadrate, Dreiecke, Polygone, Linien und Polylines. Des Weiteren können auch die für das Erstellen von Bewegungsanimationen notwendigen Pfade angelegt werden, sowohl in

eckiger als auch in runder Ausführung.

Zum Erstellen einer geometrischen Figur wählt man die gewünschte aus und zieht auf der Zeichenfläche ein Rechteck auf, welches die Größe der zu erstellenden Figur festlegt. Dies funktioniert aber nicht für Polygone, Polylines und die Pfade, da nicht festgelegt ist, wie viele (Eck-)Punkte sie besitzen und wie diese angeordnet sind und es daher innerhalb eines aufspannendes Rechteck eine Vielzahl von möglichen konkreten Ausprägungen der ausgewählten Figur geben kann. Für die Lösung dieses Problems hat unsere Gruppe mehrere neue Werkzeuge entwickelt, die sich für die/den BenutzerIn in der Anwendung aber alle nicht unterscheiden. Man klickt einen Punkt auf der Zeichenfläche mit der linken Maustaste an, dieser wird zum Startpunkt. Für jeden weiteren (Eck-)Punkt macht man einen erneuten Linksklick. Ist man fertig, kann man die Erstellung der Figur mittels Rechtsklick beenden. Auch eine einfache Linie wird auf diese Weise erstellt. Man könnte sie zwar auch durch das aufspannende Rechteck erstellen, aber alle Linien sollen auf dieselbe Weise angelegt werden. Daher wird für die einfache Linie dasselbe Werkzeug wie für die Polylines verwendet. Näher auf die von uns entwickelten Werkzeuge wird in dem Projektbericht von Christian Wilms [Wilms 2011] eingegangen.

Bereits erstellte Elemente können mittels des standardmäßig aktivierten Auswahlwerkzeuges ausgewählt werden. Ist ein Objekt ausgewählt, kann es verschoben oder skaliert werden. Es existiert zudem ein Tool, um mehrere Objekte auf einmal auszuwählen, die dann gemeinsam verschoben oder skaliert werden können.

Sind ein oder mehrere Objekte selektiert worden, können diese auch wieder gelöscht werden. Wenn ein einzelnes Objekt ausgewählt ist, kann man des Weitern bestimmte Eigenschaften der Figur über den Eigenschafteneditor verändern (s. Abschn. 4.4). Zusätzlich bietet unser Editor auch eine UNDO- sowie REDO-Funktion an.

4.2 Modellklassen und ihre Hierarchie

Der Graphikeditor basiert auf GEF. Das hierfür benötigte Modell ist das in Abschnitt 3 eingeführte zentrale Szenenmodell, das mit dem zugrundeliegenden SVG-Dokument zusammenhängt und es verwaltet. Wenn im Graphikeditor eine neue geometrischen Figur erstellt wird, muss auch ein neues Modellobjekt erzeugt werden. Diese Modellobjekte werden im zentralen Szenenmodell, das über eine baumartige Struktur verfügt (s. Abschn. 3), an das Exemplar der Klasse Scene, der Wurzel des zentralen Szenenmodells, als Kinder angehängt.

Für jede Figur, die man mit Hilfe des Graphikeditors erstellen kann, gibt es eine eigene Klasse, von der die Exemplare für das zentrale Szenenmodell erstellen werden. Daraus resultieren folgende Klassen: Circle, Ellipse, Rectangle, Square, Triangle, Polygon, Line, Polyline, Path und RoundedPath. Da viele dieser Klassen eine Reihe von Gemeinsamkeiten haben, sind einige von diesen Gemeinsamkeiten in gemeinsame Oberklassen ausgelagert worden. Daraus resultiert eine Typhierarchie mit mehreren Ebenen. Abbildung 3 auf der nächsten Seite zeigt diese Hierarchie.

Die oberste Klasse ist die abstakte Klasse SceneNode. Alle Objekte, die in das zentrale Szenenmodell eingefügt werden, sind von diesem Typ. Diese Klasse wiederum implementiert das Interface SceneNodeConstants, in dem eine Reihe von Konstanten deklariert werden, so dass alle Klassen des zentralen Szenenmodells auf diese zurückgreifen

können. SceneNode implementiert auch das Interface IAdaptable. Hierauf gehe ich in Abschnitt 4.4 ein. SceneNode definiert zwei abstrakte Methoden: eine zum Setzen und eine zum Abfragen des Layouts. Unter Layout wird das kleinste Rechteck, das die dazugehörige Figur beinhaltet, verstanden. Diese Methoden müssen alle Klassen des Modells anbieten, daher werden sie von SceneNode definiert. Da sich aber die Umsetzung bei den einzelnen Klassen zum Teil stark unterscheidet, sind diese Methoden abstrakt. Des Weiteren beinhaltet SceneNode Informationen über den Eltern- und die Kindknoten im zentralen Szenenmodell und Methoden, um diese zu erfragen oder zu verändern.

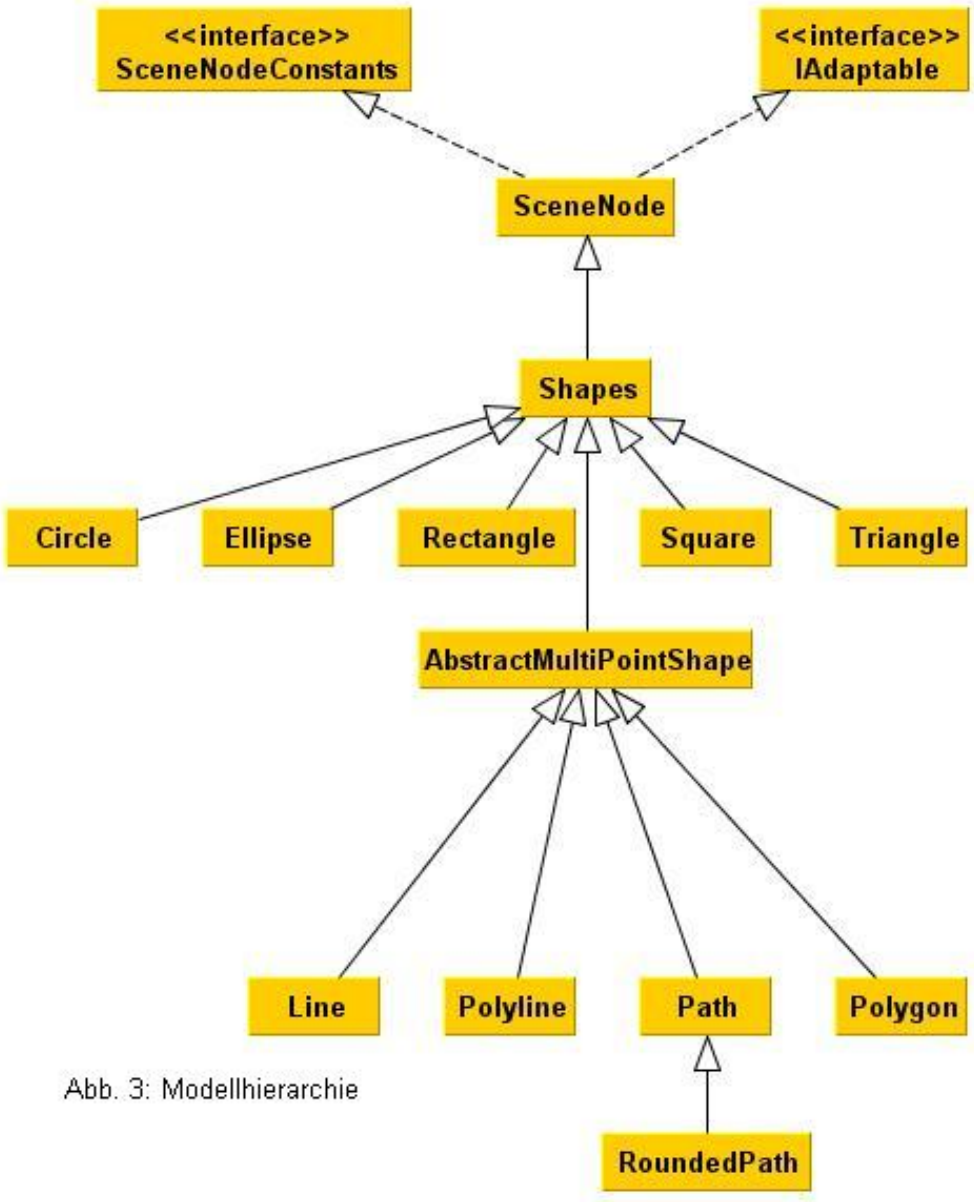

Abb. 3: Modellhierarchie

Es gibt drei Unterklassen von SceneNode. Aber nur Shapes ist von Bedeutung für den Graphikeditor. Daher sind die anderen beiden in der Modellhierarchie in Abbildung 3

weggelassen worden.

Shapes fasst eine ganze Reihe weiterer Eigenschaften zusammen, die für alle Klassen von Nöten sind: Exemplarvariablen für die Farbe einer Figur, für sowohl Farbe als auch Breite der Umrandung der Figur und dafür, ob die Figur überhaupt farblich ausgefüllt werden soll oder nicht, sowie Methoden zum Erfragen als auch zum Setzen dieser Werte.

Die Klassen für den Kreis, die Ellipse, das Rechteck, das Quadrat und das Dreieck erben direkt von Shapes, nicht aber die anderen Klassen. Für diese gibt es noch eine weitere Oberklasse, die direkt von Shapes erbt: AbstractMultiPointShape. Bei den von AbstractMultiPointShape erbenden Klassen handelt es sich um die, deren dazugehörige Figuren mit den von uns entwickelten neuen Erzeugungswerkzeugen angelegt werden, und sie unterscheiden sich grundlegend von den restlichen Klassen in einem Punkt. Es ist nicht vorgegeben, wie viele (Eck-)Punkte die zu den Klassen gehörenden Figuren haben. Es können beliebig viele sein. Diese werden jeweils in einem Exemplar der Klasse PointList gespeichert. Hierbei handelt es sich um Listen von Punkten im zweidimensionalen Raum, also von Punkten auf der Zeichenfläche im Fall des Graphikeditors. Zur Verwaltung dieser PointLists müssen alle diese Klassen eine Reihe von Methoden anbieten. Daher gibt es für diese Klassen die gemeinsame Oberklasse AbstractMultiPointShape, deren Schnittstelle diese Methoden beinhaltet, wie zum Beispiel eine Methode, um die bisherige PointList zu löschen oder um eine ganze PointList der bisherigen hinzuzufügen. Bei letztgenannter unterscheidet sich aber die konkrete Umsetzung, daher ist die Methode abstrakt und die Klasse AbstractMultiPointShape ebenfalls. Diese Methoden spielen vor allem bei der Erzeugung der Figuren mit den von uns entwickelten Werkzuegen eine große Rolle (s. Projektbericht von Christian Wilms [Wilms 2011]). Außerdem wird in dieser Klasse die Methode zum Setzen der Farbe überschrieben, da bei den Figuren, die mittels Punktlisten erstellt werden, die Farbe der Linien relevant ist und nicht mit welcher Farbe die Figuren ausgemalt sind, wie bei den übrigen.

Es gibt noch eine letzte Vererbungshierarchie zwischen den Klassen des Modells, die für den Graphikeditor von Interesse ist: Die Klasse für die runden Pfade erbt von der für die normalen Pfade. Bei den runden Pfaden gibt es zwar einige Unterschiede zu den normalen Pfaden (s. Projektbericht von Christian Wilms [Wilms 2011]), es bleibt aber auch einiges gleich. Daher erbt die Klasse der runden Pfade von der der normalen Pfaden, so dass nur die Unterschiede behandelt werden müssen.

In den Modellklassen zu den einzelnen Figuren gibt es dann für jede Eigenschaft dieser Figuren, die nicht den Standardwerten der Figuren aus Draw2D, die für die Darstellung der Figuren verwendet werden, entsprechen sollen, eine Exemplarvariable. Diese wird jeweils mit dem gewünschten Startwert, der zu Beginn für jede Instanz dieser Figur gelten soll, initialisiert. Für jede dieser Eigenschaften wird eine Methode benötigt, um sich den aktuellen Wert der Eigenschaft ausgeben zu lassen. Soll es der/dem BenutzerIn möglich sein, den Wert später zu editieren, wird zudem eine Setter-Methode benötigt (mehr dazu in Abschnitt 4.4). Bei diesen Eigenschaften handelt es sich beispielsweise um die Farbe des Kreises oder die Farbe und Breite seiner Umrandung. Wird eine dieser Eigenschaften verändert, muss daran gedacht werden, auch das entsprechende Attribut des zur Figur gehörenden SVG-Objektes zu aktualisieren. Wird die Figur skaliert oder verschoben, geschieht dies über die bereits vorgegebene Methode setLayout.

Diese Methode unterscheidet sich von Figur zu Figur, vor allem zwischen Figuren, die

mit PointLists arbeiten, und den übrigen. Beim Skalieren bei den Figuren, die auf PointLists basieren, wird die neue Breite des kleinsten, die Figur umgebenen Rechtecks durch die alte geteilt. Der hierbei ermittlete Faktor wird dann mit dem bisherigen Abstand jedes Punktes zum linken Rand des umspannenden Rechtecks multipliziert und so wird der neue Abstand berechnet beziehungsweise die neue Lage des Punktes auf der x-Achse. Die Lage auf der y-Achse wird auf vergleichbare Weise ermittelt, hierfür wird mit der alten und neuen Höhe des umspannenden Rechtecks und dem Abstand zum oberen Rand gearbeitet.

Wird eine Figur, die auf PointLists basiert, verschoben, betragen die Faktoren jeweils null. Somit wird der linke obere Eckpunkt des kleinsten, die Figur umgebenen Rechtecks genommen, sowohl für die x- als auch für die y-Achse berechnet, um wie viel sich der Punkt verglichen mit vorher verschoben hat und diese Differenz für jeden Punkt der PointList auf die Werte für beide Koordinatenachsen addiert.

Da das Skalieren und Verschieben für alle Figuren, die auf PointLists basieren gleich abläuft, findet sich diese Methode bereits in der abstrakten Klasse AbstractMultiPointShape.

4.3 Graphikeditor und GEF

GEF stellt bereits einiges zur Verfügung, was für einen Editor benötigt wird. Vor allem wird eine Struktur für die noch fehlenden Bausteine vorgegeben.

Für jede Figur, die im Editor erstellt wird, wird ein Modellobjekt benötigt. Dafür muss für jede Art von Figur, deren Erzeugung im Editor möglich sein soll, eine Klasse im Package model erstellt werden. An welcher Stelle in der dort bereits existierenden Klassenhierarchie die neu erstellte Klasse eingefügt werden muss, sollte nach den im Abschnitt 4.2 beschriebenen Kriterien geschehen. Auch die Umsetzung der relevanten Eigenschaften der zur Klasse gehörenden Figur und das Ermöglichen des Editierens dieser Eigenschaften ist dort zu finden.

Wenn eine Eigenschaft einer im Editor dargestellten Figur geändert wird, müssen die zum Modellobjekt gehörigen EditParts hierüber in Kenntnis gesetzt werden. Hierfür wird an allen Objekten, die als Listener am Modellobjekt angemeldet sind, bei denen es sich um EditParts handelt, die Methode firePropertyChange aufgerufen. Als Parameter für diese Methodenaufrufe dienen eine Konstante, die Auskunft darüber liefert, welche Eigenschaft sich verändert hat und der alte sowie der neue Wert der Eigenschaft. Der alte Wert wird benötigt, da der EditPart die Figur nur dann neuzeichnet, wenn sich der neue vom alten Wert unterscheidet.

Eine Figure wird benötigt, die die Figur auf der Zeichenfläche repräsentiert. Hierfür muss jede Art von Figur eine Klasse im Package figures angelegt werden. Diese Klassen sollten jeweils von einer Figur aus Draw2D oder von einer anderen bereits bestehenden Figure-Klasse erben. Da später die Figures von EditParts gezeichnet werden sollen, kann die Schnittstelle der Figure-Klassen um Methoden erweitern werden, die von den EditParts genutzt werden können, um mehrere Eigenschaften der Figures auf einmal zu verändern. Dafür rufen diese Methoden bereits bestehende Methoden der Figures zum Manipulieren von Eigenschaften der Figures auf. Beispielsweise sind die Figure-Klassen um die Methode setColor erweitert worden, mit der sowohl die Farbe, mit der die Figure ausgemalt ist, als auch die der Umrandung der Figure gesetzt wird. Hierfür werden die schon vorhandenen

Methoden setBackgroundColor und setForegroundColor benutzt.

Damit die/der BenutzerIn die Möglichkeit hat, Exemplare der Figur im Graphikeditor zu erzeugen, muss die Palette von diesem um diese Option erweitert werden. Dies geschieht in der Klasse GraphicalEditor, in der Methode getPaletteRoot. Die Palette wird um ein Werkzeug erweitert, mit dem Exemplare der Figur erstellt werden können. Soll beispielsweise ein aufspannendes Rechteck die Figur erzeugen, wird das CreationToolEntry benötigt. Als Parameter hierbei wird neben einem Schriftzug für die Benutzeroberfläche zu sehen sein soll und einem Tooltipp eine NodeCreationFactory übergeben.

Die NodeCreationFactory erzeugt die Modell-Objekte zu der Figur. Bei der Erzeugung einer solchen Factory wird dieser als Parameter übergeben, von welcher Modellklasse sie Objekte erzeugen soll, und in der Klasse NodeCreationFactory muss in der Methode getNewObject die Alternativen noch um die neue Modellklasse ergänzt werden. Die Erzeugung der Factorys geschieht in der Klasse GraphicalEditor und es gibt für jede Modellklassen jeweils eine NodeCreationFactory.

Der Graphikeditor hält zudem eine Referenz auf eine EditPartFactory. Diese erstellt zu jedem Modell-Objekt einen EditPart. In der Methode initializeGraphicalViewer in der Klasse GraphicalEditor werden sowohl die NodeCreationFactorys als auch die EditPartFactory registriert. Wird der Editor um eine neue Figur ergänzt, muss diese Methode angepasst werden, genauso wie die Klasse, zu der die EditPartFactory gehört. In dieser Klasse muss festgelegt werden, von welcher EditPartKlasse ein Objekt erzeugt wird, wenn es ein neues Modell-Objekt gibt. Zu jeder Klasse von Modellobjekten gibt es jeweils eine EditPartKlasse. Wird der Graphikeditor um eine neue Figur erweitert und gibt es somit eine neue Modellklasse, sollte eine neue EditPart-Klasse angelegt werden. In dieser sind vor allem zwei Methoden von Interesse: propertyChange und refreshVisuals.

Verändert sich das Modell, benachrichtigen die von den Änderungen betroffenen Modellobjekte alle zu ihnen gehörenden EditParts. Die Methode propertyChange verarbeitet die dadurch erhaltenen Informationen und kann anhand der erhaltenen Konstante, die Auskunft darüber gibt, welche Eigenschaft verändert worden ist, entscheiden, was zu tun ist. In der Regel gibt es nur zwei Optionen: die Figure soll neu gezeichnet werden oder nicht. Soll dies geschehen, wird die Methoden refreshVisuals aufgerufen. Diese zeichnet die Figure neu, indem sie alle Methoden am Figureobjekt aufruft, die für die Darstellung der Eigenschaften zuständig sind, die am Modell-Objekt geändert werden können. Damit beim Neuzeichnen mit den aktuellen Werten der Eigenschaften gearbeitet wird, holen sich die EditParts diese von ihrem Modellobjekt und übergeben sie als Parameter an das jeweilige Figureobjekt. In der Methode propertyChange muss für jede Eigenschaft des Modell-Objektes, die verändert werden kann festgelegt werden, was im Falle einer Änderung geschieht.

Wenn die/der BenutzerIn Aktionen im Editor durchführt, sieht die Struktur von GEF vor, dass die entstehenden Requests von EditPolicies in Commands umgewandelt werden, die dann abgearbeitet werden. Für jede Aktion gibt es jeweils eine Command-Klasse, in der auf alle Klassen von Modell-Objekten eingegangen wird. Für jede Art von Request gibt es jeweils eine EditPolicy-Klasse. Wird der Graphikeditor um eine neue Figur erweitert, müssen daher nur in den Command-Klassen, die zu Requests gehören, auf die bei der neuen Figur reagiert werden soll, die Alternativen um die zur neuen Figur gehörende Klasse von Modellobjekten ergänzt werden. Eine Außnahme sind die Figuren dar, die auf

PointLists basieren. Für diese gibt es für ihre Erzeugung und die Veränderung von ihren Eigenschaften Commands, die sich von denen für die anderen Figuren unterscheiden (s. Projektbericht Christian Wilms [Wilms 2011]).

Was man machen muss, damit Eigenschaften einer Figur über den Eigenschafteneditor editiert werden können, wird in Abschnitt 4.4 erläutert.

4.4 Eigenschafteneditor

GEF bietet die Möglichkeit Eigenschaften von Modellobjekten über den Eigenschafteneditor zu manipulieren. Dieser besteht aus einer zweispaltigen Tabelle. In der ersten Spalte findet man die Namen der aufgelisteten Eigenschaften. In der zweiten Spalte werden die aktuelle Werte der Eigenschaften angezeigt und dort können sie verändern werden. In diesem Abschnitt wird es weniger darum gehen, wie der Eigenschafteneditor, auf den sowohl über den Graphik- als auch den Animationseditor zugegriffen werden kann, in diese integriert wird. Dies kann in dem GEF-Tutorial, auf das sich dieser Abschnitt bezieht [GEF 2007], entnommen werden. Stattdessen behandelt dieser Abschnitt, wie der Eigenschafteneditor um weitere Eigenschaften ergänzt werden kann.

Wenn Eigenschaften von Modellobjekten über den Eigenschafteneditor editierbar gemacht werden sollen, muss dieser auf die zu den jeweiligen Modellobjekten gehörenden Modellklassen zugreifen können. Dies kann auf zwei Arten geschehen. Dieser Projektbericht wird aber nur auf die eingehen, die in unserer Anwendung Verwendung gefunden hat. Hierbei müssen alle betroffenen Modellklassen das Interface IAdaptable implementieren. Dies tut in unserem Programm abstrakte Klasse SceneNode. Da alle Modellklassen von ihr erben, können theoretisch Eigenschaften von jedem Modellobjekt über den Eigenschafteneditor geändern werden. Es gibt eine zu implementierende Operation in diesem Interface, die Operation getAdapter. Diese Operation liefert ein Objekt vom Typ IPropertySource, in unserem Programm ein Exemplar der Klasse SceneNodePropertySource. Exemplare dieser Klasse dienen als Vermittler zwischen dem eigentlichen Eigenschafteneditor und dem zu verändernden Modellobjekt. Zum einen kann der Eigenschafteneditor so auf die aktuellen Werte der Eigenschaften des Modellobjektes zugreifen, die er anzeigen soll. Zum anderen werden Änderungen an diesen Wert, die über ihn vorgenommen werden, im Modellobjekt übernommen.

Alle Eigenschaften, die über den Eigenschafteneditor verändert werden können sollen, brauchen einen eindeutigen Bezeichner, damit der Eigenschafteneditor alle eindeutig identifizieren kann. Es ist nicht festgelegt von welchem Typ diese sein müssen. Es bieten sich beispielsweise Strings als Bezeichner oder Integers als Id's an.

Für alle diese Eigenschaften müssen Methoden zum Auslesen der aktuellen Werte und zum Setzen von neuen Werten vorhanden sein. Bei den Methoden zum Setzen der Werte muss darauf geachtet werden, dass auch das SVG-Dokument angepasst wird und dass alle Listener der Modellobjekte, die zu ihnen gehörenden EditParts, darüber informiert werden, dass es Veränderungen gegeben hat.

Auch die zu den Modellklassen, die von den Anpassungen betroffen sind, gehörigen EditPart-Klassen müssen ergänzen. In diesen wird festlegt, wie mit den Änderungen der Eigenschaften umgegangen werden soll. Zudem müssen die zu den Modellklassen gehörenden Figure-Klassen angepasst werden. Dies geschieht, indem in den Figure-Klassen Methoden

zur Verfügung gestellt werden, mit denen die zu den Modellobjekten gehörenden EditParts die Eigenschaften der Figures auf die Werte der entsprechenden Eigenschaften des zugehörigen Modellobjektes setzen können.

Soll es für einige der betroffenen Eigenschaften Standardwerte gben und die aktuellen Werte dieser Eigenschaften auf diese Standardwerte zurückgesetzt werden können, so kann auch dies über den Eigenschafteneditor gemacht werden. Dafür muss es für diese Eigenschaften Defaultwerte geben, die in den Modellklassen, deren betroffen sind, deklariert werden.

Die Klasse, die das Interface IPropertySource implementiert, muss definiert werden, in unserem Programm die Klasse SceneNodePropertySource. Bei der Erzeugung von Exemplaren dieser Klasse wird dem Konstruktor das Modellobjekt übergeben, zu dem das Exemplar gehört. Dieses wird intern in einer Exemplarvariablen gespeichert, der Einfachheit wegen in einer von Typ SceneNode, da alle Modellklassen von jener Klasse erben. Im Folgenden heißt sie node.

Es gibt in dieser Klasse fünf Methoden, die von Interesse sind. In der Methode getPropertyDescriptors wird mittels if-Abfragen festgestellt, zu welcher Modellklasse node gehört. Für jede wird festgelegt, auf welche Weise die Eigenschaften, die in den Eigenschafteneditor editierbar gemacht werden sollen, verändert werden können sollen. Dies kann beispielsweise durch ein Textfeld geschehen, einen TextPropertyDescriptor, oder bei Farben mittels einer vom jeweiligen Betriebssystem abhängigen Farbpalette, einem ColorPropertyDescriptor. Ein PropertyDescriptor hingegen zeigt Eigenschaften nur an, sie verändern kann man sie mit ihm aber nicht.

Die Methode getPropertyValue holt sich die gegenwärtigen Werte der Eigenschaften vom Modellobjekt. Die Methode isPropertySet prüft, ob der gegenwärtige Wert einer Eigenschaft vom Standardwerte abweicht. Nur wenn dies der Fall ist, setzt er den Wert der Eigenschaft auf den Standardwerte zurück, wenn die/der BenutzerIn den entsprechenden Button für diese Eigenschaft drückt. Das eigentliche Zurücksetzen geschieht dann in der Methode resetPropertyValue. Das alles ist nur für Eigenschaften möglich, für die ein Defaultwert festgelegt worden ist. Das Aktualisieren der Modellobjekte nach einer Veränderung von Werten von Eigenschaften geschieht in der Methode setPropertyValue.

Diese fünf Methoden arbeiten alle nach demselben Prinzip. Sie bekommen eine ID als Parameter übergeben. Diese vergleichen sie mittels if-Abfragen mit allen Bezeichnern von Eigenschaften, die über den Eigenschafteneditor verändert werden können sollen. Für jede Eigenschft ist festgelegt, wie die jeweilge Methode ihre Aufgabe durchführt.

Für die Modellobjekte sind eine ganze Reihe von Eigenschaften über den Eigenschafteneditor editierbar gemacht worden. Die einzige, die allen Modellobjekten gemein ist, ist die Farbe. Die Modellobjekte, die nicht auf PointLists basieren, und die Polygone haben außerdem die Farbe und Breite der Umrandung gemeinsam. Dazu kommen weitere Eigenschaften, wie beispielsweise der Radius bei Kreisen, die Breite bei Quadraten oder die (Eck-)Punkte bei auf PointLists basierenden Modellobjekten. Letztgenanntes Beispiel ist ein gutes Beispiel für Eigenschaften, für die es keine Defaultwerte gibt. Bei Linien und Polylines spielt die Breite der Linien eine Rolle, anders als bei Pfaden, da diese nur den Zweck haben, dass sich an ihnen andere Figuren bei Bewegungsanimationen entlang bewegen. Dafür ist bei den Pfaden die Visibility eingebaut, mit der festgelegt werden kann, ob Pfade beim späteren Abspielen von Animationen zu sehen sein sollen.

5 Evaluation

Es ist eine Anwendung entwickelt worden, mit der es möglich ist, eine Vielzahl verschiedener Figuren zu erstellen, diese auf verschiedene Weise zu animieren und die gesamte Szene abzuspielen. Auch können die erstellten Szenen gespeichert werden. Die wesentlichen Ziele dieses Projektes sind somit erreicht worden.

Unserer Gruppe, die für die Entwicklung des Graphikeditors zuständig gewesen ist, hat fast alle ihre Ziele erreicht. Alle Figuren, die wir zur Verfügung stellen wollten, sind realisiert worden, sowie eine zusätzliche. Die runden Pfade waren anfangs nicht vorgesehen. Auch die wichtigsten Möglichkeiten zur Manipulierung der Figuren konnten umgesetzt werden. Leider ist es uns nicht gelungen eine interaktive Manipulierung der einzelnen (Eck-)Punkte von auf PointLists basierenden Figuren zu ermöglichen, da uns hierfür am Ende des Projektes die Zeit fehlte. Die stattdessen gewählte Lösung durch Manipulation der Punkte über den Eigenschafteneditor ist daher eine in unseren Augen akzeptable Lösung.

Für den Graphikeditor diente GEF als Grundlage. Es hat zwar anfangs eine Menge Zeit in Anspruch genommen, sich in GEF einzuarbeiten, aber es hat sich gelohnt, da alles, was von GEF bereitgestellt wird, für einen Editor benötigt wird. Die dadurch gewonne Zeit konnten wir für die Entwicklung einiger Besonderheiten, wie beispielsweise den Werkzeugen zur Erzeugung von Figuren mit unbestimmter Anzahl von (Eck-)Punkten investieren.

Die Verwendung von SVG als Dokumenttyp zur Speicherung war eine gute Wahl, da SVG eine Reihe von geometrischen Figuren sowie Animationen für diese anbietet. Alle Figuren, die unser Graphikeditor anbietet, bietet auch SVG an beziehungsweise man kann sie an solche anlehnen, wie beispielsweise Dreiecke. Diese gibt es nicht in SVG, deshalb werden diese von uns in SVG einfach als Polygone mit drei Eckpunkten gespeichert. Darüber hinaus lassen sich SVG-Dokumente einfach mit Browsern abspielen. Am besten ist hierfür Opera geeignet.

Schlussendlich sind wir also in der Lage simple geometrische Figuren zu erstellen, sie zu animieren und das Ergebnis abzuspielen. Es könnten mit unserem Animationswerkzeug also Filme im Stile der Heider-Simmel-Filme erstellt werden (s. Abschnitt 1). Ob es bei diesen Filmen nun auch so ist, dass deren Betrachter die geometrischen Figuren personifizieren und ihnen Handlungen und Absichten unterstellen, und ob es somit möglich wäre, zu untersuchen, welche Reaktionen das Verhalten von künstlichen Objekten bei Menschen hervorruft, ließe sich nur durch Tests mit unbefangenen, also nicht am Projekt beteiligten Personen herausfinden. Hierüber lässt sich zum gegenwärtigen Zeitpunkt somit noch keine Aussage treffen.

6 Literaturverzeichnis

[Draw2D] *The Draw2D API Specification.*
http://www-i5.informatik.rwth-aachen.de/ java/Draw2d-doc/reference/api/overview-summary.html
[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*, S. 151-177. 2005
[GEF 2007] Mammana, Jean-Charles; Meson, Romain; Gramain, Jonathan: *GEF (Graphical Editing Framework) Tutorial.* 2007
[Geiger 2011] Geiger, Alexej: *SVG-Darstellung.* Projektbericht, 2011
[Heider u. Simmel 1944] Heider, Fritz; Simmel, Marianne: *An experimental study of apparent behavior.* In: *The American Journal of Psychology*, Vol. 57, No. 2 (Apr. 1944), S. 243-259, University of Illinois Press.
[Mori 1970] Mori, Masahiro: *The Uncanny Valley.* 1970
http://www.androidscience.com/theuncannyvalley/proceedings2005/uncannyvalley.html
Übersetzung von Karl F. MacDorman und Takashi Minato
[Röhling 2011] Röhling, Sven: *GEF als Werkzeug für Editoren Evaluation auf Basis einer Fallstudie.* Projektbericht, 2011
[SVG] *Scalable Vector Graphics (SVG) 1.1 (Second Edition).*
http://www.w3.org/TR/SVG11/
[Wilms 2011] Wilms, Christian: *Ein Multi-Selection-Tool für GEF.* Projektbericht, 2011