JOHANNES KEPLER
UNIVERSITÄT LINZ | JKU

TNF

Technisch-Naturwissenschaftliche
Fakultät

Editor für die einfache Erstellung und Bearbeitung von Vektorgrafiken

MASTERARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Masterstudium

SOFTWARE ENGINEERING

Eingereicht von:
Philipp Hörtler

Angefertigt am:
Institut für Systemsoftware

Beurteilung:
a.Univ.Prof. Dr. Günther Blaschek

Linz, Mai 2013

Kurzfassung

Eine meist sehr aufwändige Tätigkeit beim Erstellen wissenschaftlicher Arbeiten ist das Anfertigen von ansprechenden Bildern und Diagrammen.
Die meisten Illustrationen in technischen Werken bestehen aus einfachen grafischen Elementen. Oftmals werden nur Rechtecke, Pfeile, Linien und Beschriftungen benötigt.
Um sich nicht unnötig lange mit der Erzeugung von Diagrammen aufzuhalten, wünschen sich Autoren einen Grafik-Editor, mit dem sie diese Aufgaben schnell und unkompliziert erledigen können.
Der Großteil der erhältlichen Vektorgrafik-Editoren besitzt einen Funktionsumfang, welcher den tatsächlich benötigten weit übertrifft. Dies hat leider zur Folge, dass einfache Illustrationen nur mit erheblichem Lern- und Arbeitsaufwand zu erstellen sind – wenn man nicht gerade professioneller Grafikdesigner ist.

Das Ziel dieser Masteraufgabe ist die Entwicklung eines neuen Vektorgrafik-Editors, dessen Funktionsumfang sich auf die wesentlichen, wirklich benötigten Elemente beschränkt.
Die Software soll das Erzeugen von Grafiken hoher Qualität erlauben und den Einarbeitungsaufwand durch intuitive Bedienkonzepte auf ein Minimum reduzieren.
Die folgende schriftliche Arbeit beschäftigt sich mit der Entstehung dieser Software und zeigt neben den Konzepten und der Architektur auch Ansätze zur Weiterentwicklung auf.

Abschließend folgen ein Vergleich mit ähnlichen Grafik-Editoren sowie Softwaremetriken, Laufzeitanalysen, eine rückblickende kritische Beurteilung und diverse Verbesserungsvorschläge.

Abstract

One of the most time consuming activities in creating a scientific paper is creating attractive pictures and diagrams.
Most illustrations in technical papers consist of very simple graphical elements. Often you only need rectangles, arrows, lines and labels.
In order not to stay unnecessarily with the creation of diagrams, an author needs a graphics editor, with which he can perform these tasks quickly and in a simple way.
Most of the available vector graphics editors have a huge set of functions, which exceeds the actually required amount by far. As a result, simple illustrations require a time consuming creation process with a high learning effort – if you're not a professional graphics designer.

The goal of this master thesis is to develop a new vector graphics editor, whose set of functions is limited to the essential and really needed elements.
The software should allow you to create high-quality graphics and reduce the learning effort with intuitive user interfaces and operation concepts to a minimum.
Besides the development process, the concepts and the architecture, the following paper also shows some approaches for extensions.

The final part contains a comparison with similar graphics editors, some software metrics, run-time analysis, a retrospective critical appraisal and various suggestions for further improvements.

Inhaltsverzeichnis

Abbildungsverzeichnis

Masterarbeitsaufgabe

Editor für die einfache Erstellung und Bearbeitung von Vektorgrafiken

Kurztitel: Vektorgrafik-Editor

Bearbeiter: Philipp Hörtler

SKZ/Matr.Nr.: 937/0557607

Institut für Systemsoftware
a.Univ.Prof. Dr. Günther Blaschek

Tel.: +43 732 2468-3434
Fax: +43 732 2468-7138
gue@jku.at

Referentin:
Birgit Kranzl / DW 7131
birgit.kranzl@jku.at

Linz, 13.9.2011

Für Illustrationen in technischen Werken benötigt man häufig einfache Diagramme mit Rechtecken, Kreisen, Linien, Pfeilen und Beschriftungen. Die meisten im Handel erhältlichen Grafik-Editoren bieten jedoch so viele Funktionen, dass solche einfachen Aufgaben nur mit erheblichem Lern- und Arbeitsaufwand zu bewältigen sind.

Das Ziel dieser Masterarbeitsaufgabe ist die Entwicklung eines Grafikeditors, der leicht zu bedienen und zu erlernen ist und dennoch die Erzeugung von Grafiken hoher Qualität erlaubt, die in Textdokumente eingebettet werden können.

Zum Mindestumfang gehören die folgenden Funktionen:

- Grafische Grundelemente: Rechtecke, abgerundete Rechtecke, Ovale, Linien, Pfeile, Polygone
- Füll- und Linienfarben, verschiedene Strichstärken (inkl. „Haarlinien“)
- Textblöcke mit variablen Schriften, Größen und Stilen
- Ausrichtung von Elementen zueinander und an einem Raster
- Export in gängige Austauschformate (z.B. über die Zwischenablage)

Darüber hinaus gehende Erweiterungen sind möglich, sie dürfen aber nicht auf Kosten der einfachen Benutzbarkeit gehen.

Für die Benutzerschnittstelle gibt es ein Vorbild; die Art der Bedienung ist daher mit dem Betreuer abzusprechen.

a.Univ.-Prof. Dr. Günther Blaschek

1. Einleitung

Um das Lesen technischer Werke angenehm und interessant zu gestalten, greifen Autoren auf Bilder zurück, welche komplexe Abläufe und abstrakte Daten verständlicher darstellen können.

Derzeit erhältliche Grafik-Software bietet dem Anwender eine Fülle von Funktionen, welche einer einfachen Bedienung häufig im Wege stehen.
Das Ziel dieses Projektes war es, eine Software zu erstellen, welche nur die benötigten Grundfunktionen zur Verfügung stellt. Durch den eingeschränkten Funktionsumfang und eine ausgeklügelte, intuitive Bedienung, werden die Erlernbarkeit und ein effizientes Arbeiten gefördert.

Bereits zu Beginn der Arbeit gab es viele grundlegende Fragen zu klären, welche das Endergebnis maßgeblich beeinflussten.
Ein zentraler Punkt war das Austauschformat. Hierbei fiel die Entscheidung auf *Scalable Vector Graphics (SVG)*, da dieses Format weit verbreitet ist und eine empfohlene W3C-Spezifikation für Vektorgrafiken darstellt. [1]

1.1. Computergrafik als Teilgebiet der Informatik

Die Computergrafik befasst sich mit der computergestützten Erzeugung und Bearbeitung von digitalen Bildern. Sie ist interdisziplinär und liegt in der Schnittmenge unterschiedlicher Fachgebiete wie Informatik, Mathematik, Physik und Mechanik.
Die möglichen Anwendungsgebiete sind vielfältig und reichen weit über Computerspiele, Spezialeffekte in Filmen und Simulationen hinaus, denn auch viele andere Bereiche wie die Steuerung von Industrieanlagen oder medizinischen Geräten bedienen sich der Computergrafik, um ihre Mensch-Maschine-Interaktion zu unterstützen und die Ergebnisse zu präsentieren [2].

1.2. Funktionsüberblick

- Grafische Elemente
  - Pfeil
  - Linie
  - Rechteck
  - abgerundetes Rechteck
  - Rechteck mit runden Seiten
  - Ellipse / Kreis
  - Polygon (erstellen und bearbeiten)
  - Text (inklusive Hintergrund und Rahmen)
- Eigenschaften Grafischer Elemente
  - Linienstärke von 1 bis 20 (inklusive Haarlinien)
  - Linienfarbe (inklusive Transparenz)
  - Füllfarbe (inklusive Transparenz)
  - Textoptionen
    - Schriftgröße
    - Schriftfarbe
    - Ausrichtung (linksbündig, zentriert, rechtsbündig)
    - Stil (Fett, Kursiv, Unterstrichen, Durchgestrichen
- Zoom-Funktion
- Magnetisches Raster
- Ausrichtung von Grafischen Elementen
- Gruppierung (erstellen/auflösen/bearbeiten)
- Verschiebung in der Z-Achse
- Verschieben von Elementen (inklusive Fixierung von X- bzw. Y-Achse)
- Duplizieren von Elementen
- Größenveränderung bzw. Verzerren von Elementen (inklusive Proportionserhaltung)
- Kopieren in die Zwischenablage
- Exportieren/Importieren von erstellten Bildern als SVG, PDF, PNG

2. Grundlagen

Dieses Kapitel befasst sich mit einigen wichtigen Grundlagen, die das Verständnis der weiteren Arbeit fördern.

2.1. Java Graphics2D

Die *Java2D*-API[1] kann zum Erstellen von qualitativ hochwertigen Grafiken verwendet werden.
Die API bietet neben diversen Zeichenfunktionen auch geometrische Transformationen, Antialiasing, Text-Layout und vieles mehr.

Im Gegensatz zu *Java2D* besitzt das AWT[2] Graphics Toolkit einige gravierende Einschränkungen:

- Linien werden mit 1-Pixel Dicke gezeichnet.
- Nur wenige Schriftarten sind verfügbar.
- Skalierungen und Rotationen sind individuell zu implementieren.
- Es bietet keine Unterstützung für Farbverläufe oder Muster.
- Der Anwender hat nur eingeschränkte Kontrolle über die Transparenz.

Die *Java2D*-API bietet einige geometrische Standardformen (Linien, Rechtecke, Ellipsen, Bögen) an, welche sehr einfach zu neuen und komplexeren Formen kombiniert werden können.
Die Linien, aus denen ein grafisches Objekt besteht, können in verschiedenen Stärken, Farben und Stilen (durchgehend, gepunktet, …) gezeichnet werden.
Es ist möglich Objekte mit Farben, Farbverläufen oder Mustern zu füllen.
Alle – mittels *Java2D* erzeugten Elemente – können verzerrt und rotiert werden. [3]

[1] Application Programming Interface.
[2] Abstract Window Toolkit.

2.2. Vektorgrafik

Vektorgrafiken sind Computergrafiken, welche aus verschiedenen grafischen Primitiven wie Linien, Kreisen, Polygonen, Kurven oder Rechtecken bestehen.
Anders als Rastergrafiken, basieren Vektorgrafiken nicht auf einem Pixelraster, sondern auf einer textuellen Beschreibung des Bildes.
Jedes – in einer Vektorgrafik enthaltene – Element bestimmt durch seinen Typ und seine Beschreibung wie es darzustellen ist. Die Summe aller – in einer Grafik enthaltenen Elemente – definiert deren geometrische Eigenschaften.

Vektorgrafiken lassen sich ohne Qualitätsverlust skalieren und besitzen hervorragende Kompressionseigenschaften.
Die Dateigrößen sind sehr gering, da hier nur geometrische Informationen festgehalten werden und nicht jeder einzelne Bildpunkt gespeichert werden muss.
Da im Gegensatz zu Rastergrafiken, die Linien- und Kurveninformationen der einzelnen Elemente nach dem Abspeichern erhalten bleiben, können diese – in geeigneten Programmen – nachträglich bearbeitet werden.

Der Hauptgrund für die Verwendung von Vektorgrafiken ist die stufenlose und verlustfreie Skalierbarkeit, welche vor allem bei qualitativ anspruchsvollen Druckmedien unverzichtbar ist.

Das Anwendungsfeld für Vektorgrafiken liegt hauptsächlich bei Firmenlogos, Diagrammen oder Icons. Da Vektorgrafiken nur aus grafischen Grundelementen bestehen, sind sie ungeeignet für typisch pixelbasierte Anwendungsfälle wie Fotografien.
Die heutigen Bildschirme arbeiten mit Rastergrafiken weshalb Vektorgrafiken immer erst gerastert werden müssen, bevor sie dargestellt werden können.
Hier liegt leider ein großer Nachteil der Vektorgrafiken, denn das Rastern großer Grafiken kann mitunter sehr rechenintensiv sein. Der Aufwand für das Rendern ist – im Gegensatz zu pixelbasierten Bildern – abhängig vom Inhalt und deshalb nicht vorab abschätzbar.

2.3. Rastergrafik

Rastergrafiken – auch Pixelgrafiken genannt – beschreiben im Gegensatz zu Vektorgrafiken nicht die Objekte im Bild, sondern speichern Informationen zu jedem einzelnen Bildpunkt.
Jede Rastergrafik besitzt eine Bildgröße, Auflösung, Farbtiefe und enthält einen Farbwert für jeden einzelnen Bildpunkt.
Das Anwendungsfeld liegt im Bereich komplexer Bilder wie zum Beispiel Fotos, welche nicht durch Vektoren (grafische Primitive) beschrieben werden können.

Im Gegensatz zu Vektorgrafiken ist der Wiedergabeaufwand bei pixelbasierten Bildformaten konstant und nicht vom Inhalt, sondern nur von ihrer Bildgröße abhängig.

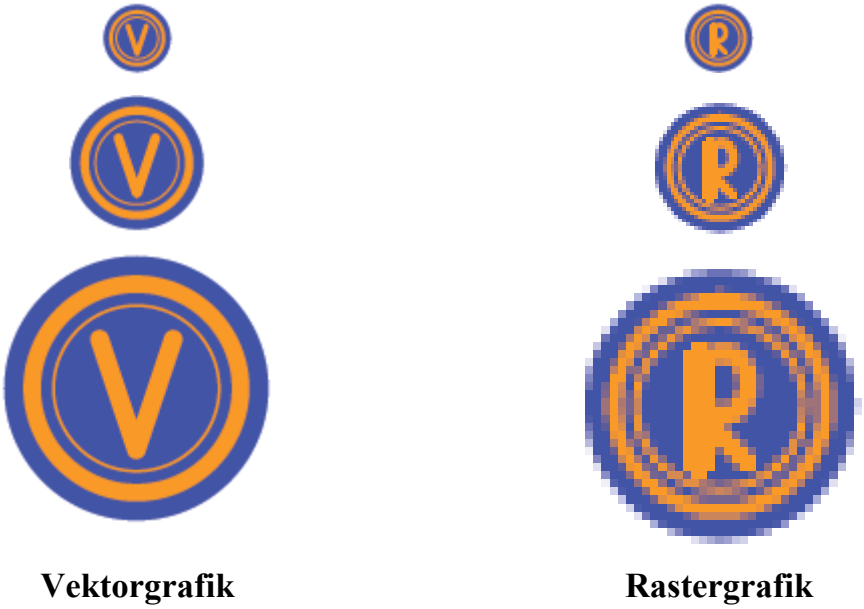


Abbildung 1: Skalierbarkeit Vektorgrafik vs. Rastergrafik

2.4. Draw

Draw ist ein Grafiktool, welches von Günther Blaschek 1987 – 1989 für Macintosh Plus und mindestens System 4.1 entwickelt wurde.

Diese Software kann auf aktuellen Computern nur mit speziellen PowerPC-Emulatoren betrieben werden. Sie bietet nur Grundfunktionalitäten und hat ihre Stärken in der einfachen und teilweise einzigartigen Bedienung. Diese Diplomarbeit wurde vergeben, um erneut ein Grafiktool zu entwickeln, welches den Anwendern ähnliche Vorzüge bietet. Draw dient als Vorlage, da sich einige der dort eingesetzten Konzepte als sehr komfortabel erwiesen haben. [4]

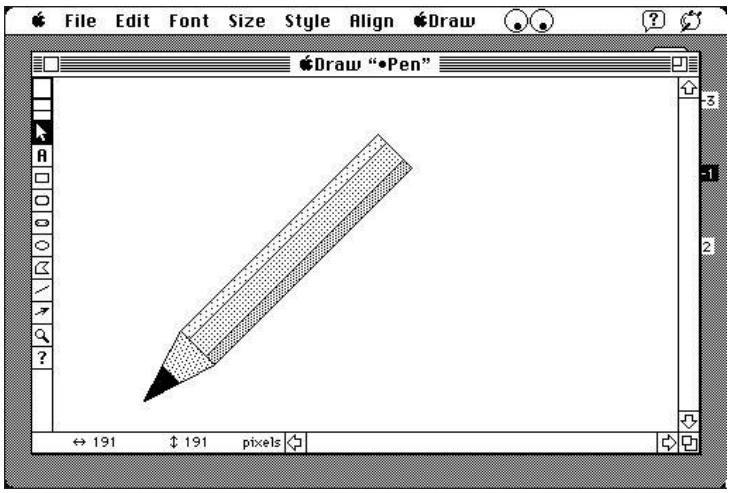


Abbildung 2: Draw - Stift

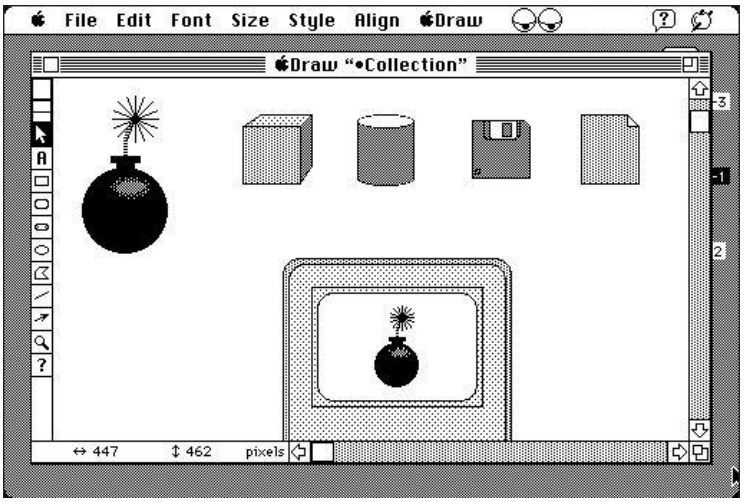


Abbildung 3: Draw - Diverses

3. Überblick / Konzept

Wie bereits im letzten Kapitel erläutert, basieren einige Besonderheiten in der Bedienung der hier entwickelten Software auf dem Grafikwerkzeug Draw von Günther Blaschek. Die entwickelten und übernommenen Konzepte werden auf den folgenden Seiten detaillierter erklärt.

3.1. Grafikelemente allgemein

Eine Grafiksoftware benötigt Objekte zur Repräsentation der grafischen Elemente in den Bildern. Diese Elemente können erstellt, bearbeitet und gelöscht werden. Darüber hinaus besitzen sie typspezifische Eigenschaften, welche die Software interpretieren muss, um die Elemente korrekt darzustellen.
Jedes Element durchläuft bei seiner Erzeugung und Weiterverarbeitung verschiedene Zustände. Je nach Zustand kann sich das Verhalten und die Darstellung ändern.

3.1.1. Zustände

Ein Element kann sechs Zustände annehmen. Diese Zustände sind wichtig um für ein Element verschiedene Bearbeitungsmodi zur Verfügung stellen zu können und zu entscheiden wie es zu rendern ist.
Zum Beispiel werden im Zustand *Edit* die Bearbeitungspunkte (Handles) zum Verzerren angezeigt, damit das Element auch verschoben werden kann.

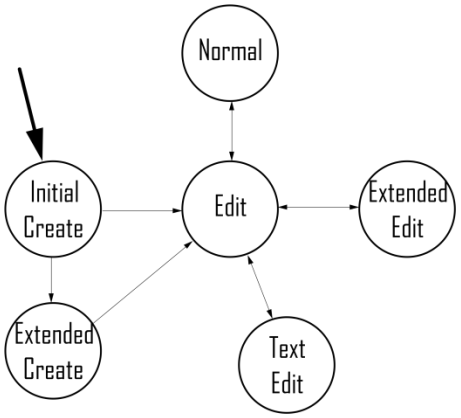


Abbildung 4: Modi der Elemente

Initial-Create Ist der Startzustand eines jeden neuen Elementes.

Extended-Create Ist ein optionaler erweiterter Erstellmodus, welcher für das Erzeugen komplexerer Elemente verwendet werden kann. (z. B. Polygon)

Edit Ist der Zustand in dem ein Element verschoben und an seinen äußeren Bearbeitungspunkten verändert werden kann.

Extended-Edit Ist ein optionaler erweiterter Editiermodus, welcher zur Implementierung von aufwändigen und komplexen Bearbeitungstechniken verwendet werden kann.

Text-Edit Ist ein optionaler Textbearbeitungsmodus um einem Element eine Beschriftung zu verleihen, oder diese zu verändern.

Normal Ist der Standardzustand eines nicht selektierten Elementes.

Das Standard Rechteck besitzt keinen *Extended-Create* oder *Extended-Edit* Zustand. Für dieses Element ist definiert, dass mit einem Doppelklick ein Zustandswechsel von *Edit* → *Text-Edit* erfolgt.
In einem Polygon hingegen ist festgelegt, dass mit einem Doppelklick ein Zustandswechsel von *Edit* → *Extended-Edit* erfolgt. Dies ist der Bearbeitungsmodus des Polygons, in dem Kanten und Ecken verschoben, sowie neue Eckpunkte eingefügt werden können.

3.2. Zeichnen

Zur Steuerung des Zeichnens wird das für Grafikprogramme übliche Interaktionsmuster verwendet.
Diesem zufolge wird zuerst das gewünschte Werkzeug (z. B. Rechteck) ausgewählt und anschließend damit auf der Leinwand gearbeitet.

Üblicherweise startet der Erstellprozess des Elementes mit dem Drücken der Maustaste. Bei dieser Aktion wird die Startposition des neuen Elementes bestimmt.

Als Nächstes kann die Maus bei gedrückter Maustaste über die Bildfläche gezogen werden.
Mittels des hierbei aufgespannten Vektors wird das zu erstellende Objekt berechnet, um dieses korrekt darstellen zu können.

Wird die Maustaste losgelassen, überprüft die Software ob sich das Element in einem gültigen und damit darstellbarem Zustand befindet und speichert dieses.

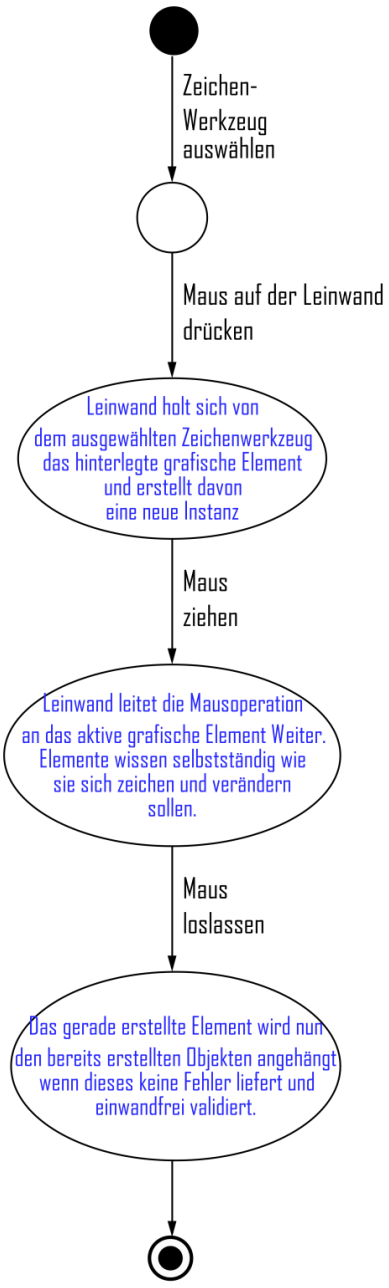


Abbildung 5: konzeptioneller Ablauf beim Zeichnen

3.3. Selektieren und Bearbeiten

Das Selektionswerkzeug wird dazu verwendet um ein oder mehrere Elemente auszuwählen oder bereits selektierte Elemente verschieben oder bearbeiten zu können.

Im ersten Schritt wird aufgrund der Klick-Position zwischen vier möglichen Fällen unterschieden:

1. selektiertes Element getroffen
2. nicht-selektiertes Element getroffen
3. Handle eines selektierten Elementes getroffen
4. nichts getroffen

3.3.1. Selektiertes Element getroffen

Wurde ein bereits selektiertes Element getroffen, so müssen drei Fälle unterschieden werden:

1. Durch das Ziehen der Maus, wird das Element über die Zeichenfläche bewegt.
2. Das Ziehen der Maus bei gedrückter Strg-Taste kopiert das Element und bewegt die Kopie – entsprechend der Mausbewegung – über die Leinwand.
3. Wird die Maus einfach losgelassen, so passiert nichts.

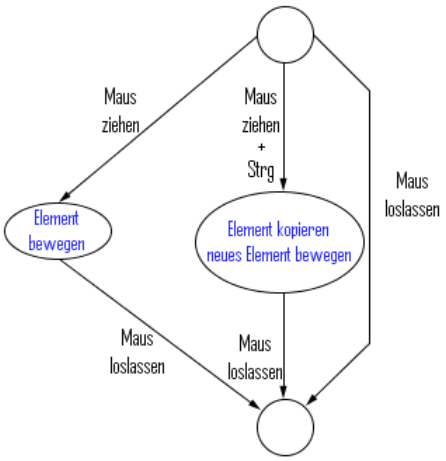


Abbildung 6: Selektiertes Element getroffen

3.3.2. Nicht-selektiertes Element getroffen

Wird ein nicht selektiertes Element getroffen, so wird dieses selektiert.

Wird die Maustaste einfach losgelassen, so bleibt es selektiert und es folgt keine weitere Aktion.

Um das neu selektierte Element über die Leinwand zu bewegen, wird die Maus bei gedrückter Taste bewegt.

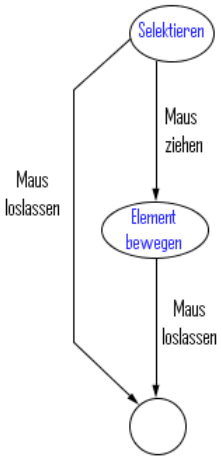


Abbildung 7: Nicht selektiertes Element getroffen

3.3.3. Handle getroffen

Um ein Handle treffen zu können, muss ein Element selektiert sein.

Wurde mit dem Selektionswerkzeug ein Bearbeitungspunkt getroffen, kann das Element verzerrt werden.

Beim Loslassen der Maustaste ohne weitere Aktion bleibt das Element unverändert und weiterhin selektiert.

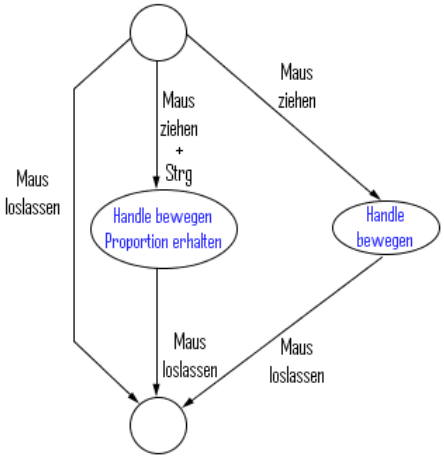


Abbildung 8: Handle getroffen

Wird die Maus bei gedrückter Taste über die Leinwand bewegt, verschiebt sich das Handle und verändert somit das Element.

Bei zusätzlich gedrückter Strg-Taste, wird auch das Handle bewegt, jedoch werden die Proportionen des Elementes beibehalten.

3.3.4. Nichts getroffen

Wird nur die leere Zeichenfläche getroffen und die Maustaste wieder losgelassen, werden alle Selektionen aufgehoben.

Um ein Auswahlrechteck aufzuspannen, wird die Maus bei gedrückter Taste über die Bildfläche bewegt um mehrere Elemente in diesem Bereich zu selektieren.

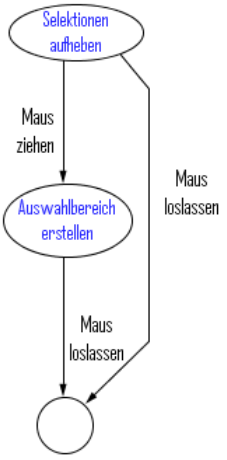


Abbildung 9: Nichts getroffen

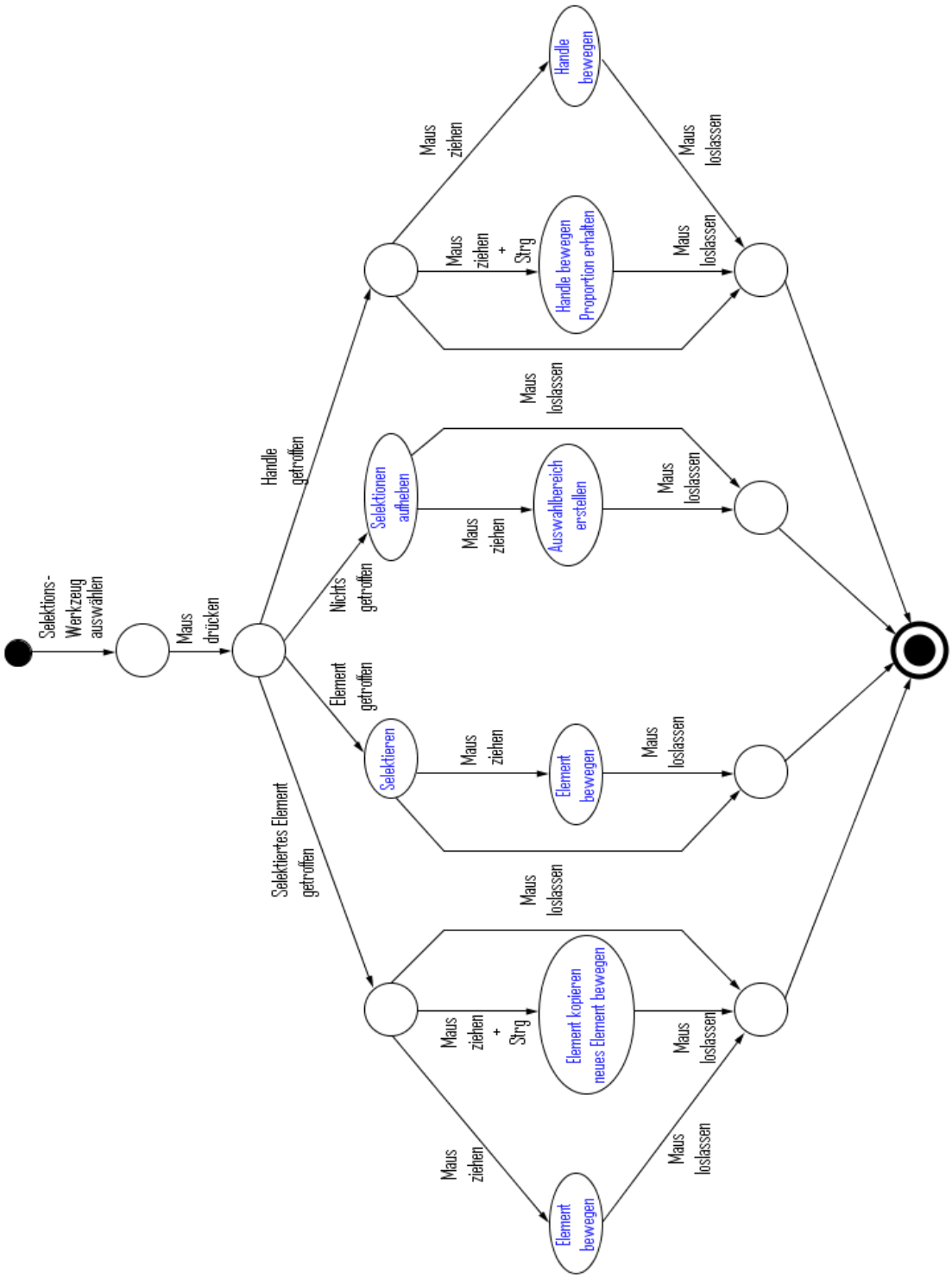


Abbildung 10: konzeptioneller Ablauf beim Selektieren und Bearbeiten

3.4. Eigenschaften ändern

Beim Einstellen der Eigenschaften – welche die Elemente annehmen können – gibt es grundsätzlich zwei Ansätze. Zum einen können die Standardeigenschaften geändert werden, welche sich auf zukünftig gezeichnete Elemente beziehen, oder es werden die Eigenschaften von bereits selektierten Elementen geändert.

Beispiele für änderbare Eigenschaften sind zum Beispiel Linienstärke, Linienfarbe und Füllfarbe.

Die Verwaltung der Standardeigenschaften übernimmt eine Klasse Namens *OptionManager*.

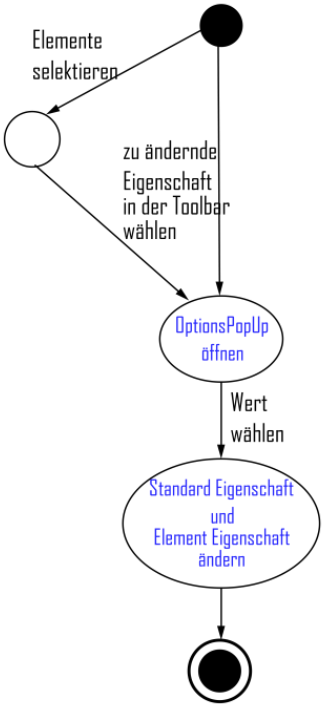


Abbildung 11: konzeptioneller Ablauf bei Eigenschaften

3.5. Aktionen ausführen

Das Ausführen von verschiedenen Aktionen, um bestehende grafische Elemente zu modifizieren, gehört zum Standardprozedere bei allen Grafikprogrammen.
Aktionen können entweder auf eine ausgewählte Untermenge oder aber auch auf alle Elemente angewandt werden.
Beispiele für Aktionen welche insbesondere für ausgewählte Elemente Verwendung finden, wären zum Beispiel das Löschen, Gruppieren, Duplizieren von Elementen oder auch das Verändern der Z-Position.
Das Kopieren oder das Ausrichten kann – falls keine Elemente ausgewählt sind – auch auf alle vorhandenen Elemente angewandt werden.

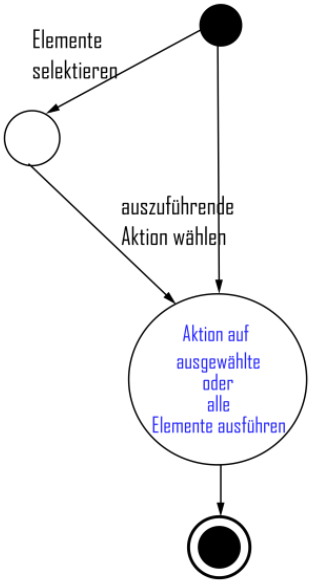


Abbildung 12: konzeptioneller Ablauf bei Aktionen

3.6. Grafikelemente im Detail

Einige grafische Elemente benötigen aufgrund ihrer Eigenheiten eine gesonderte Behandlung, was das Erstellen und Bearbeiten betrifft.

3.6.1. Polygon

Das Polygon ist eines dieser besonderen Elemente.

Polygone können offen oder geschlossen sein und im Prinzip beliebig viele Kanten aufweisen Sie können im für diese Arbeit erstellten Programm mit Farbe gefüllt, mit zusätzlichen Eckpunkten erweitert und an den Enden mit Pfeilspitzen versehen werden.

Anhand der nachfolgenden Grafiken wird ein Konzept für komfortables und intuitives Arbeiten mit Polygonen vorgestellt. Dieses verdeutlicht welche Stationen ein Polygon in seinem Lebenszyklus durchlaufen kann.

Erstellen und Bearbeiten eines Polygons

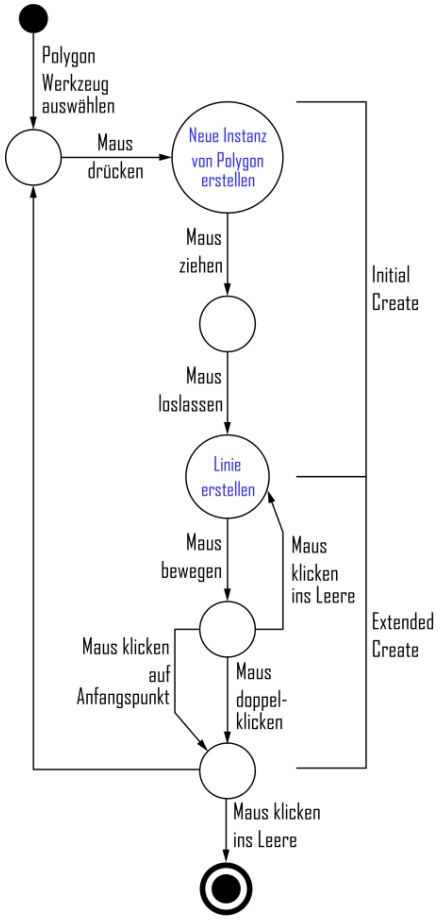


Abbildung 13: Erstellen eines Polygons

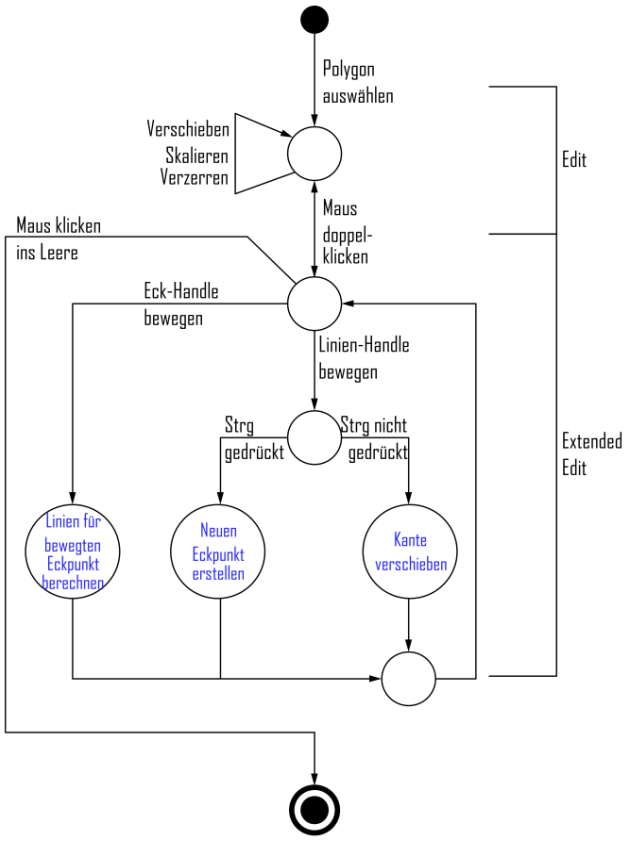


Abbildung 14: Bearbeiten eines Polygons

In Abbildung 15: Bearbeitungsmodi eines Polygons werden die möglichen Bearbeitungsschritte veranschaulicht, um das Konzept der unterschiedlichen Zustände von Polygonen besser verstehen zu können.

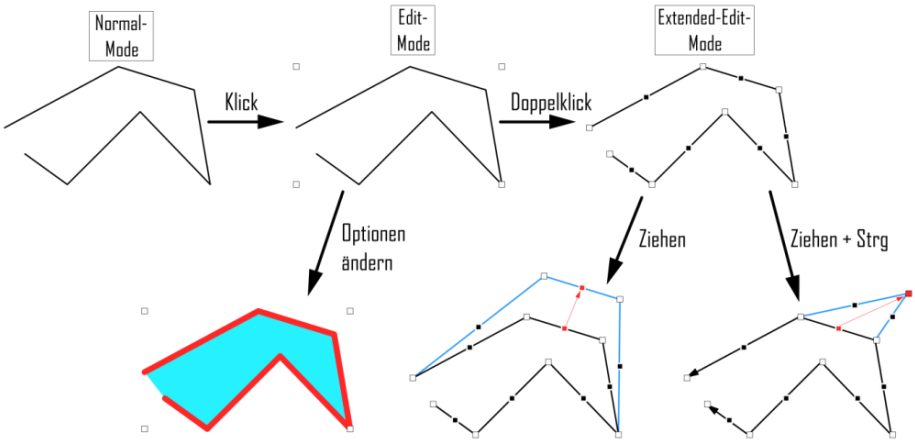


Abbildung 15: Bearbeitungsmodi eines Polygons

3.6.2. Gruppierung

Existieren mehrere Objekte welche zusammen eine Einheit bilden, ist es hilfreich diese Einzelteile zu gruppieren um damit ihre Anordnung zueinander zu fixieren. Eine ausgewählte Gruppe kann wie ein einzelnes Objekt verschoben und verzerrt werden.

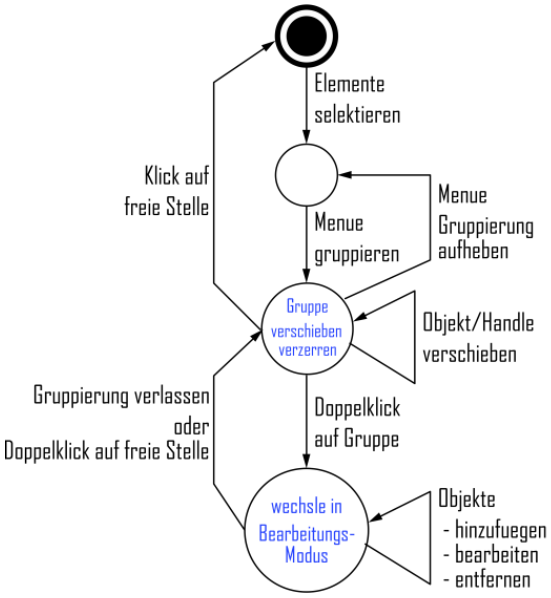


Abbildung 16: Erstellen und Bearbeiten von Gruppen

Elemente gruppieren

Bei Elemente welche in einer Gruppe verpackt sind, werden – anstatt der Bearbeitungspunkte der einzelnen Elemente – nur noch die vier äußeren Punkte eingeblendet welche die gesamte Gruppe umschließen.

Diese äußeren Handles werden verwendet um die Gruppierung zu verschieben und zu skalieren.

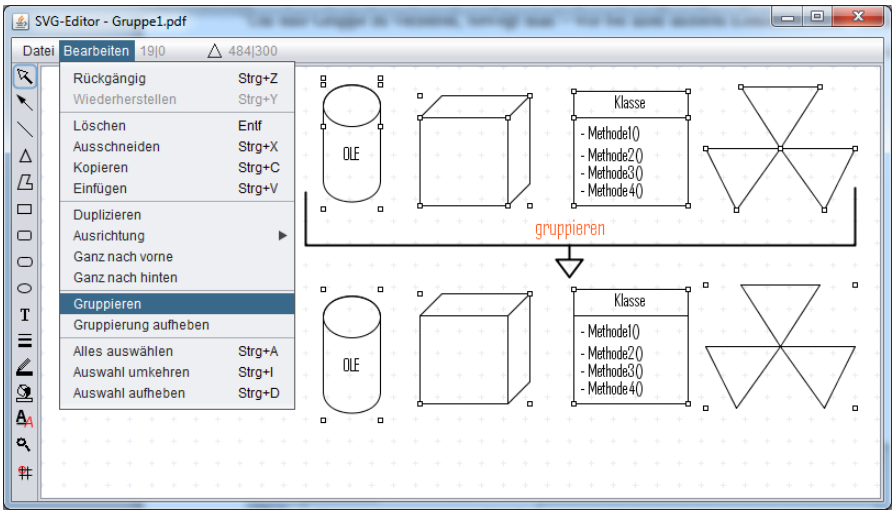


Abbildung 17: Gruppierungen

Gruppe als Ganzes bearbeiten

Durch Verschieben von Eckpunkten können Gruppierungen verzerrt werden. Alle enthaltenen Elemente verändern sich analog ihrer Gruppe.

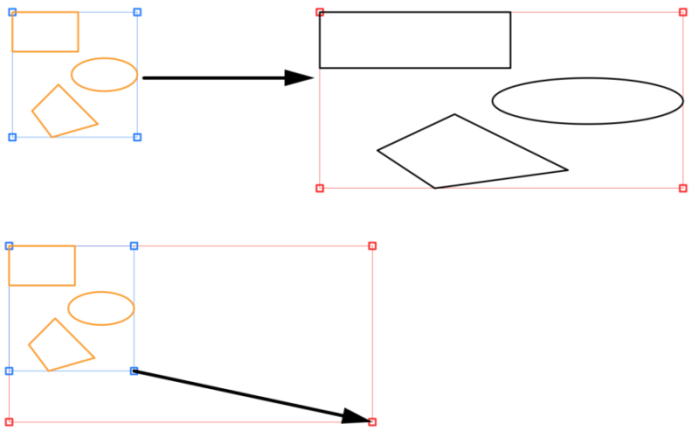


Abbildung 18: Gruppierung verschieben und verzerren

Elemente der Gruppe bearbeiten

Das Bearbeiten von Gruppierungen ist ein wichtiger Teil der effizienten Bedienung einer Grafiksoftware.

Um nicht für jeden Bearbeitungsvorgang die Gruppe auflösen und anschließend wieder erstellen zu müssen, bietet das Programm den Anwendern einen speziellen Bearbeitungsmodus an, welcher mit einem Doppelklick auf die Gruppe betreten wird.

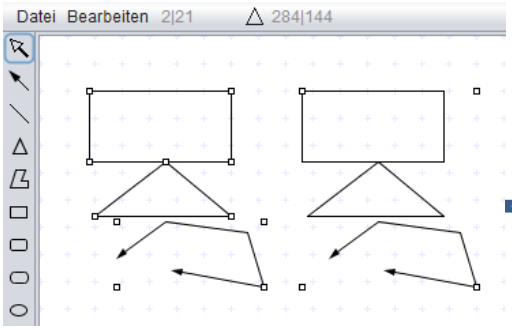


Abbildung 19: Normal und gruppiert

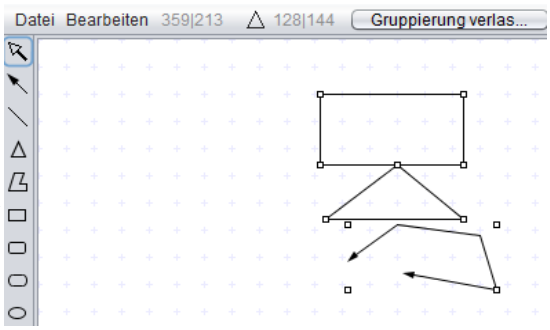


Abbildung 20: Gruppierung bearbeiten

Wie den Abbildungen zu entnehmen ist, bleiben beim Bearbeiten einer Gruppe nur deren Elemente sichtbar. Der Rest der Zeichenfläche wird ausgeblendet bis der Bearbeitungsmodus beendet wurde. Dies geschieht entweder durch den Knopf „Gruppierung verlassen“ oder durch einen Doppelklick auf eine freie Stelle der Leinwand.

Im Bearbeitungsmodus können die einzelnen Elemente verändert, gelöscht oder neue Elemente hinzugefügt werden.

Diese praktische Bearbeitungsmöglichkeit, unterstützt auch ineinander verschachtelte Gruppen, welche der Reihe nach betreten und wieder verlassen werden können.

4. Schnittstellen

Erst die Möglichkeit erstellte Bilder speichern und laden zu können macht eine Grafiksoftware sinnvoll. Um die Grafiken auch außerhalb des Bildbearbeitungsprogrammes nutzen zu können, ist es notwendig diese in gebräuchliche Formate exportieren zu können.

In diesem Kapitel werden die umgesetzten Schnittstellen zur Außenwelt vorgestellt.

4.1. Export

Das zentrale Format ist das *Scalable Vector Graphics (SVG)* Format von W3C.

Jedes grafische Element implementiert eine Methode um eine SVG-Repräsentation von sich erstellen zu können.

Durch das freie Grafikformat können auch andere SVG-fähige Editoren die exportierten Bilder darstellen.

Beim Exportvorgang – egal in welches Zielformat – werden überschüssige Ränder auf eine Stärke von 10 Pixeln reduziert.

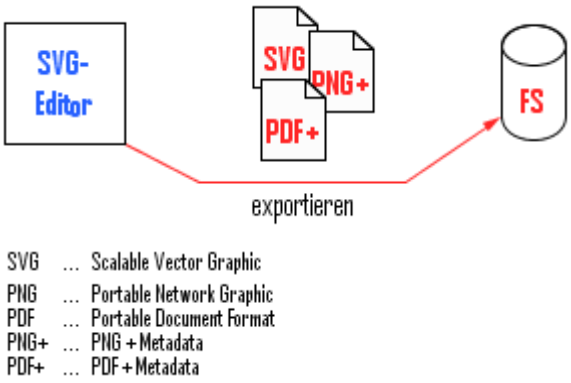


Abbildung 21: Export-Formate

4.1.1. Scalable Vector Graphics (SVG)

Da einige der Elemente Zusätze wie Pfeilspitzen oder Texte unterstützen, werden diese beim Exportieren in SVG-Gruppierungen verpackt und mit entsprechenden Kennungen versehen.

Ein valides SVG-Dokument benötigt einen gültigen Eintrag für seine Abmessungen. Für die – aus dieser Arbeit – entstandene Software, wäre eine Angabe über die Größe des Bildes keine Voraussetzung, um ein solches Dokument zu importieren. Da jedoch die meisten Vektorgrafikprogramme eine Größenangabe für eine korrekte Darstellung voraussetzen, wurde dies beim Exportieren berücksichtigt und auf korrekte Werte geachtet.

Nachfolgend wird auf die mögliche Struktur eines exportierten SVG-Dokumentes und dessen Elemente näher eingegangen.

```
<svg width="303" height="208" xmlns="http://www.w3.org/2000/svg">
  <!-- Created with SVGEditor by Philipp Hoertler -->
    <g id="SVGLine"> ..... </g>
    <g id="SVGPolygone"> ..... </g>
    <g id="SVGRectangle"> ..... </g>
    <g id="SVGRoundSideRectangle"> .... </g>
    <g id="SVGEllipse"> ..... </g>
    <g id="SVGText"> ..... </g>
</svg>
```

Die Elemente *SVGLine* und *SVGRectangle* implementieren mehrere grafische Repräsentationen.

Eine *SVGLine* kann mit 1-2 Pfeilspitzen versehen werden und mutiert quasi zum Pfeil und ein *SVGRectangle* kann mit abgerundeten Kanten ausgestattet werden.

SVGRectangle (+ abgerundetes Rechteck):

```
<g id="SVGRectangle">
   <rect x="60" y="136" height="36" width="100" stroke="rgb(0,0,0)"
       fill="rgb(255,154,38)" stroke-width="0.25"
   />
</g>
```

Rechteck

```
<g id="SVGRectangle">
   <rect x="144" y="12" height="36" width="72" stroke="rgb(36,36,36)
       fill="rgb(212,255,38)" stroke-width="8.0" rx="20" ry="20"
   />
</g>
```

Rechteck abgerundet

SVGPolygone:

```
<g id="SVGPolygone">
   <path d="M32,80L116,44L164,64L148,120L68,120L60,92L44,100"
       stroke="rgb(0,0,0)" fill="none" stroke-width="2.0"
   />
   <path d="M32,80L50,77L46,68L32,80" stroke="rgb(0,0,0)"
       fill="rgb(0,0,0)" stroke-width="0.25"
   />
</g>
```

Polygon

Pfeilspitze

SVGLine (Linie + Pfeil):

```
<g id="SVGLine">
    <text x="39" y="34" fill="rgb(0,0,0)" font-family="Agency FB"
        font-size="16" text-anchor="start">
        <tspan x="216" dy="16" >Test</tspan>
        <tspan x="216" dy="16" >Text</tspan>
    </text>
    <line x1="36" y1="60" x2="91" y2="39" stroke="rgb(0,0,0)"
        stroke-width="1.0"
    />
    <path d="M28,64L37,63L35,58L28,64" stroke="rgb(0,0,0)"
        fill="rgb(0,0,0)" stroke-width="0.25"
    />
    <path d="M100,36L90,36L92,41L100,36" stroke="rgb(0,0,0)"
        fill="rgb(0,0,0)" stroke-width="0.25"
    />
</g>
```

Text

Linie

Pfeilspitzen

SVGRoundSidedRectangle:

```
<g id="SVGRoundSideRectangle">

  <path stroke="rgb(36,36,36)" fill="rgb(212,255,38)"
    stroke-width="2.0"
    d="M204,84
       C184,84 184,124 204,124
       L252,124
       C272,124 272,84 252,84
       L204,84Z"
  />

</g>
```

SVGEllipse:

```
<g id="SVGEllipse">

  <ellipse cx="218" cy="164" rx="38" ry="20" stroke="rgb(36,36,36)"
    fill="rgb(255,38,212)" stroke-width="2.0"
  />

</g>
```

SVGText:

```
<g id="SVGText">

  <text x="233" y="41" fill="rgb(0,0,0)" font-family="Agency FB"
    font-size="16" text-anchor="start">
    <tspan x="233" dy="16" >TestText</tspan>
    <tspan x="233" dy="16" >:-)</tspan>
  </text>

</g>
```

4.1.2. Portable Network Graphics (PNG)

Damit die Kompatibilität zu allen gängigen Textverarbeitungsprogrammen gewährleistet ist, bietet die Software die Möglichkeit, die erstellten Grafiken in ein Rasterformat zu exportieren.

Des Weiteren werden PNG-Grafiken beim Exportieren mit einer SVG-Beschreibung als Metadaten gespeichert, um die Möglichkeit einer nachträglichen Bearbeitung zu gewährleisten.

Java beherrscht standardmäßig bereits das Erstellen von Rastergrafiken im PNG-Format. Für die korrekte Darstellung von Haarlinien und eine entsprechend hohe Bildqualität, wird die Rastergrafik in 8-facher Vergrößerung erzeugt.

Da das PNG-Format neben der verlustlosen Komprimierung auch Transparenz anbietet, weisen die exportierten Pixelbilder – wie die Vektorgrafiken – keinen Hintergrund, sondern lediglich die gezeichneten Elemente auf. [5]

4.1.3. Portable Document Format (PDF)

Die Export-Funktion für PDF-Dokumente wurde erst in der späteren Entwicklungsphase ergänzt.

PDF wurde als zusätzliches Austauschformat gewählt, da Microsoft Word keine SVG-Grafiken unterstützt.

Im Gegensatz zu PNG, werden in PDF Vektorgrafiken erstellt. Das PDF Format bietet die Möglichkeit, Dokumente mit Metadaten zu versehen.

Mit Hilfe dieser Metadaten ist es möglich, die von der Software erstellten PDF-Dokumente wieder zu importieren.

4.2. Import

Die drei Exportformate wurden bereits erläutert und es wurde gezeigt, dass das XML basierte Grafikformat SVG eine wichtige Rolle spielt.
Dies wird noch offensichtlicher wenn die – eigens implementierte – Import-Klasse betrachtet wird. Diese Klasse ist für das Parsen und Erzeugen der grafischen Elemente beim Öffnen von Bildern zuständig und arbeitet ausschließlich mit den beim Speichern erzeugten XML-Elementen.

Da erzeugte Bilder welche im PNG- oder PDF-Format abgelegt wurden, Metadaten mit deren SVG-Repräsentation enthalten, können diese wieder geladen und weiterverarbeitet werden.
Wird eine PNG-/PDF-Grafik geöffnet, werden die jeweiligen Metadaten ausgelesen und an die Import-Klasse übergeben, welche die einzelnen XML-Elemente lädt, die Daten interpretiert und daraus Java-Objekte erstellt.

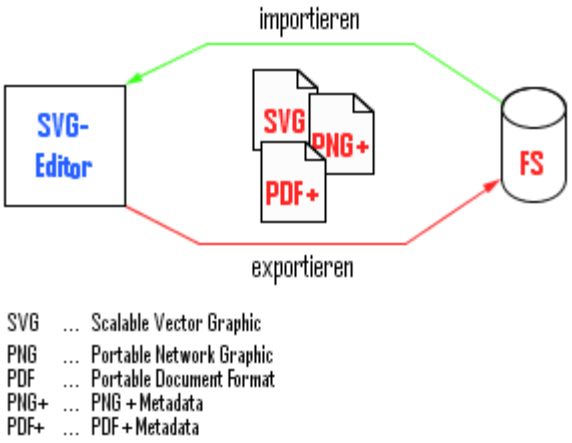


Abbildung 22: Import-/Export-Formate

4.3. Zwischenablage

Ein gängiger Weg um Daten zu duplizieren oder zwischen Programmen auszutauschen ist die Zwischenablage.
Um die gewünschte intuitive Bedienung weiter zu fördern, soll das Programm diesen vertrauten Mechanismus unterstützen.

Die aktuelle Implementierung der Zwischenablage ermöglicht den Datenaustausch in drei Formaten:

- String
- Bild
- Liste von Dateien

Von diesen werden allerdings vorerst nur String und Dateien verwendet.

Beim Kopieren von – vorab selektierten – Objekten werden an das Systemclipboard ein String mit der SVG-Beschreibung und eine PNG-Datei inkl. SVG-Metadaten übergeben.
Da das PNG-Bild als Datei vorliegt, kann dieses unter Windows auch direkt im Explorer eingefügt werden. Dies klappt natürlich auch auf dem umgekehrten Weg, sofern es sich um eine Bilddatei handelt, welche die benötigten Metadaten enthält.
Kopieren&Einfügen aus dem Explorer in das Grafikprogramm funktioniert auch mit zuvor erstellten PDFs, welche die entsprechenden Voraussetzungen erfüllen.

4.4. Anbindung an die Textverarbeitung

Das Ziel dieser Arbeit war die Entwicklung eine Grafiksoftware, welche darauf ausgelegt ist, hochwertige Bilder für technische Schriftstücke zu erstellen. Daher ist die Anbindung an gängige Textverarbeitungsprogramme einer der wichtigsten Punkte.

Während der Entwicklung stellte sich heraus, dass die meisten Textverarbeitungsprogramme keine Unterstützung für SVG-Grafiken bieten.
Microsoft Word kann nur Vektorgrafiken in Formaten wie WMF, EMF[3] und PDF[4] importieren und darstellen. Da sich das PDF-Format durch die hohe Verbreitung anbot, schien es die perfekte Ergänzung zu SVG zu sein.
Nach weiterer Recherche und einigen Versuchen stellte sich jedoch heraus, dass MS Word diese PDF-Grafiken nur mit schlechter Qualität zu Papier bringen kann und dass beim Herauskopieren aus einem Dokument kein PDF in der Zwischenablage landet welches über die Java Zwischenablage weiterverarbeitet werden könnte.
Die Zwischenablage bietet verschiedene Rastergrafikformate, OLE[5]-Objekte sowie EMF und WMF.
Es fanden sich auch zwei RAW-Formate in der Zwischenablage, welche den gewünschten SVG-Code enthielten. Leider war es nicht möglich über die Java-System-Zwischenablage Zugriff auf diese Daten zu bekommen.
Es stellte sich heraus, dass Java nur einen Teil der tatsächlichen Zwischenablage auslesen kann.

MS Word ermöglicht es aber eingefügte Grafiken wieder abzuspeichern. Ein PDF kann demnach auch als solches wieder exportiert werden.
Bei einem solchen Export von Bildern im PDF-Format bleiben sämtliche Metadaten erhalten, wodurch diese mit der Grafiksoftware wieder importiert und bearbeitet werden können.

---

[3] Windows Enhanced Metafile
[4] Windows Metafile
[5] Object Linking and Embedding

MS Word kann auch PNG-Grafiken aufnehmen und diese erneut abspeichern. Leider sind nach dem Exportieren keine Metadaten mehr vorhanden was ein nachträgliches Bearbeiten der einzelnen Elemente unmöglich macht.

Bei einem Versuch mit *OpenOffice* wurde die Erkenntnis gewonnen, dass dieses nicht einmal PDF-Bilder unterstützt.
Auch eingefügte PNG-Grafiken können wieder exportiert werden. Jedoch verhält sich der Exportmechanismus genau wie bei MS Word und verwirft sämtliche Metadaten.
Die nachfolgende Abbildung soll die Interaktionsmöglichkeiten verdeutlichen.

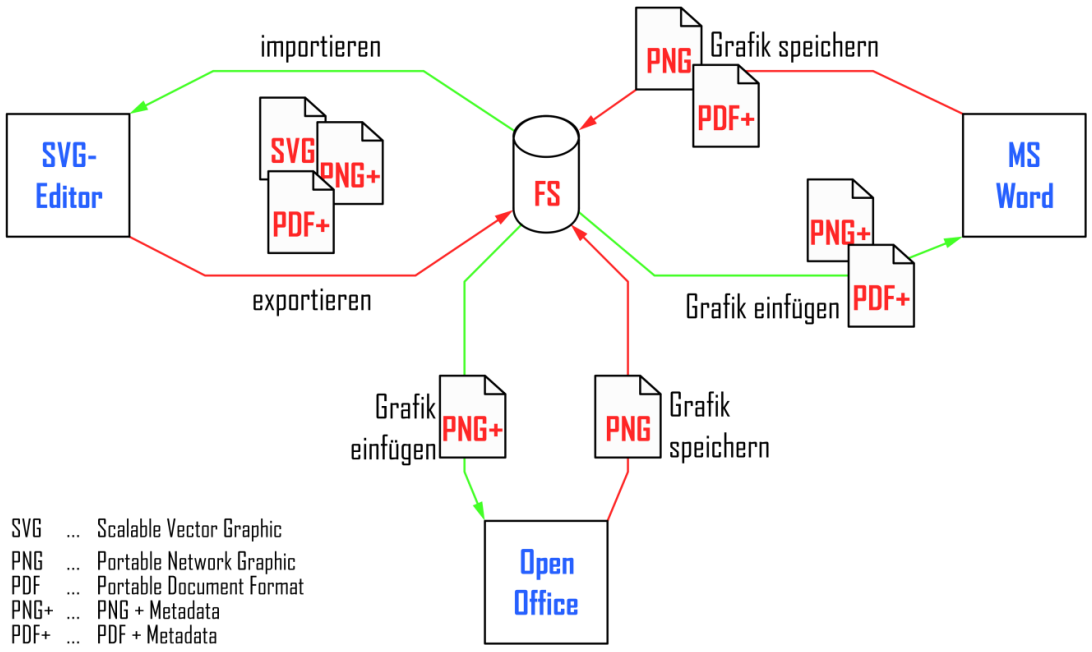


Abbildung 23: Anbindung an die Textverarbeitung

5. Ergebnisse

Bevor nun ein paar Beispielbilder gezeigt werden, sollte noch erwähnt werden, dass alle Grafiken in dieser Arbeit – außer Bildschirmschnappschüssen – mit der behandelten Software erstellt wurden.

Da im Laufe dieser Arbeit viel Anschauungsmaterial entstanden ist, werden an dieser Stelle nur noch ein paar ausgewählte Dinge vorgestellt und erklärt.

Häufig benutzte Objekte

Objekte die sehr häufig benötigt werden können als Grafiken abgespeichert und bei Bedarf immer wieder eingefügt werden. Dies führt zu effizienterem Arbeiten und ermöglicht das Designen von komplexen, zusammengesetzten Objekten nach den eigenen Vorstellungen.

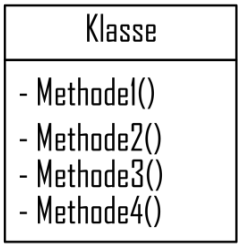


Abbildung 24: Klasse

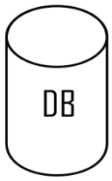


Abbildung 25: Physischer Speicher

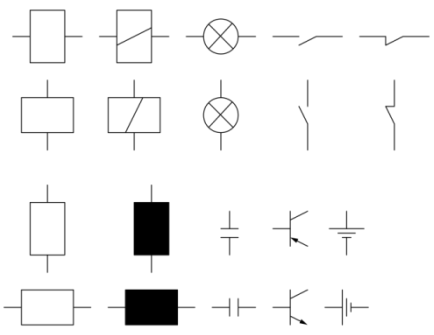


Abbildung 26: Elektronik

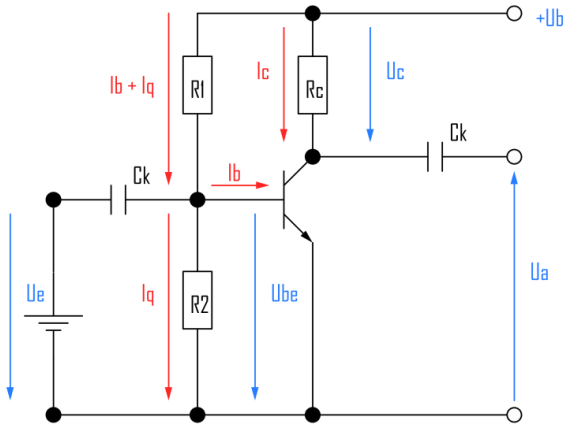


Abbildung 27: Strom-/Spannungsteiler

Producer-Consumer Ablauf

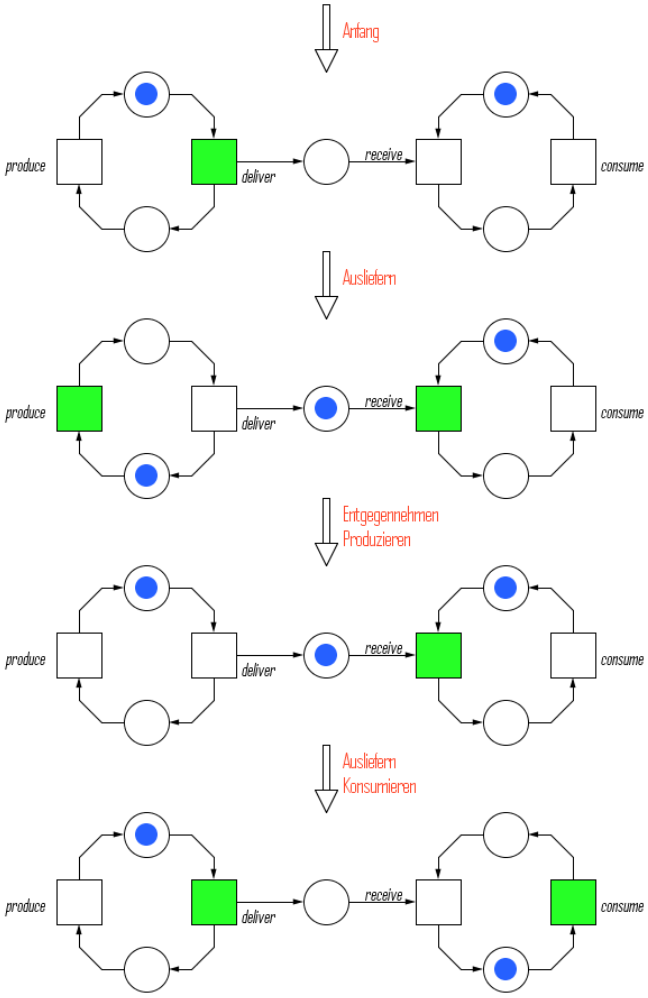


Abbildung 28: Producer-Consumer

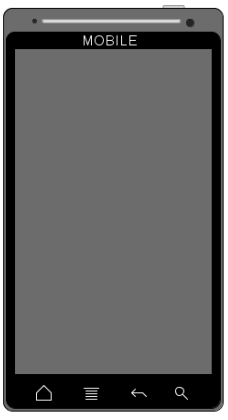


Abbildung 29: Mobiltelefon

Abbildung 30: Stift

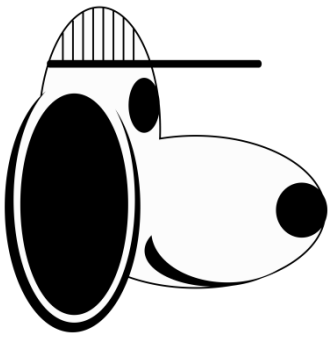


Abbildung 31: Snoopy

6. Programmstruktur

Der Editor wurde in der objektorientierten Sprache *Java* erstellt und besteht aus verschiedenen kooperierenden Komponenten.
In diesem Kapitel werden die strukturellen Bestandteile näher betrachtet und ihr Aufbau und Zusammenspiel verdeutlicht.

Die GUI besteht wie für *Java* typisch aus einem *JFrame*, diversen *JPanel*, einer *JMenuBar*, einer *JToolBar*, sowie einer *JScrollPane* mit einer individuell implementierten Leinwand.
Die Menüleiste und die Werkzeugleiste werden beide von Wrappern[6] verwaltet und besitzen sowohl Standardelemente als auch eigens entwickelte Steuerelemente.
Nachfolgende Grafik gewährt einen kleinen Überblick über die wichtigsten Bestandteile:

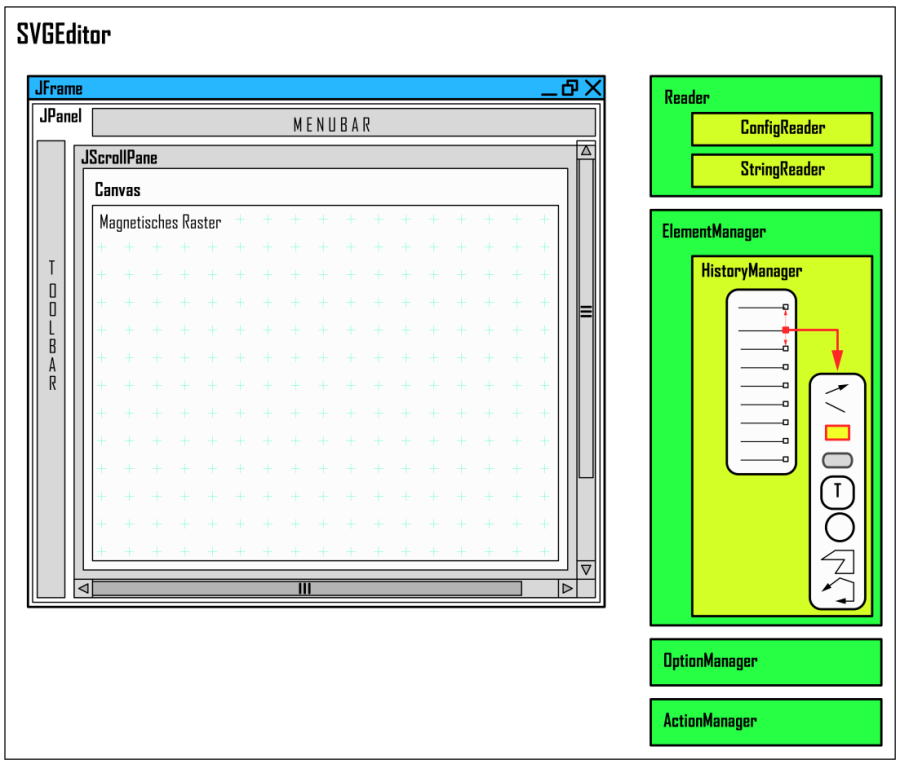


Abbildung 32: SVGEditor Komponenten

[6] Entwurfsmuster aus der objektorientierten Programmierung

Auf den nächsten Seiten werden die meisten Komponenten genauer gezeigt.

Die Reader Komponente wird außen vor gelassen, da diese eigentlich nur dazu dient, Werte von verschiedenen Ressourcen zu lesen wie zum Beispiel ToolTip-Texte sowie Beschriftungen von Menüelementen. Dies macht es einfach, verschiedene Sprachversionen des Programms zu erstellen (z. B. Englisch, Französisch, Spanisch, …)

6.1. Zeichenfläche – *Canvas*

Die Zeichenfläche ist die wichtigste Schnittstelle für den Benutzer um mit dem Programm zu interagieren.

Sie nimmt die Maus-Aktionen entgegen, verarbeitet diese und leitet sie gegebenenfalls an andere Objekte weiter.

Die Leinwand ist außerdem für das Anzeigen der grafischen Elemente zuständig und informiert andere interessierte Klassen mittels der *CanvasStateListener*-Schnittstelle über ihren aktuellen Zustand. Über diese Schnittstelle, wird ein Ereignis übertragen, welches Auskunft über die aktuelle Mausposition, Bewegungsinformationen (Startpunkt, Endpunkt, Delta), selektierte Elemente und vorhandene Elemente gibt.
Durch die zu zeichnenden Elemente „weiß“ die Zeichenfläche stets wie groß sie sein muss und skaliert sich selbstständig.

Die Zeichenfläche „kennt“ sämtliche zu zeichnenden Elemente und kann sich mit Hilfe von Informationen über deren Abmessungen selbstständig skalieren und die Scroll-Balken ein- oder ausschalten

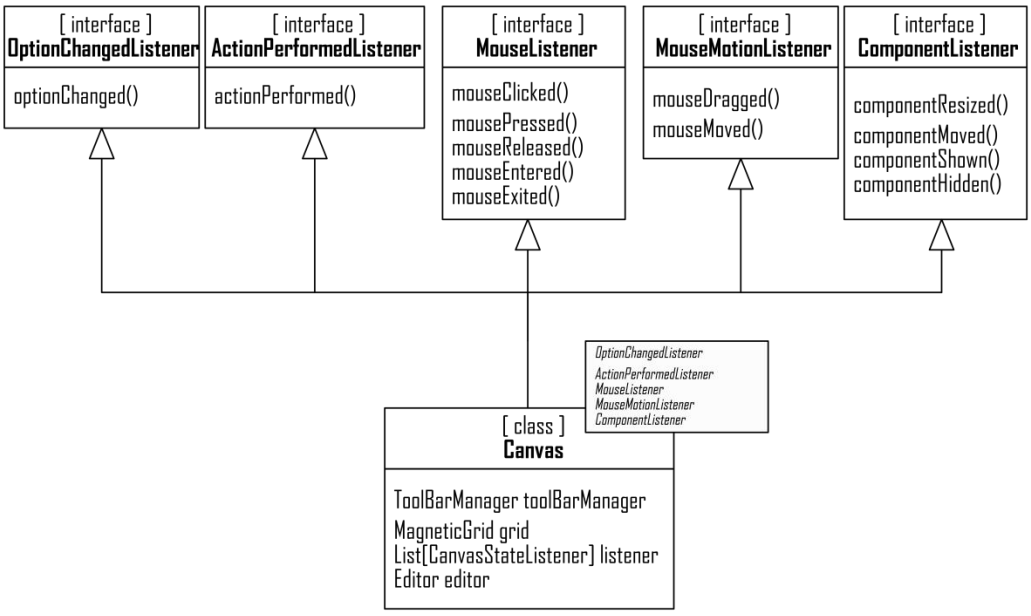


Abbildung 33: Canvas.java

6.2. Magnetisches Raster – *MagneticGrid*

Das magnetische Raster unterstützt den Benutzer beim Arbeiten mit Pixel-genauem Einrasten der Elemente.

Das von der Leinwand verwaltete Raster besitzt zwei unterschiedliche Ausprägungen.

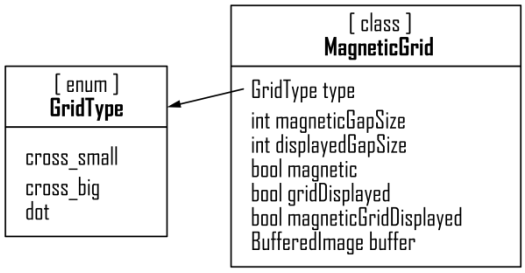


Abbildung 34: MagneticGrid.java

Das magnetische Raster beeinflusst das Verhalten der Maus beim Arbeiten und kann in seiner Größe vom Benutzer verändert werden. Das sichtbare Raster dient dem Benutzer als kleine Orientierungshilfe.

Damit die beiden Rastergrößen sich nicht in die Quere kommen, ist das angezeigte Raster immer ein Vielfaches des magnetischen. Die Größe des angezeigten Rasters kann nur indirekt – durch das Einstellen der Abstände des magnetischen Rasters – beeinflusst werden.

Das gerenderte Raster wird in einem *BufferedImage* gecached und nur im Bedarfsfall aktualisiert. Dies erspart bei jedem Renderingvorgang einige Iterationen, welche wertvolle Zeit in Anspruch nehmen würden.

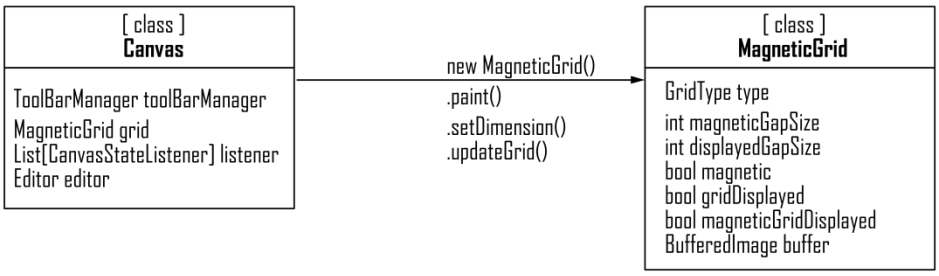


Abbildung 35: Canvas-Grid Kommunikation

6.3. Objektverwaltung – *ElementManager*

Der *ElementManager* verwaltet sämtliche erstellten grafischen Elemente.
Er ist dafür verantwortlich, dass Eigenschaftsänderungen an die betreffenden Elemente weitergeleitet werden.
Außerdem kümmert sich diese Klasse um das korrekte Ausführen von Aktionen wie zum Beispiel das Löschen, Einfügen und Duplizieren.

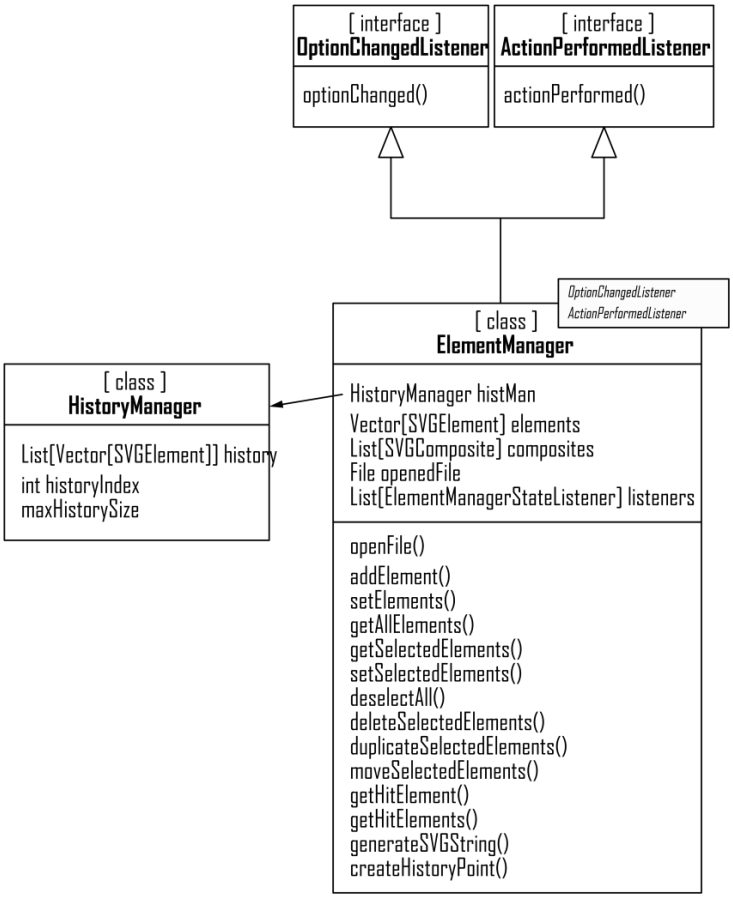


Abbildung 36: ElementManager.java

6.3.1. *Element-Verwaltung*

Bei der Verwaltung der erstellten Elemente handelt es sich hauptsächlich um die Möglichkeiten

- neue Elemente einzufügen,
- bestehende Elemente zu löschen,
- selektierte Elemente zu suchen,
- Elemente zu (de)selektieren und
- Hit-Tests auf der gesamten Zeichenfläche durchzuführen.

Einen Teil der Verwaltungsaufgaben übernimmt der abgesonderte *HistoryManager*, welcher – wie der Name schon sagt – für die Abbildung des zeitlichen Arbeitsverlaufes verantwortlich ist.

6.3.2. Element-Eigenschaften

Werden über die entsprechenden Steuerelemente Elementeigenschaften geändert, verbreitet der *OptionManager* dieses Ereignis.

Der *ElementManager* reagiert auf das gefeuerte *optionChanged*-Ereignis und leitet dieses umgehend an alle selektierten Elemente weiter.

Wie diese Eigenschaftsänderung zu handhaben ist, „weiß“ jedes der Elemente selbst.

6.3.3. Aktionen

Anders als Eigenschaften, ändern Aktionen die interne Repräsentation der Elemente nicht. Analog zu den Eigenschaften können Aktionen nicht nur einzelne Elemente, sondern auch eine Menge von selektierten Elementen im gleichen Maße beeinflussen.

So können zum Beispiel mehrere selektierte Elemente gemeinsam in einem Schritt dupliziert oder verschoben werden.

Mittels Aktionen kann der *ElementManager* auch angewiesen werden

- die Elemente in ein bestimmtes Exportformat umzuwandeln,
- selektierte Elemente zu kopieren,
- Elemente aus der Zwischenablage einzufügen,
- Elemente aus einer Datei einzufügen,
- die Elemente in der gewünschten Art und Weise auszurichten,
- Gruppierungen zu erstellen bzw. aufzulösen oder
- den Bearbeitungsmodus für Gruppierungen zu starten oder zu beenden.

6.4. Zeitliche Abfolge – *HistoryManager*

Um dem Anwender das Korrigieren von Fehlern zu erleichtern, wird eine Chronik der bis zum akutellen Zeitpunkt durchgeführten Aktionen vom *HistoryManager* verwaltet.

Diese Chronik kann benutzt werden, um Aktionen rückgängig zu machen oder diese danach wieder herzustellen.

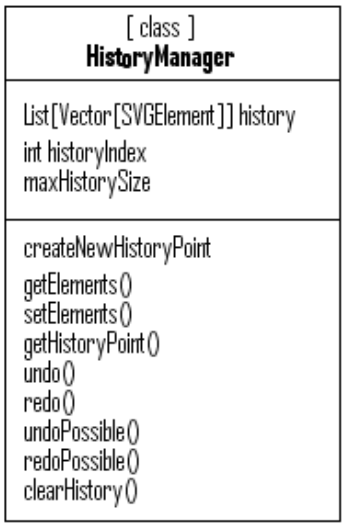


Abbildung 37: HistoryManager.java

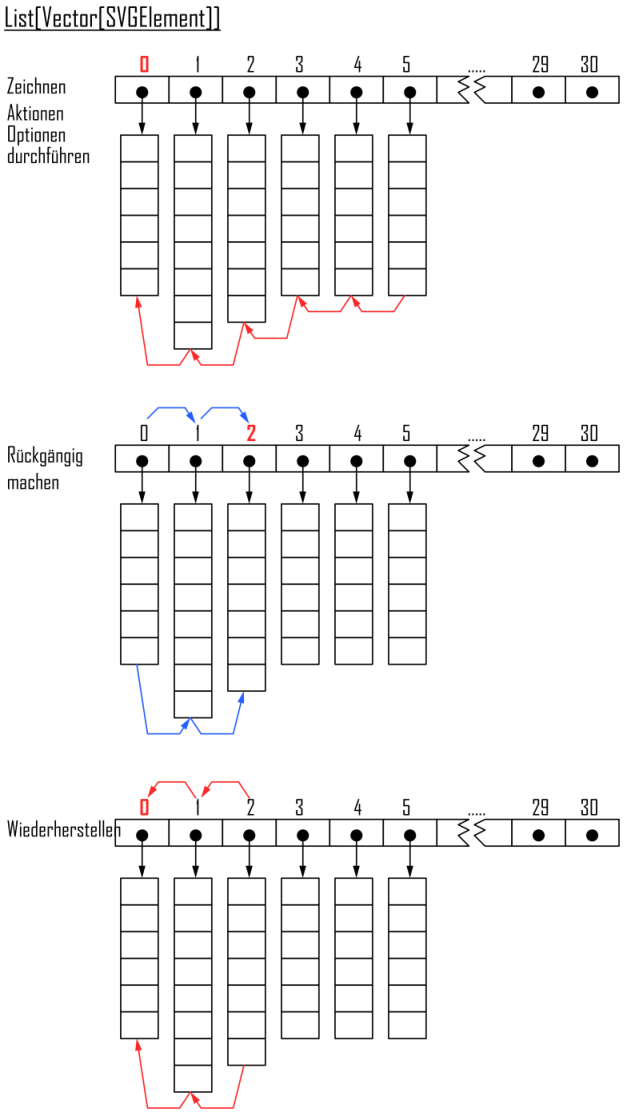


Abbildung 38: Chronik intern

Die Chronik ist als Liste von Schnappschüssen implementiert.

Bei jeder Aktion welche einen Chronik-Eintrag rechtfertigt, werden die aktuell vorhandenen Elemente gesichert und vorne angefügt.

Dies geschieht zum Beispiel beim Zeichnen, Verschieben, Verzerren, Ändern von Eigenschaften und wenn Aktionen ausgelöst werden.

Um zu einem früheren Zustand zu wechseln oder einen späteren wieder herzustellen muss lediglich der Index für den Zugriff auf die Liste verändert werden.

6.5. Grafische Elemente – *SVGElement*

Jeder Bestandteil eines erstellten Bildes benötigt eine proprietäre Repräsentation welche die Grafiksoftware verarbeiten kann. *SVGElement* ist die Oberklasse all dieser grafischen Elemente.
Von ihr erben sie eine Fülle von Eigenschaften und Grundfunktionen.

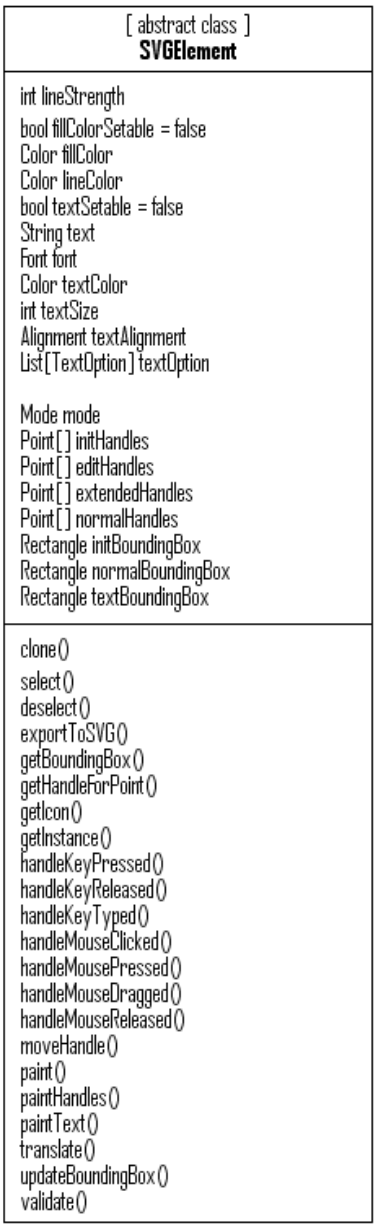


Abbildung 39: SVGElement.java

Ein neues Element muss nur noch die wichtigsten Methoden überschreiben, welche es braucht um erstellt, bearbeitet und dargestellt zu werden.
(*Mouse-Pressed/-Dragged/Released, paint, validate, getInstance, clone, ...*)

Durch diese Oberklasse wird die Entwicklung von neuen Elementen vereinfacht, denn dadurch kann eine Menge doppelter Quellcode eingespart werden und durch die Wiederverwendung reduziert sich die Fehlerhäufigkeit.

Manche Elemente sind mit Besonderheiten ausgestattet.
SVGLine und *SVGPolygone* besitzen z. B. eine zusätzliche Ausprägung zum Setzen von Pfeilspitzen.
SVGRectangle drückt eigentlich drei verschiedene Elemente aus welche über einen Typ unterschieden werden (das normale Rechteck, ein Rechteck mit abgerundeten Ecken und ein Rechteck mit zwei runden Seiten).

Gruppierungen werden durch Objekte der Klasse *SVGComposite* abgebildet und enthalten keine Informationen über ihr Aussehen. Das Erscheinungsbild einer Gruppe ergibt sich aus dem Aussehen der einzelnen Elemente.

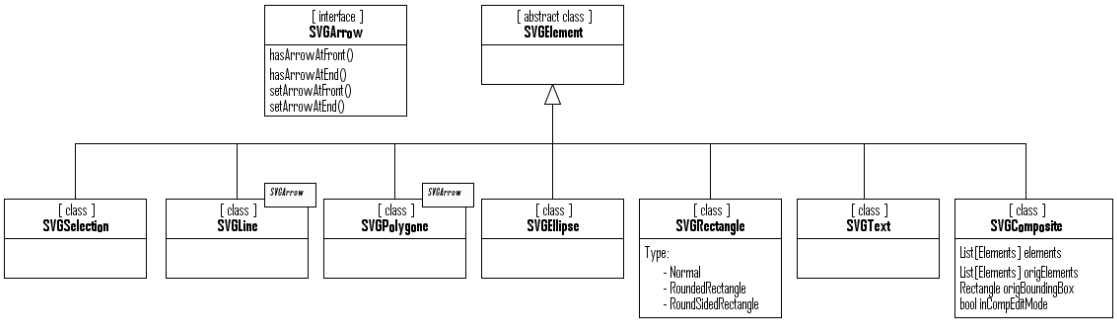


Abbildung 40: Vererbung grafischer Elemente

6.6. Werkzeuge und Eigenschaften – *ToolBar*

Die Toolbar beinhaltet die Zeichenwerkzeuge, sowie einige der Steuerelemente zum Ändern von Element Eigenschaften und wird von einer Klasse namens *ToolBarManager* verwaltet.

Der *ToolBarManager* ist für das Anlegen der ToolBar-Elemente verantwortlich und sorgt für das korrekte Wechseln zwischen den Werkzeugen aufgrund der Benutzereingaben.

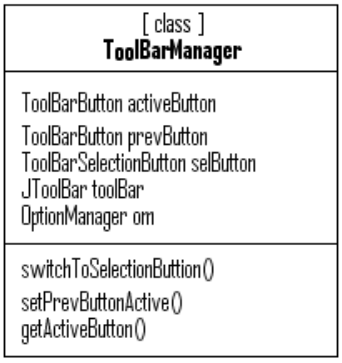


Abbildung 41: ToolBarManager.java

Die Toolbar setzt sich aus verschiedenen Werkzeug-Steuerelementen zusammen, welche alle von der Oberklasse *ToolBarButton* abgeleitet sind.

Die benötigten Werkzeuge können grob in wenige Kategorien zusammengefasst werden. Diese Werkzeugkategorien werden durch vier Elemente in der Toolbar repräsentiert:

- *ToolBarSelectionButton*: Selektionswerkzeuge
- *ToolBarDrawingButton*: Zeichenwerkzeuge (Linie, Pfeil, Rechteck, …)
- *ToolBarActionButton*: Aktionen ausführen (gruppieren, löschen, …)
- *ToolBarOptionButton*: Eigenschaften ändern (Linienstärke, Linienfarbe)

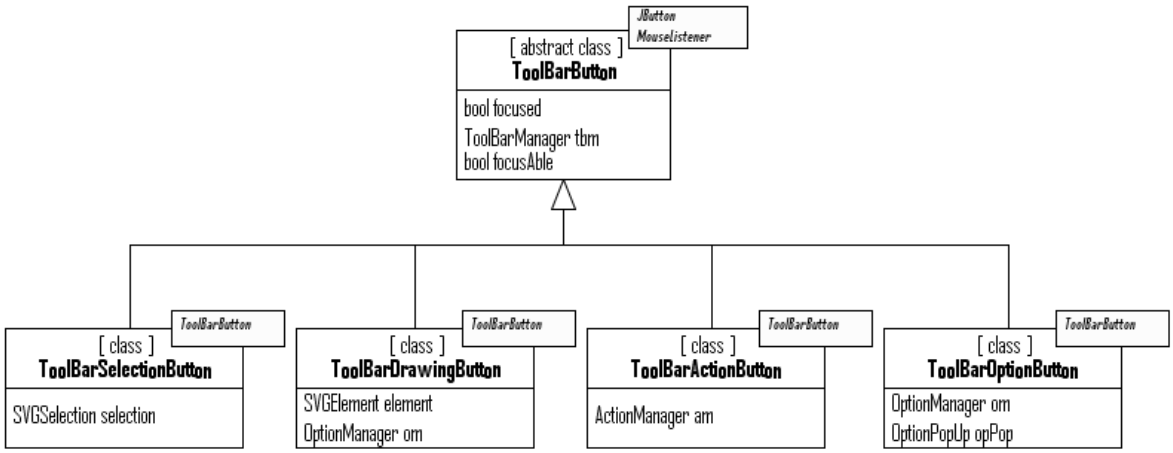


Abbildung 42: ToolBarButton

6.6.1. ToolBarSelection- und ToolBarDrawingButton

Das Selektions- und das Zeichenwerkzeug verhalten sich sehr ähnlich.

Beide Werkzeuge besitzen ein grafisches Element, welches bei der Verwendung als Vorlage dient. Das Zeichenwerkzeug besitzt das jeweilige zu zeichnende Element und das Selektionswerkzeug hat eine Vorlage für ein Auswahlrechteck hinterlegt.

Ist eine dieser Werkzeugklassen aktiv, wird bei einem Klick auf die Leinwand eine neue Instanz der hinterlegten Vorlage erstellt.

Dieses neue Element befindet sich nun im Erstellmodus und nimmt – bis es diesen Zustand verlässt – sämtliche Mausaktionen zur Verarbeitung entgegen.

Im Gegensatz zum Auswahlrechteck, bekommt ein richtiges grafisches Element noch die aktuell gesetzten Eigenschaften vom *OptionManager*.

Das Auswahlwerkzeug besitzt als grafische Repräsentation ein gewöhnliches Auswahlrechteck, falls dieses durch das Ziehen der Maus aufgespannt wird. Das Auswahlwerkzeug kann – im Gegensatz zu einem Zeichenwerkzeug – dazu benutzt werden bei anderen Elementen den Modus zu wechseln (z. B. durch Selektieren) und diese zu bearbeiten.

6.6.2. ToolBarActionButton

Ein Aktionsschalter kann dazu verwendet werden, Kommandos an den *ElementManager* zu senden, welche entweder auf die gesamte Zeichenfläche oder auf bereits selektierte Elemente Einfluss nehmen.

Die meisten der zur Verfügung stehenden Aktionen sind allerdings bereits über die Menüleiste zu erreichen.

6.6.3. ToolBarOptionButton

Die Eigenschaften der Elemente werden mit Hilfe des OptionManagers verändert. Um dem OptionManager die nötigen Anweisungen zukommen zu lassen, werden in der Toolbar Optionswerkzeuge angeboten.

Optionswerkzeuge zählen zu den kompliziertesten Steuerelementen der Werkzeugleiste. Aufgrund unterschiedlicher Arten von Eigenschaften, werden verschiedene grafische Bedienoberflächen benötigt.

Um diese beliebig erweitern zu können ohne neue ToolBar-Werkzeuge erstellen zu müssen, wurde das Auswählen der Eigenschaften über Pop-ups realisiert.

Wird in der Werkzeugleiste ein Optionswerkzeug ausgewählt, erscheint ein kleines Pop-up-Menü welches speziell für diese Element-Eigenschaft entwickelt wurde.

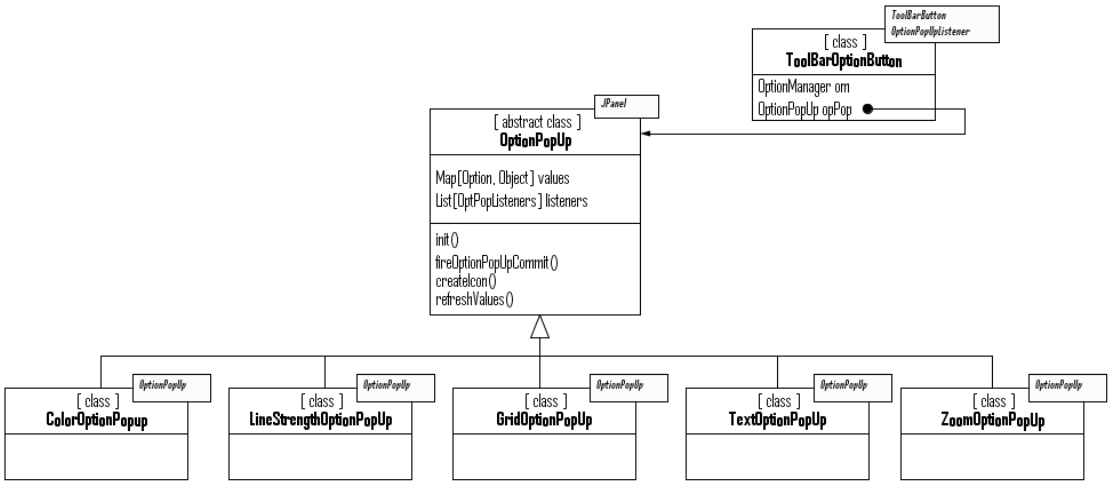


Abbildung 43: OptionPopUp

6.7. Das Menü – *MenuBarManager*

Die Menüleiste bietet dem Benutzer zahlreiche Befehle zur Auswahl an.
Ihre Strukturierung orientiert sich an den üblichen Standards für Datei- und Anwendungsmenü-Menüs.

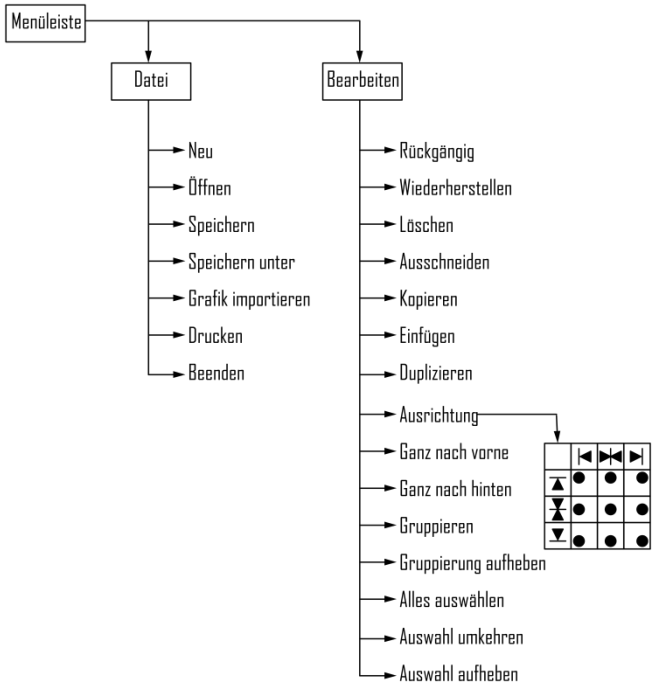


Abbildung 44: Struktur der Menüleiste

Die Menüleiste wird nicht nur dazu verwendet die verschiedenen Befehle auszuführen. Eine wichtige Aufgabe ist auch das Anzeigen von verschiedenen Daten, welche dem Anwender beim Entwurf seiner Grafiken behilflich sind.

Wie auch bei der Werkzeugleiste gibt es eine Manager-Klasse namens *MenuBarManager* welche die Menüleiste verwaltet.
Diese Manager-Klasse ist für den Aufbau der Menüleiste zuständig, und sorgt für das korrekte Ausführen der gewählten Befehle.
Wie dem Klassendiagramm zu entnehmen ist, erhält der *MenuBarManager* verschiedene Informationen von der *Leinwand* und vom *ElementManager*.

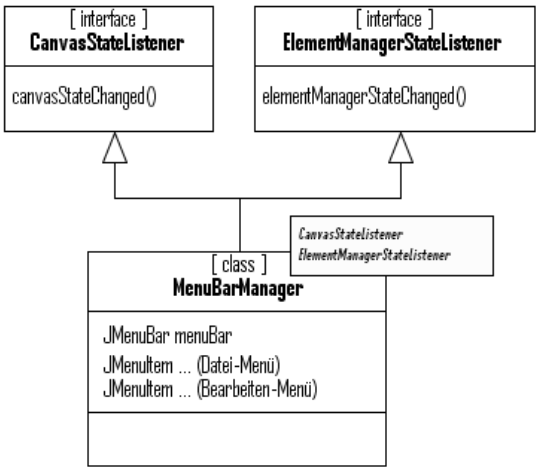


Abbildung 45: MenuBarManager.java

Von der Leinwand kommen Informationen über Mausinteraktionen und Selektionen. Diese werden dazu benutzt das Maus-Delta, die Mausposition und die Objektgröße anzuzeigen.

Da die Ausführbarkeit einiger Menüaktionen vom Vorhandensein von Elementen oder selektierten Elementen abhängen kann, werden diese – mit Hilfe der Informationen des *ElementManagers* – ausgegraut, falls die Bedingungen nicht erfüllt werden.

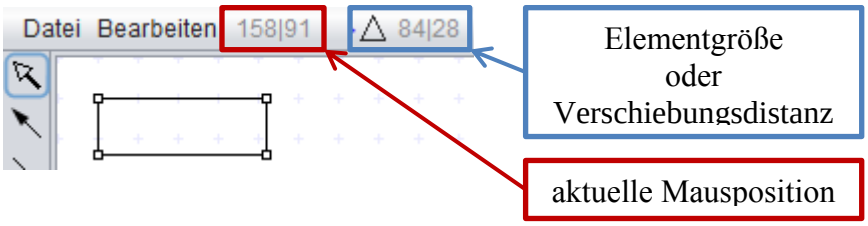


Abbildung 46: Informationsanzeige in der Menüleiste

6.8. Eigenschaften ändern

Das Ändern der Standardeigenschaften oder der Eigenschaften selektierter Elemente übernimmt der *OptionManager*.

Bei dieser Klasse werden auch die Standard Eigenschaftswerte abgefragt wenn neue Elemente gezeichnet werden.

Für eine Änderung muss lediglich ein *OptionChangedEvent* gefeuert werden. Dieses wird an den *ElementManager* weitergeleitet, welcher sich darum kümmert, dass jedes der selektierten Elemente von der Änderung erfährt.

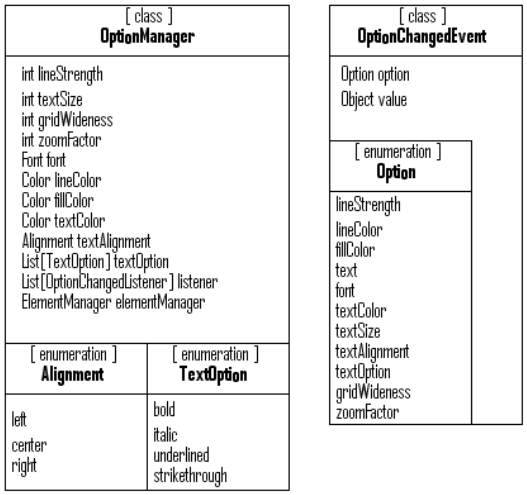


Abbildung 47: OptionManager.java

6.9. Aktionen durchführen

Sämtliche Aktionen innerhalb des Programms werden über die statische Klasse *ActionManager* abgewickelt.

Jede Klasse kann sich als Empfänger für so genannte *ActionPerformedEvents* beim *ActionManager* eintragen und kann anschließend auf jedes gefeuerte Ereignis reagieren.

Der aktuell wichtigste Zuhörer ist der *ElementManager*. Er übernimmt die meisten Aufgaben der Elementverwaltung, sowie das Öffnen und Speichern der erstellten Grafiken.

Das *ActionPerformedEvent* muss vor dem Auslösen entsprechend konfiguriert werden.

Ein Ereignis muss mindestens mit einem Aktionstyp ausgestattet werden, welcher von einer – stetig erweiterbaren – Enumeration stammt.

Um mehr Flexibilität zu erhalten, können Ereignisse noch zusätzliche Informationen enthalten. Für die Anordnung der Elemente kann zum Beispiel das ausgewählte *Alignment* übergeben werden.

Werden beliebige Werte als *Object* übergeben, so muss der Empfänger – für eine erfolgreiche Verarbeitung dieser – stets wissen, welche Daten er bekommt.

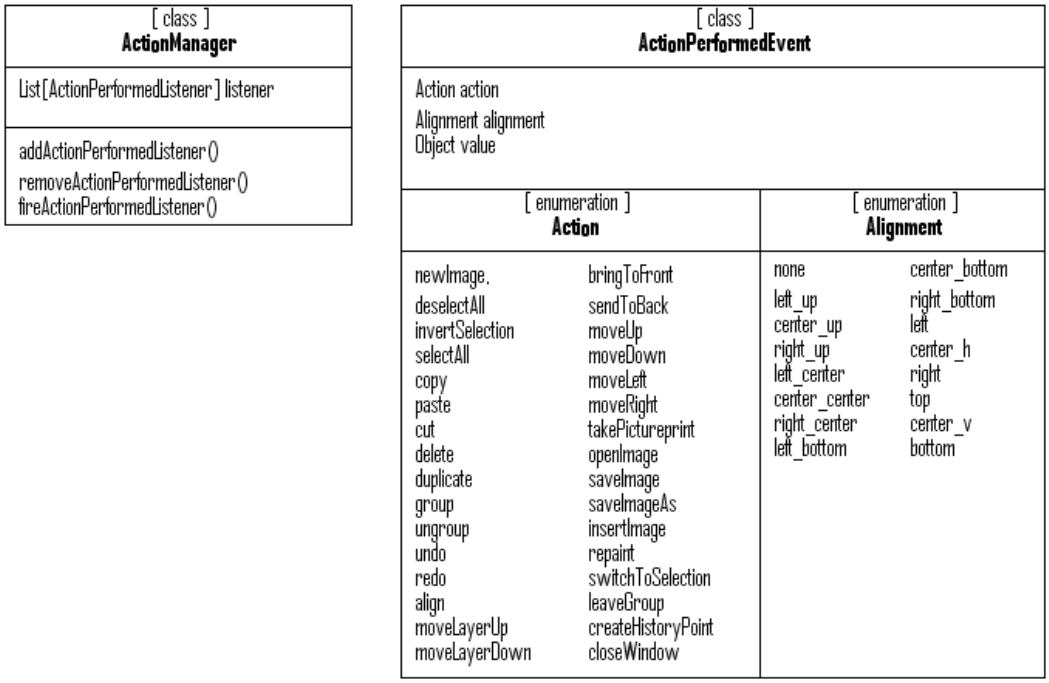


Abbildung 48: ActionManager.und ActionPerformedEvent

7. Implementierung

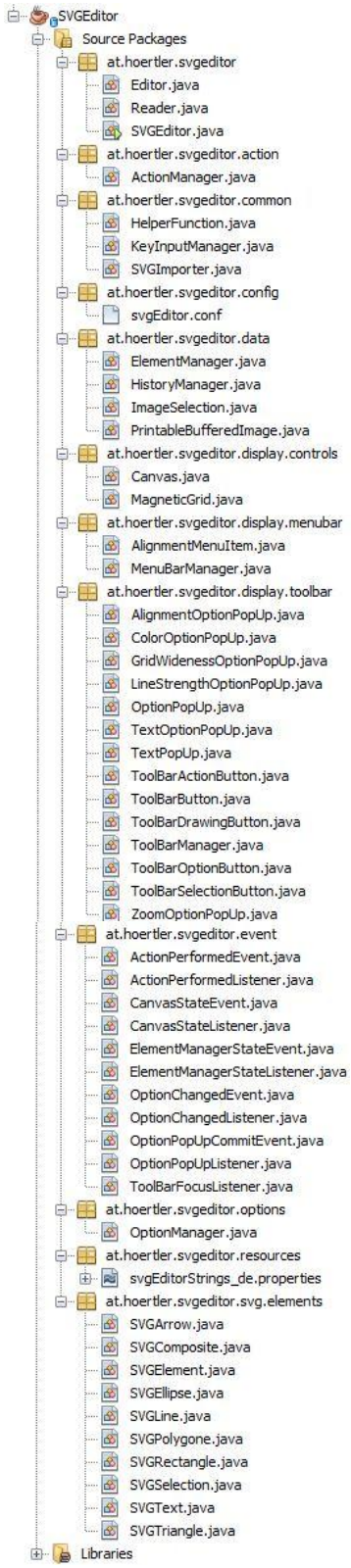


Abbildung 49: Java Packages Überblick

Das folgende Kapitel beschäftigt sich mit den Details der Implementierung.

Um den nötigen Überblick zu bewahren, wird auf der linken Seite (anhand einer Abbildung aus dem Projekt-Explorer der Java IDE NetBeans) die physische Dateiablage gezeigt.

Viele der Dateien sind durch sprechende Namen selbsterklärend, weshalb nicht auf jede einzeln eingegangen wird.
Auf den folgenden Seiten, werden die wichtigsten Aspekte beleuchtet und wird das Zusammenspiel einiger Klassen erklärt.

7.1. Leinwand

Die Leinwand ist wie in der realen Welt dafür zuständig ein Bild kreieren und anzeigen zu können.
Die Leinwand wird durch die Klasse „*Canvas*“ repräsentiert. Sie besitzt eine *paint()*-Methode um die entwickelten Elemente darzustellen und reagiert auf Benutzereingaben mittels Maus (*press, drag, release, click, move, …*).

7.1.1. Elemente zeichnen

Das Zeichnen von neuen Elementen erfolgt wie üblich mit Hilfe der Maus.
Hierzu bedient sich die Leinwand der Ereignismethoden aus den Java Schnittstellen *MouseListener* und *MouseMotionListener*.

Beim Zeichnen und Selektieren kommen die Ereignismethoden des *MouseListeners* zum Einsatz.
Die Leinwand ist nicht für die korrekte Interpretation der Mauseingaben zuständig, sondern lediglich dafür, dass Interaktionen an das Raster angepasst und anschließend an die richtigen Interessenten weitergeleitet werden.

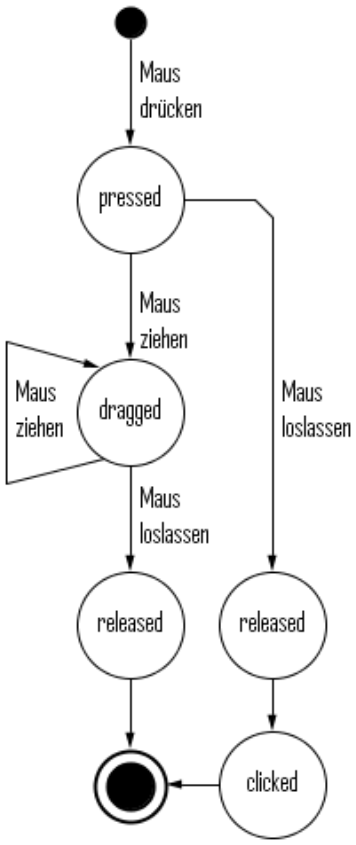


Abbildung 50: Canvas MouseListener

Maus bewegen

Bewegungen der Maus ohne gedrückte Maustaste sind für die Leinwand meistens nur für so genannte „*Mouse-Hover*“-Effekte von Bedeutung und dienen der Aktualisierung des Mauszeiger Symbols durch Abfrage der unter dem Zeiger befindlichen Elemente. Um diese Daten zu erhalten, wird das *MouseMoved*-Ereignis des *MouseMotionListener*s verwendet.

Maustaste drücken

Da die Leinwand beim Drücken der Maustaste versucht einen Empfänger für die eingehenden Informationen zu finden, wird der *ToolBarManager* gefragt welches Werkzeug gerade aktiviert ist.

Zeichen- und Selektionswerkzeuge besitzen hierfür ein geeignetes *SVGElement*, welches Mauseingaben verarbeiten kann.

Wie bereits erwähnt, ist die Leinwand dafür zuständig, die Daten von Mausinteraktionen an das magnetische Raster anzupassen. Diese Anpassung soll jedoch nur beim Zeichnen und beim Bearbeiten von grafischen Elementen, nicht aber beim Selektieren vorgenommen werden. Aus diesem Grund unterscheidet die Leinwand zwischen Zeichen- und Selektionswerkzeugen und überprüft beim Selektionswerkzeug zusätzlich, ob ein Element bearbeitet oder selektiert werden soll.

```
public void mousePressed(MouseEvent e) {
  // Get selected ToolBarButton
  ToolBarButton tbb = this.toolBarManager.getActiveButton();

  // For SelectionButton
  if (tbb.getClass().isAssignableFrom(ToolBarSelectionButton.class)) {
    this.activeElement = null;
    this.selection = ((ToolBarSelectionButton) tbb).getSelection();
    // Let SVGSelection handle the mousePressed Event
    this.selection.handleMousePressed(e);
  // For DrawingButton
  } else if (tbb.getClass().isAssignableFrom(ToolBarDrawingButton.class)) {
    this.handleGrid(e);
    this.selection = null;
    this.activeElement = null;
    this.activeElement = ((ToolBarDrawingButton) tbb).getElement();
    // Let SVGElement handle the mousePressed Event
    if(this.activeElement.isInitialCreateMode())
      this.activeElement.handleMousePressed(e);
  }
  this.grabPoint = e.getPoint();
  this.releasePoint = e.getPoint();
}
```

Maus mit gedrückter Taste ziehen

Beim Ziehen der Maus mit gedrückter Maustaste wird mit einem Verschiebungsvektor gearbeitet, der vom Startpunkt (siehe *mousePressed()*) auf die aktuelle Mausposition zeigt. Dieser Verschiebungsvektor wird – wenn notwendig – an das magnetische Raster angepasst und – entsprechend zusätzlicher definierter Tastaturkommandos –

umgerechnet (ALT → Raster ignorieren, SHIFT → x-/y-Achse fixieren, STRG → Proportionen beibehalten).

```
public void mouseDragged(MouseEvent e) {
 // DeltaPoint for dragging operation
 Point deltaPoint = new Point(0,0);
 // Calculate delta between grab point and actual point
 deltaPoint = HelperFunction.calculateTranslationPoint(this.grabPoint,e.getPoint());
 // Update release point
 this.releasePoint = e.getPoint();
 // Handle ALT-Key: If ALT is pressed then grid is ignored (pixel steps)
 if(!e.isAltDown()) {
  // ALT is not pressed -> calculate nearest grid point
  Point trans = this.getGridTranslationPoint(deltaPoint);
  // and adjust the delta
  deltaPoint.translate(trans.x, trans.y);
 }
 // deltaPoint represents the whole dragging process
 // now translate the deltaPoint to get the grid aligned release point
 deltaPoint.translate(this.grabPoint.x, this.grabPoint.y);
 // calculate the delta between actual mouse position and the wanted release point
 deltaPoint = HelperFunction.calculateTranslationPoint(e.getPoint(), deltaPoint);
 if(e.isShiftDown()) { // if SHIFT is pressed -> fix X- or Y-Axis
  Point trans = HelperFunction.calculateTranslationPoint(
                                    e.getPoint(),this.grabPoint );
  /* ... CALCULATE AXIS FIXED TRANSLATION ... */
 }
 // translate the point of the mouse event to the wanted release point
 e.translatePoint(deltaPoint.x, deltaPoint.y);

 // For selection
 if (this.selection != null) {
  // Let SVGSelection handle the mousePressed Event
  this.selection.handleMouseDragged(e);
 // For drawing
 } else if (this.activeElement != null &&
         this.activeElement.isInitialCreateMode()) {
  // if CTRL is pressed, the element must keep its proportions.
  if(e.isControlDown()) {
   Point trans = HelperFunction.calculateTranslationPoint(
                                     this.grabPoint, e.getPoint() );
   /* ... CALCULATE PROPORTION KEEPING TRANSLATION ... */
   // Let SVGElement handle the mouseDragged Event
   this.activeElement.handleMouseDragged(e);
  } else {
   // Let SVGElement handle the mouseDragged Event
   this.activeElement.handleMouseDragged(e);
  }
 }
 // Fire a CanvasStateEvent to keep all interested listeners up to date
 this.fireCanvasStateEvent(new CanvasStateEvent(. . .));
 tryRepaint(e.getPoint());        // Only repaint if really neccessary
}
```

Maustaste loslassen

Beim Loslassen der Maustaste wird die aktuelle Interaktion beendet.

Wurde ein neues Element gezeichnet, ist dieses mit dem Ereignis des Loslassens fertiggestellt, wird abschließend noch einmal überprüft und wenn alles in Ordnung ist eingefügt.

```
public void mouseReleased(MouseEvent e) {
  // For selection
  if (this.selection != null) {
    // Let SVGSelection handle the mouseReleased Event
    this.selection.handleMouseReleased(e);
    // Adjust the canvas size if neccessary
    this.adjustCanvasSize(this.selection.getActiveElement());
    // Fire a CanvasStateEvent to keep all interested listeners up to date
    this.fireCanvasStateEvent(. . .));
  // For drawing
  } else if (this.activeElement != null) {
    this.handleGrid(e);
    // Let SVGElement handle the mouseReleased Event
    this.activeElement.handleMouseReleased(e);
    // If element is new
    if(!getAllElements().contains(this.activeElement)) {
      boolean added = false;
      // Check elements validation
      if (this.activeElement.validate()) {
        added = addElement(this.activeElement);
        // Adjust the canvas size if neccessary
        this.adjustCanvasSize(this.activeElement);
      }
      // If element couldn't be added -> switch to selection button
      if (!added) this.toolBarManager.switchToSelectionButton();
    }
  }
  this.grabPoint = new Point(0,0);
  this.repaint();
}
```

Maustaste klicken

Als „klicken“ wird der Vorgang des Drückens und Loslassens der Maustaste ohne Positionsveränderung bezeichnet.

Dabei gibt es drei mögliche Szenarien.

Ist das Selektionswerkzeug aktiv, wird das *MouseEvent* an dieses weitergeleitet und das hinterlegte *SVGSelection* Element kümmert sich um den Rest.

Ist ein Zeichenwerkzeug aktiv, will der Benutzer entweder auf das Selektionswerkzeug wechseln, oder das zu zeichnende Element besitzt einen Erweiterten Erstellmodus welcher auf Klick-Ereignisse reagiert. Ist ein solcher erweiterter Erstellmodus vorhanden wird das *MouseEvent* an das aktive *SVGElement* weitergeleitet.

```
public void mouseClicked(MouseEvent e) {
 ToolBarButton tbb = this.toolBarManager.getActiveButton();
 // For DrawingButton
 if (!tbb.getClass().isAssignableFrom(ToolBarSelectionButton.class)) {
  this.selection = null;
  // If there is no element or
  // an element which doesn't support ExtendedCreate/-Edit Mode or
  // TextEditMode
  if(this.activeElement == null ||
     ((!activeElement.isExtendedCreateMode() &&
       !activeElement.isExtendedEditMode()) &&
      (!activeElement.getClass().isAssignableFrom(SVGText.class) ||
        !activeElement.isTextEditMode())
     )) {
   // Switch to SelectionButton and deselect all existing elements
   this.toolBarManager.switchToSelectionButton();
   this.activeElement = null;
   this.editor.getElementManager().deselectAll();
  } else {
   // if there is an element in an Extended Mode
   // forward the mouseClicked event
   this.handleGrid(e);
   this.activeElement.handleMouseClicked(e);
  }
 // For SelectionButton
 } else {
  if(this.selection != null)
   this.selection.handleMouseClicked(e);
 }
 this.selection=null;
}
```

7.1.2. Elemente darstellen

Die zweite essentielle Aufgabe der Leinwand ist das Darstellen der vorhandenen Elemente.

Das Rendern der gezeichneten Elemente erfolgt – wegen der Überdeckung entsprechend der Z-Achsen Hierarchie – in einer bestimmten Reihenfolge.

Zuerst wird das Raster gezeichnet, da dieses unter allen Elementen liegen soll.

Dem Raster folgen die bestehenden Elemente in der Reihenfolge in der diese erstellt bzw. in der sie auf der Z-Achse angeordnet wurden.

Anschließend folgt die Darstellung der aktuellen Interaktion mit der Leinwand. Dies kann entweder ein Auswahlrechteck oder ein Element sein, welches gerade erstellt wird.

Da die Handles der selektierten Elemente immer sichtbar sein sollen werden diese als Letztes gezeichnet.

```
public void paint(Graphics g) {
 super.paint(g);
 this.magneticGrid.paint((Graphics2D)g.create());

 // Paint elements
 Vector<SVGElement> handles = new Vector<SVGElement>();
 Vector<SVGElement> remove = new Vector<SVGElement>();
 for (SVGElement elem : getAllElements()) {
  if(!elem.isSelected() && !elem.validate())
   remove.add(elem);
  else {
   elem.paint(g, this.getZoomFactor());
   if (elem.isSelected()) // Store handles if element is selected
    handles.add(elem);
  }
 }
 // Remove elements that dont validate
 if(!remove.isEmpty())
  getAllElements().removeAll(remove);

 // Paint selection or element while creating
 if (this.selection != null) {
  this.selection.paint(g, this.getZoomFactor());
 } else if (this.activeElement != null) {
 if(!getAllElements().contains(this.activeElement))
  this.activeElement.paint(g, this.getZoomFactor());
 }

 // Paint Handles of selected elements
 for (SVGElement elem : handles)
  elem.paintHandles(g, this.getZoomFactor());
}
```

Da das Rendern einer großen Anzahl von Elementen sehr aufwändig sein kann, muss das erneute Zeichnen auf ein Minimum beschränkt werden.

Die Zeichenfläche kann nur bei *MouseDragged()*- und *MouseMoved()*-Ereignisse aktualisiert werden.

Bei diesen zwei Ereignissen wird die Methode *tryRepaint()* aufgerufen, in welcher darüber entschieden wird, ob es wirklich notwendig ist, die Leinwand aufzufrischen.

Außerdem wird hier der *Garbage Collector*[7] manuell aufgerufen um die speicherhungrige Java-VM im Zaum zu halten.

```
private boolean tryRepaint(Point p) {
  boolean repaint = false;
  if (this.startPoint == null) {
    this.startPoint = p;
  } else {
    if(startPoint.distance(p) >= magneticGrid.getMagneticGapSize()) {
      repaint = true;
      this.startPoint = p;
    }
  }
  if (repaint) {
    System.gc();
    System.runFinalization();
    this.repaint();
  }
  return repaint;
}
```

[7] Routine zur automatischen Speicherbereinigung

7.2. Magnetisches Raster

Das Raster hat zwei Bestandteile:

1. nicht sichtbares magnetisches Raster
2. sichtbares Raster

Der Abstand des sichtbaren ist immer ein Vielfaches des Abstands des magnetischen Rasters. Dadurch wird gewährleistet dass beide sich nicht in die Quere kommen und für Unstimmigkeiten in der Anzeige sorgen.

Das magnetische Raster kann nur mit Punkten dargestellt werden.
Das sichtbare Raster hingegen kann unterschiedliche Formen annehmen:

kleines Kreuz:
(Standard)

großes Kreuz:

Punkt:

Das Raster wird im Gegensatz zu den einzelnen Elementen nicht bei jedem Zeichenvorgang neu erstellt.
Um den Rendering Prozess nicht unnötig zu belasten, wird das Gitternetz in ein *BufferedImage* gezeichnet, im Speicher gehalten und nur im Bedarfsfall aktualisiert.
Um eine Aktualisierung des Rasterbildes manuell anzustoßen kann die Methode *MagneticGrid.updateBuffer()* aufgerufen werden.
Der Anwender kann das Raster neu erstellen lassen indem er den Rasterabstand oder die Leinwandgröße verändert.

7.3. PDF-Dokumente erzeugen

Auf der Suche nach einer Möglichkeit, die Elemente in einer PDF-Grafik abspeichern zu können, wurde eine frei verfügbare Programmierschnittstelle für Java namens *iText* entdeckt.

iText dient der dynamischen Erzeugung und Bearbeitung von PDF-Dateien und wird unter der *GNU Affero General Public License (AGPL)* [6] vertrieben.

Es ist eine sehr mächtige API welche hier nur bruchstückhaft Verwendung findet [7].

In diesem Kapitel soll gezeigt werden, wie mit Hilfe der *iText*-Api ein Grafikdokument samt Metadaten erstellt werden kann.

7.3.1. Dokument erstellen

Zur Erstellung eines PDF-Dokumentes werden nur wenige *iText*-Klassen benötigt:

Document ..... *Repräsentiert das PDF-Dokument*

PdfWriter ..... *Gibt Zugriff auf den Inhalt des Dokuments*

PdfContentByte ..... *Repräsentiert den Inhalt des Dokuments*

PdfGraphics2D ..... *iText Pendant zu Java Graphics2D*

Nachfolgender Beispielcode zeigt das Erstellen eines Dokumentes und das Zeichnen einer einfachen Linie:

```
Document document = new Document(new Rectangle(<BREITE>, <HÖHE>));
PdfWriter writer = PdfWriter.getInstance(document, new     FileOutputStream(<DATEI.pdf>));
document.open();
PdfContentByte cb = writer.getDirectContent();
Graphics2D g2 = new PdfGraphics2D(cb, <BREITE>, <HÖHE>);
g2.drawLine(5,5,40,50);
g2.dispose();
document.close();
writer.close();
```

Um die Qualität der Grafik zu erhöhen können wie in *Java Graphics2D* einige *RenderingHints* gesetzt werden:

```
g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                    RenderingHints.VALUE_ANTIALIAS_ON);
g2.setRenderingHint(RenderingHints.KEY_RENDERING,
                    RenderingHints.VALUE_RENDER_QUALITY);
g2.setRenderingHint(RenderingHints.KEY_TEXT_ANTIALIASING,
                    RenderingHints.VALUE_TEXT_ANTIALIAS_ON);
g2.setRenderingHint(RenderingHints.KEY_STROKE_CONTROL,
                    RenderingHints.VALUE_STROKE_NORMALIZE);
```

Da das Zeichnen über das *PdfGraphics2D* auf dieselbe Weise funktioniert wie unter Java gewohnt, lassen sich die Elemente problemlos darstellen.

Ein Problem gibt es aber bei Texten.
Um den Text in einem PDF-Dokument korrekt darstellen zu können, werden die vom Autor verwendeten Schriftarten verwendet.
Um sicherzustellen, dass jeder Betrachter das exakt gleiche Bild sieht, gibt es die Möglichkeit, Schriftarten in ein PDF-Dokument einzubetten.
Dadurch erhöht sich zwar die Dateigröße minimal, aber dafür besitzen die Texte auf jedem Anzeigegerät die Schriftart, wie sie vom Autor vorgesehen wurde.
Zum Integrieren der gewünschten Schriftart, werden die *Font*-Objekte der *iText*-API verwendet. Die *BaseFonts* – eine Teilmenge der *Font*-Objekte – können per Parameter angewiesen werden, sich in das Dokument einzubetten.

```
BaseFont.createFont(BaseFont.HELVETICA,     /* Schriftart */
                    BaseFont.WINANSI,       /* Encoding */
                    BaseFont.EMBEDDED);   /* oder NOT_EMBEDDED */
```

Da mit den *BaseFonts* nur wenige Basis-Schriftarten zur Verfügung stehen, kann das Angebot mit Hilfe der *FontFactory* erweitert werden:

```
FontFactory.getFont(<FONT-NAME>, BaseFont.CP1252, BaseFont.EMBEDDED);
```

Alle grafischen Elemente zeichnen sich selbst auf die Leinwand und rendern auch ihren Text selbstständig. Bei *Java Graphics2D* wird die Schriftart mit Hilfe der *setFont()*-Methode konfiguriert.

Um diese Methode kompatibel zu *iText* zu machen, ist ein *FontMapper* nötig, welcher zwischen AWT- und PDF-Schriftarten übersetzt. Dieser *FontMapper* kann an das *PdfGraphics2D* Objekt übergeben werden und erledigt dort seine Aufgaben.

```
FontMapper fm = new FontMapper() {
  public BaseFont awtToPdf(Font font) {
    itextpdf.text.Font f = FontFactory.getFont(
                            font.getName(),
                            BaseFont.CP1252,
                            BaseFont.EMBEDDED
                          );
    BaseFont bf = f.getBaseFont();
    return bf;
  }

  public Font pdfToAwt(BaseFont font, int size) {
    return null;              /* Nicht notwendig da nie verwendet */
  }
};
Graphics2D g2 = new PdfGraphics2D(cb, <BREITE>, <HÖHE>, fm);
```

7.3.2. Metadaten anhängen

Um die Grafik aus dem PDF Format wieder importieren zu können, fehlen jetzt noch die Metadaten. Das Einbetten dieser, gestaltet sich über folgenden Befehl äußerst einfach:

```
document.add(new Meta("unknown", <SVG-STRING>);
```

Mit einem Programm zur Visualisierung von PDF-Dokumenten, können die benutzerdefinierten Eigenschaften – unter denen sich die Metadaten finden – betrachtet werden.

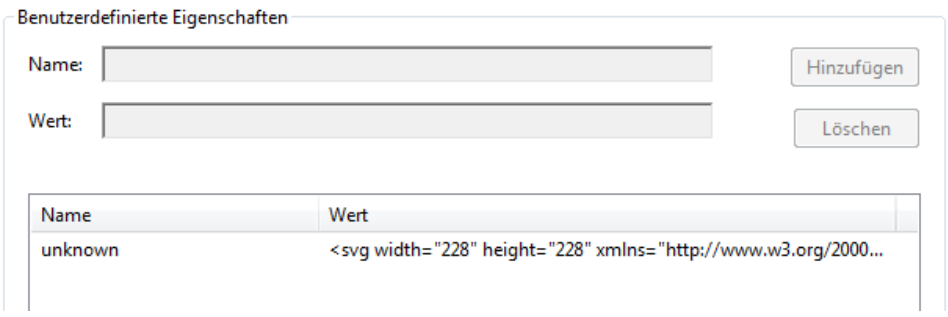


Abbildung 51: Metadaten im PDF Dokument

7.4. PNG-Grafiken generieren

Neben PDF-Dokumenten sollen natürlich auch PNG-Grafiken inklusive Metadaten exportiert werden können.

7.4.1. Grafik erstellen

Wurde in Java ein Bild mit *AWT* oder *Graphics2D* gezeichnet, ist das Erstellen einer Rastergrafik ein Leichtes. Die Klasse *BufferedImage* repräsentiert eine Pixelgrafik, welche – wie beim Zeichnen – über ein Grafikobjekt befüllt werden kann.

```
BufferedImage img = new BufferedImage(
                    <BREITE>,
                    <HÖHE>,
                    BufferedImage.TYPE_INT_ARGB );
Graphics2D g2 = (Graphics2D)img.createGraphics();
for (SVGElement elem : <SVG-ELEMENTE>)
   elem.paint(g2);
```

7.4.2. Metadaten anhängen

Im Gegensatz zu PDF-Dokumenten, ist das Einbetten von Metadaten bei Rastergrafiken nicht so einfach, obwohl das PNG-Format diese unterstützt.

Die Java Klasse *BufferedImage* ist zwar perfekt dazu geeignet Bilder zu erstellen, aber sie kann keine Metainformationen aufnehmen.

Zusätzliche Informationen können nur direkt im Byte-Strom der Grafik eingefügt werden. Um die Daten lesen und schreiben zu können, wurden zwei statische Hilfsmethoden implementiert:

```
public static  byte[] writeCustomData(
   BufferedImage buffImg,
   String key,
   String value
);
public static String readCustomData(
   byte[] imageData,
   String key
);
```

Auch das Bearbeiten der Metadaten ist in Java leider nur direkt im Byte-Strom möglich. Wird der Byte-Strom in ein *BufferedImage* konvertiert, gehen sämtliche zusätzlichen Informationen wieder verloren.

Deshalb darf ein PNG-Bild beim Importieren auch nicht als Bild importiert werden sondern muss als Datenstrom gelesen werden um Zugriff auf die Metadaten zu erhalten.

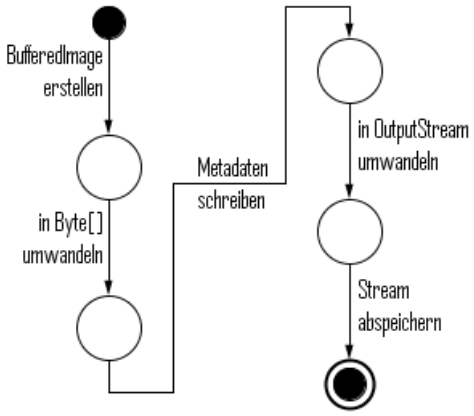


Abbildung 52: PNG-Metadaten schreiben

7.5. SVGElement

Die Klasse *SVGElement* beinhaltet als Oberklasse aller grafischen Elemente die Schnittmenge ihrer Funktionalitäten.

Bei vielen Methoden ist die Implementierung für den Großteil der abgeleiteten Elemente identisch. Hierzu gehören zum Beispiel Methoden zum

- Setzen von Eigenschaften (Linienfarbe, Linienstärke, …),
- Verarbeiten von Tastatureingaben,
- Verschieben des Elementes,
- Ändern des Modus,
- Testen ob ein und welches Handle getroffen wurde,
- Zeichnen der Handles,
- Rendern des Textes
- und viele mehr.

Auf den folgenden Seiten werden einige dieser Funktionen näher betrachtet.

7.5.1. Tastatureingaben

Sämtliche Tastatureingaben werden von einem globalen *KeyListener* abgefangen.

Die *KeyListener*-Schnittstelle besteht aus 3 Methoden:

handelKeyPressed(KeyEvent e)
handleKeyReleased(KeyEvent e)
handleKeyTyped(KeyEvent e)

Der globale Tastatur-Listener gibt ankommende Eingaben an selektierte Elemente weiter indem er deren *handleKey…()*-Methoden aufruft.

KeyPressed

Auf das Drücken einer Taste auf der Tastatur reagiert ein Element nur, wenn es sich um die Löschen Taste handelt.

Dadurch wird erreicht, dass bei längerem Gedrückt halten der Löschen-Taste mehr als ein Zeichen gelöscht werden kann.

```
public void handleKeyPressed(KeyEvent e) {
 switch(e.getKeyCode()) {
  case KeyEvent.VK_BACK_SPACE:
   if(this.getText().length()>0)
    this.setText(this.getText().substring(0, this.getText().length()-1));
   break;
 }
}
```

KeyReleased

Auch beim Loslassen einer Taste wird der Text noch nicht verändert, sondern lediglich dafür gesorgt, dass im Falle einer losgelassenen Return Taste bei gedrückter Shift-Taste der Bearbeitungsmodus nicht verlassen, sondern eine neue Textzeile eingefügt wird.

```
public void handleKeyReleased(KeyEvent e) {
 switch(e.getKeyCode()) {
  case KeyEvent.VK_ENTER:
   if(!e.isShiftDown()) {
    this.setMode(Mode.Edit);
    this.setText(this.getText().substring(0, this.getText().length()-1));
   }
   break;
 }
}
```

KeyTyped

Erst beim Aufruf der *handleKeyTyped()*-Methode wird der Text des selektierten Elementes verändert.

Diese Methode wird nur aufgerufen, wenn es sich um ein Unicode Zeichen handelt, welches eine grafische Repräsentation besitzt.

Bevor ein Zeichen angehängt wird, wird erneut überprüft ob es sich auch wirklich um ein gültiges Zeichen handelt.

Gültige Zeichen sind generell alle darstellbaren Zeichen und *Line Feeds* bzw. *Carriage Returns*.

```
public void handleKeyTyped(KeyEvent e) {
 char c = e.getKeyChar();
 if(
      (this.getFont().canDisplay(c) && c >= 32 /* Space */ && c < 127 /* DEL */
          && c != 60 /* < */ && c != 62 /* > */)
      || c==10 /* NL */ || c==13 /* CR */) {
  this.setText(this.getText()+c);
 }
}
```

7.5.2. Rendern von Texten

Nachdem die Tastatureingaben an die Elemente weitergeleitet wurden und überprüft wurde, dass diese einen darstellbaren Text besitzen, muss dieser auch noch angezeigt werden.

Jedes *SVGElement* besitzt eine so genannte *BoundingBox*. Die *BoundingBox* ist ein Rechteck welches das gesamte Element umschließt.
Innerhalb dieser *BoundingBox* kann jedes Element anders aussehen. Hier könnte sich z. B. eine Linie, ein Polygon oder eine Ellipse befinden.
All diese Elemente würden innerhalb dieses Rechtecks liegen. Bei einem *SVGRectangle* würde die *BoundingBox* mit dem Element übereinstimmen.
Da jedes Element anders aussehen kann, ist es beim Rendern eines Textes wichtig, anzugeben an welcher Stelle dieser angezeigt werden soll.
Der Bereich in dem der Text dargestellt werden kann, wird durch ein zweites Rechteck, die *TextBoundingBox* definiert. Innerhalb dieser *TextBoundingBox* kann der Text auch horizontal ausgerichtet werden.
Normalerweise wird diese Textfläche automatisch mittels eines eingestellten *Margins* berechnet.
Bei eigenen Elementen kann dies aber manuell an die eigenen Bedürfnisse gepasst werden, indem die Methode *updateTextBoundingBox()* überschrieben wird.

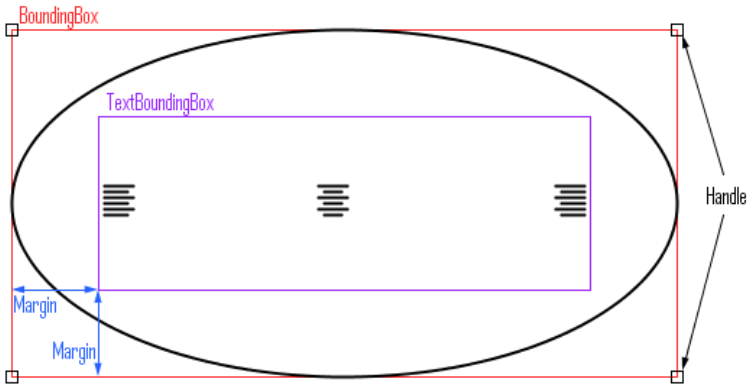


Abbildung 53: TextBoundingBox

Da das Darstellen der Texte für alle Elemente identisch ist, wurde dies direkt in die Oberklasse eingebaut.

```
SVGElement.renderText(int zoomFactor);
```

Grundsätzlich funktioniert das Rendern eines Strings ähnlich dem Zeichnen einer Linie oder eines anderen grafischen Primitives.

```
Graphics2D.drawString(AbstractCharacterIterator it, float x, float y);
```

Die Texte werden zum Zeichnen in einzelne Zeilen zerlegt. Aufgrund der *TextBoundingBox*, der Ausrichtung und der aktuell zu zeichnenden Zeile (bei mehrzeiligen Texten) kann bestimmt werden, wo der String gezeichnet werden soll.
Um die exakte Positionierung berechnen zu können, werden die exakten Abmessungen der einzelnen Zeilen benötigt. Die Maße eines Textes sind abhängig von Schriftart, Schriftgröße und Stil. Deshalb muss jede Zeile mit Hilfe eines *AttributedString* formatiert werden.

```
SVGElement.getAttributedString(String text, int zoomFactor);
```

Anhand des formatierten Textes ist es möglich einen *GlyphVector* zu erstellen. Dieser besitzt eine Liste von Glyphen welche die geometrische Repräsentation für die einzelnen Schriftzeichen des Textes enthalten.

```
Font f = (Font)AttributedString.getIterator().getAttribute(TextAttribute.FONT);
GlyphVector vec = f.createGlyphVector(Graphics2D.getFontRenderContext(), String text);
```

Erst durch diese Glyphen können die visuellen Abmessungen des Textes exakt bestimmt werden.

```
GlyphVector.getVisualBounds()
```

7.6. Polygone

Das Polygon wird durch die Klasse *SVGPolygone* repräsentiert. Diese Klasse wurde von der Klasse *SVGElement* abgeleitet und erweitert.
Um ein Element vom Typ Polygon erstellen und bearbeiten zu können, besitzt sie einen erweiterten Erstell- und Bearbeitungsmodus. Außerdem können Polygone offen oder geschlossen sein und die Enden mit Pfeilspitzen versehen werden.
Die grundlegenden Prinzipien sind bereits aus dem Kapitel *Polygon* bekannt. Auf den nächsten Seiten wird die Implementierung der bereits vorgestellten Konzepte betrachtet.

7.6.1. Erstellen eines neuen Polygons

Die meisten Elemente werden – nachdem das entsprechende Werkzeug ausgewählt wurde –auf der Leinwand bis zur gewünschten Größe aufgezogen.
Ein Polygon jedoch besteht aus mehreren Punkten. Beim Erstellen wird nur die erste Linie aufgezogen. Alle weiteren gewünschten Punkte werden mit Mausklicks hinzugefügt.

Beim ersten Drücken der Maustaste werden der Ursprungspunkt und der Endpunkt der ersten Linie angelegt.

```
public void handleMousePressed(MouseEvent e) {
  if(this.isInitialCreateMode()) {
    this.getHandles().set(0, e.getPoint());
    this.getHandles().set(1, e.getPoint());
    this.updateBoundingRect(this.getHandles());
  }
}
```

Beim Ziehen der Maus, wird der Endpunkt der Linie entsprechend der Mausposition verändert.

```
public void handleMouseDragged(MouseEvent e) {
  if(this.isInitialCreateMode()) {
    this.getHandles().set(1, e.getPoint());
    this.updateBoundingRect(this.getHandles());
  }
}
```

Beim Loslassen der Maustaste, wird der Endpunkt fixiert, ein neuer Punkt für die nächste Linie hinzugefügt und das Polygon wechselt in den erweiterten Erstellmodus.

```
public void handleMouseReleased(MouseEvent e) {
  if(this.isInitialCreateMode() && !this.getHandles().get(0).equals(e.getPoint())) {
    this.getHandles().set(1, e.getPoint());
    this.updateBoundingRect(this.getHandles());
    this.getHandles().add(new Point(e.getPoint()));
    this.setMode(Mode.ExtendedCreate);
  }
}
```

Befindet sich das Polygon im erweiterten Erstellmodus, wird beim Bewegen der Maus eine Linie vom zuletzt erstellten Punkt zur aktuellen Mausposition gezeichnet.

```
public void handleMouseMoved(MouseEvent e) {
  if(this.isExtendedCreateMode())
    this.getHandles().lastElement().setLocation(e.getPoint());
}
```

In diesem Modus, wird dem Polygon bei jedem weiteren Mausklick ein neuer Punkt hinzugefügt.
Eine Ausnahme hiervon wird gemacht falls der Anfangspunkt angeklickt wird. In diesem Fall wird der Erstellmodus beendet und das Polygon geschlossen.

```
public void handleMouseClicked(MouseEvent e) {
  int handle = -1;
  if(e.getClickCount()==1) {
    switch(this.getMode()) {
      case ExtendedCreate:
        handle = this.getHandleIndex(e.getPoint());
        if(handle == 0) {
          this.setClosedPolygone(true);
          this.getHandles().remove(this.getHandles().size()-1);
          this.setMode(Mode.Edit);
        } else if(handle==this.getHandles().size()-2) {
          this.getHandles().remove(this.getHandles().size()-1);
          this.setMode(Mode.Edit);
        } else {
          this.getHandles().add(new Point(e.getPoint()));
        }
        ActionManager.fireActionPerformedEvent(
          new ActionPerformedEvent(ActionPerformedEvent.Action.createHistoryPoint)
        );
        break;
    }
  }
}
```

Wird im erweiterten Erstellmodus ein Doppelklick registriert, wird das Polygon mit dem Punkt an dem geklickt wurde beendet. Geschieht dies am Ausgangspunkt, so wird das Polygon geschlossen.

```
public void handleMouseClicked(MouseEvent e) {
  int handle = -1;
  // If handle is double-clicked, toggle an arrow head at this side
  if(e.getClickCount()>=2) {
    switch(this.getMode()) {
      case ExtendedCreate:
        this.getHandles().remove(this.getHandles().size()-1);
        this.setMode(Mode.Edit);
        handle = this.getHandleIndex(e.getPoint());
        if(handle == 0) {
          this.setClosedPolygone(true);
          this.getHandles().remove(this.getHandles().size()-1);
        }
        break;
    }
  }
}
```

7.6.2. *Bearbeiten eines Polygons*

Um den normalen Bearbeitungsmodus zu betreten um das Polygon verschieben oder verzerren zu können, muss dieses nur selektiert werden.

Um in den erweiterten Bearbeitungsmodus zu gelangen, wird ein Doppelklick auf das selektierte Polygon ausgeführt.

```
public void handleMouseClicked(MouseEvent e) {
  int handle = -1;
  if(e.getClickCount()>=2) {
    switch(this.getMode()) {
      case Edit:
        if (this.hit(e.getPoint()))
          this.setMode(Mode.ExtendedEdit);
        break;
    }
  }
}
```

Beim Wechsel in den erweiterten Bearbeitungsmodus werden neue Bearbeitungspunkte in der Mitte jeder Kante des Polygons eingefügt.
Diese Handles werden anders dargestellt und können dazu benutzt werden neue Eckpunkte zu erstellen bzw. eine Kante zu verschieben.

Wird einer der ursprünglich erstellten Punkte verschoben, werden die neu eingefügten Punkte in der Mitte der angrenzenden Linien mit verschoben.

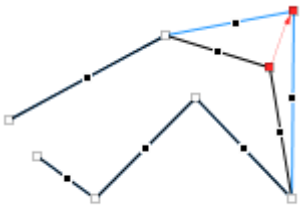


Abbildung 54: Polygonecke verschieben

Wird einer der schwarzen Punkte in der Mitte einer Kante verschoben, verschiebt man entweder die ganze Kante, oder erstellt – bei gedrückter Strg-Taste – einen neuen Eckpunkt.

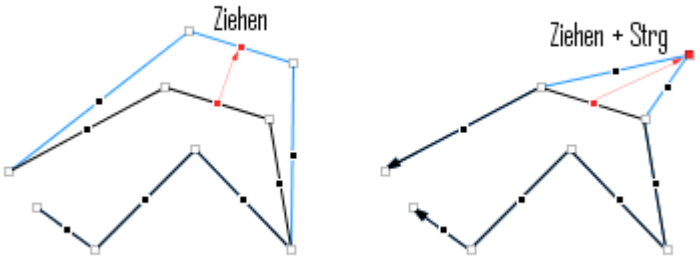


Abbildung 55: Polygonkante bearbeiten

```
public void moveHandle(int handleIndex, Point p, MouseEvent e) {
  if(this.isExtendedEditMode()) {
    this.getHandles().get(handleIndex).translate(p.x, p.y);
    this.updateBoundingRect(this.getHandles());
    if(handleIndex%2==0)
      this.calculateSurroundingHandles(handleIndex);
    else if(handleIndex%2==1 && !e.isControlDown())
      this.calculateLineMovement(handleIndex, p);
  }
}
```

Mit einem Klick auf einen Endpunkt, kann eine Pfeilspitze angebracht werden.

```
public void handleMouseClicked(MouseEvent e) {
  int handle = -1;
  if(e.getClickCount()==1) {
    switch(this.getMode()) {
      case ExtendedEdit:
        if (this.hit(e.getPoint())) {
          handle = this.getHandleIndex(e.getPoint());
          if(handle != -1 && !this.isClosedPolygone()) {
            if (handle == 0)
              this.setHasArrowAtEnd(!this.hasArrowAtEnd());
            else if (handle == this.getHandles().size()-1)
              this.setHasArrowAtFront(!this.hasArrowAtFront());
          }
          ActionManager.fireActionPerformedEvent(
            new ActionPerformedEvent(ActionPerformedEvent.Action.createHistoryPoint)
          );
        }
        break;
    }
  }
}
```

Der erweiterte Bearbeitungsmodus wird entweder mit einem Klick außerhalb des Polygons oder mit einem Doppelklick innerhalb des Polygons beendet.

7.7. Gruppe

Wie bereits bekannt ist, können mehrere Elemente zu einer Gruppe zusammengefasst werden. Solch eine Gruppe wird durch ein Element der Klasse *SVGComposite* repräsentiert und kann natürlich auch wieder aufgelöst werden.

Ein *SVGComposite* ist wegen des speziellen Bearbeitungsmodus an mehreren Stellen im Programm fix integriert.

Die grundlegenden Prinzipien sind bereits aus dem Kapitel *Gruppierung* bekannt. In diesem Kapitel wird die Implementierung der bereits vorgestellten Konzepte betrachtet.

7.7.1. Erstellen einer neuen Gruppe

Selektierte Elemente werden zu einer neuen Gruppe formiert, indem das entsprechende Kommando aus der Menüleiste ausgewählt wird. Im *MenuBarManager* wird daraufhin eine *group*-Aktion gefeuert.

```
ActionManager.fireActionPerformedEvent(
                    new ActionPerformedEvent(ActionPerformedEvent.Action.group)
);
```

Diese Aktion wird vom *ElementManager* entgegengenommen und verarbeitet.

Die selektierten Elemente werden in ein *SVGComposite* zusammengefasst und aus der Liste der Elemente entfernt.

```
public void actionPerformed(ActionPerformedEvent e) {
 switch (e.getAction()) {
  case group:
   this.handleGroup();
   break;
 }
}

private void handleGroup() {
 if (this.getSelectedElements().size() > 1) {
  SVGComposite comp = new SVGComposite();
  for (SVGElement elem : this.getSelectedElements()) {
   comp.addElement(elem);
   this.getAllElements().remove(elem);
  }
  this.addElement(comp);
 }
}
```

7.7.2. Bearbeiten einer bestehenden Gruppe

Betreten des Bearbeitungsmodus

Um eine Gruppe bearbeiten zu können, ist es nicht nötig diese aufzulösen und nach den Änderungen neu zu gruppieren.

Das Starten des Bearbeitungsvorganges ist im Selektions-Werkzeug integriert und kann mit einem Doppelklick auf eine bestehende Gruppe durchgeführt werden.

```
ActionManager.fireActionPerformedEvent(
                new ActionPerformedEvent(
                                ActionPerformedEvent.Action.enterGroup,
                                (SVGComposite) this.activeElement
                )
);
```

Während eine Gruppe bearbeitet wird, sind nur die Elemente dieser Gruppe in ihrem Originalzustand sichtbar. Damit dies möglich ist, empfängt der *ElementManager* die *enterGroup*-Aktion und schaltet sich selbst und die betroffene Gruppe in den Bearbeitungsmodus.

```
public void actionPerformed(ActionPerformedEvent e) {
 switch (e.getAction()) {
  case enterGroup:
   this.startEditComposite((SVGComposite)e.getValue());
   break;
 }
}

public void startEditComposite(SVGComposite comp) {
 this.setInCompositeEditMode(true);
 comp.setInCompositeEditMode(true);
 this.composites.addFirst(comp);
}
```

Da es auch ineinander verschachtelte Gruppen geben kann, muss es möglich sein untergeordnete Gruppen zu bearbeiten. Hierfür verwaltet der *ElementManager* eine Bearbeitungsliste für Gruppierungen. Jedes Mal wenn die Methode *startEditComposite()* aufgerufen wird, wird die übergebene Gruppe an den Anfang dieser Liste gestellt.

Die Gruppe die gerade bearbeitet wird und sichtbar ist, befindet sich in dieser Liste immer ganz oben.

Verlassen des Bearbeitungsmodus

Der Bearbeitungsmodus kann auf zwei Weisen verlassen werden: entweder über die Menüleiste oder mit einem Doppelklick auf eine freie Fläche der Leinwand.
Beides hat den gleichen Effekt und löst eine *leaveGroup*-Aktion aus.

```
ActionManager.fireActionPerformedEvent(
                    new ActionPerformedEvent(ActionPerformedEvent.Action.leaveGroup)
);
```

Beim Verlassen einer Gruppe wird wiederum der *ElementManager* benachrichtigt. Dieser entfernt die gerade bearbeitete Gruppe aus der Bearbeitungsliste.
Ist diese Liste leer wird der Bearbeitungsmodus beendet. Ist die Liste noch nicht leer, werden die Elemente der nächsten Gruppe eingeblendet und zur Bearbeitung freigegeben.
Besitzt eine Gruppe beim Verlassen nur noch ein Element, wird die Gruppe automatisch entpackt.

```
public void actionPerformed(ActionPerformedEvent e) {
 switch (e.getAction()) {
  case leaveGroup:
   this.endEditComposite();
   break;
 }
}

public void endEditComposite() {
 if(this.isInCompositeEditMode() && !this.composites.isEmpty()) {
  SVGComposite cmp = this.composites.removeFirst();
  cmp.setInCompositeEditMode(false);
  if(this.composites.isEmpty()) {
   this.setInCompositeEditMode(false);
   if(cmp.getElements().size()==1)
    ActionManager.fireActionPerformedEvent(
              new ActionPerformedEvent(ActionPerformedEvent.Action.ungroup)
     );
  }
 }
}
```

7.7.3. Verzerren einer bestehenden Gruppe

Eine Gruppe behält immer die originalen Elemente welche ursprünglich gruppiert wurden und deren *BoundingBox*.

Wird die Gruppe verzerrt oder verschoben, so werden nicht die originalen Elemente verändert, sondern es ändert sich nur die angezeigte *BoundingBox*.

Mithilfe des Verschiebungs- und Skalierungsvektors zwischen den beiden umschließenden Rechtecken, werden – ausgehend von den originalen Elementen – die angezeigten Elemente berechnet.

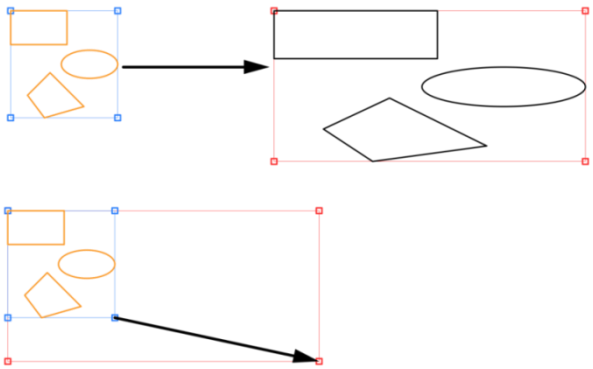


Abbildung 56: SVGComposite - Verzerren/Verschieben

7.7.4. Auflösen einer bestehenden Gruppe

Eine selektierte Gruppe kann aufgelöst werden, indem das entsprechende Kommando aus der Menüleiste ausgewählt wird. Im *MenuBarManager* wird daraufhin eine *ungroup*-Aktion gefeuert.

```
ActionManager.fireActionPerformedEvent(
                    new ActionPerformedEvent(ActionPerformedEvent.Action.ungroup)
);
```

Der *ElementManager* nimmt diese Aktion entgegen, fügt die im *SVGComposite* enthaltenen Elemente der globalen Liste hinzu und entfernt die Gruppe aus dieser.

```
public void actionPerformed(ActionPerformedEvent e) {
  switch (e.getAction()) {
    case ungroup:
      this.handleUngroup();
      break;
  }
}
private void handleUngroup() {
  SVGComposite comp = (SVGComposite) this.getSelectedElements().get(0);
  this.getAllElements().remove(comp);
  for (SVGElement elem : comp.getElements())
    this.getAllElements().add(elem);
  }
```

7.8. Pfeilspitze

Die Berechnung der Pfeilspitzen zu implementieren, war eine heikle Angelegenheit. Die Eckpunkte der Pfeilspitze müssen anhand des Vektors der Linie, der gewünschten Länge der Spitze und des Öffnungswinkels berechnet werden. Der Vektor der Linie lässt sich anhand der beiden Endpunkte einfach berechnen. Für den Öffnungswinkel gab es eine Vorgabe meines Betreuers von 30°. Auch für die Länge der Pfeilspitzen gab es eine Vorlage – mit Pfeilen verschiedener Liniendicken – welche vermessen wurde. Diese Messung ergab eine Länge von 5,5 Pixeln bei Haarlinien und 9,5 Pixel mal der Liniendicke als allgemeine Richtlinie.

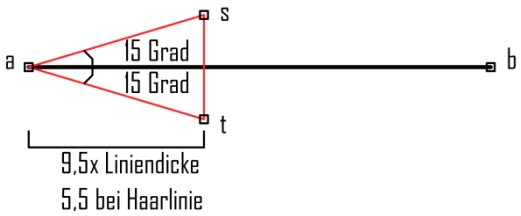


Abbildung 57: Pfeilspitzen Maße

Die Punkte *s* und *t* können mit Hilfe folgender Berechnung ermittelt werden:

$$t = 15°$$

$$d = 9{,}5 \times \text{Liniendicke}$$

$$l = \sqrt{(x1 - x0)^2 + (y1 - y0^2}$$

$$n = \tan\left(\frac{t \times \pi}{180}\right)$$

$$x = \text{x1} - \frac{(\text{x1} - \text{x0}) - n(y1 - y0)}{\sqrt{1 + n^2}} \times \frac{d}{l}$$

$$y = \text{y1} - \frac{(\text{y1} - \text{y0}) + n(x1 - x0)}{\sqrt{1 + n^2}} \times \frac{d}{l}$$

Für die Implementierung wurde die Berechnung von *x* und *y* noch etwas vereinfacht:

$$s.x = \text{x1} - [(\text{x1} - \text{x0}) \times \cos(n) - (y1 - y0) \times \sin(n)] \times \frac{d}{l}$$

$$s.y = \text{y1} - [(\text{y1} - \text{y0}) \times \cos(n) + (x1 - x0) \times \sin(n)] \times \frac{d}{l}$$

$$t.x = \text{x1} - \left[(\text{x1} - \text{x0}) \times \cos\left(n \times (-1)\right) - (y1 - y0) \times \sin\left(n \times (-1)\right)\right] \times \frac{d}{l}$$

$$t.y = \text{y1} - \left[(\text{y1} - \text{y0}) \times \cos\left(n \times (-1)\right) + (x1 - x0) \times \sin\left(n \times (-1)\right)\right] \times \frac{d}{l}$$

Damit die Linie am Ende nicht über die Pfeilspitze ragt, muss diese beim Zeichnen noch verkürzt werden

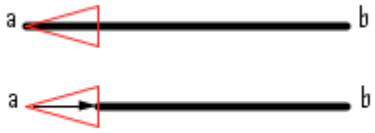


Abbildung 58: Pfeillinie verkürzen

7.9. Zwischenablage

Die Zwischenablage bietet eine komfortable Schnittstelle um Daten zwischen verschiedenen Programmen zu transferieren.
Die Daten, welche in die Zwischenablage kopiert wurden, können dort in verschiedenen Formaten abgelegt werden.
Auf den folgenden Seiten wird das Lesen aus der und Schreiben in die Zwischenablage mit Daten unterschiedlichen Formats gezeigt.

7.9.1. Aufbau und Implementierung

Um in Java auf die Zwischenablage zugreifen zu können, muss man sich ein *Clipboard* Objekt holen.

```
Clipboard clip = Toolkit.getDefaultToolkit().getSystemClipboard();
```

Der Inhalt der Zwischenablage wird durch ein Objekt vom Typ *Transferable* repräsentiert.
Da die kopierten Elemente in mehr als einem Format in der Zwischenablage platziert werden sollen, musste eine eigene *Transferable*-Klasse entwickelt werden, welche mit einem *String* (SVG-Code), einem *BufferedImage* (PNG) und einer *Liste von Dateien* umgehen kann.

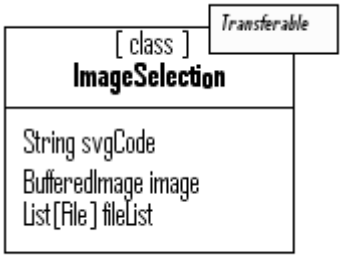


Abbildung 59: ImageSelection

Dateien könnten zum Beispiel temporär erzeugte Bilder oder PDF-Dokumente sein.

Werden Elemente selektiert und in die Zwischenablage kopiert, erzeugt die Software den entsprechenden SVG-Code, eine Rastergrafik und ein PDF-Dokument.
Mit Hilfe einer Klasse vom Typ *ImageSelection* können alle drei Formate gleichzeitig in die Zwischenablage gestellt werden.

```
BufferedImage img = new BufferedImage(<BREITE>, <HÖHE>,BufferedImage.TYPE_INT_ARGB);
Graphics2D g = (Graphics2D)img.createGraphics();
for (SVGElement elem : this.getSelectedElements())
  elem.paint(g);

String svg = this.generateSVGString(this.getSelectedElements());

// Metadaten in PNG-Bild schreiben
byte[] bArr = HelperFunction.writeCustomData(img, "SVGCode", svg);
File f = File.createTempFile("tmpTestPNG", ".png");
FileOutputStream fos = new FileOutputStream(f);
fos.write(bArr);
fos.close();
// Erstellen der ImageSelection (BufferedImage, Array<File>, String)
ImageSelection trans = new ImageSelection(null, f, svg);

// Zwischenablage holen und Transferable setzen
Clipboard c = Toolkit.getDefaultToolkit().getSystemClipboard();
c.setContents(trans, null);
```

In der aktuellen Implementierung werden nur der SVG-Code als String und ein PNG-Bild mit Metainformationen als Datei übergeben.
Durch die Repräsentation als SVG-Code, kann der kopierte Inhalt in der Grafiksoftware wieder eingefügt und können somit die Elemente dupliziert werden.
Als temporäre Datei kann die Rastergrafik aus der Zwischenablage in diverse Textverarbeitungsprogramme und sogar in den Windows Explorer eingefügt werden.

Das Lesen der Zwischenablage funktioniert natürlich genauso über die *Transferable*-Klasse, welche vom *Clipboard* geliefert wird.

```
Transferable transfer = clip.getContents();
```

Da die Grafiksoftware – wie bereits erwähnt – beim Einfügen auf den SVG-Code angewiesen ist, liest sie präferiert Daten vom Typ String.

```
if(transfer.isDataFlavorSupported(DataFlavor.stringFlavor)) {
   String data = (String) transfer.getTransferData( DataFlavor.stringFlavor );
```

Wird kein String gefunden, wird nach PNG/PDF Dateien gesucht, welche die benötigten Metadaten enthalten könnten.
Wurde SVG-Code entdeckt, kann dieser an die *Import*-Klasse übergeben werden, welche sich um die Validierung und die korrekte Weiterverarbeitung kümmert.

7.9.2. Probleme

Wie – von zukünftigen Nutzern – gewünscht, wurde nach einer Möglichkeit gesucht, die erstellten Grafiken aus einem gängigen Textverarbeitungsprogramm heraus über die Zwischenablage in das Grafikprogramm zu überführen, um daran weiterarbeiten zu können.
Dies gestaltete sich allerdings schwieriger als erwartet, da kaum ein Textverarbeitungsprogramm mit SVG-Grafiken arbeiten kann.
Entweder sie können generell keine Vektorgrafiken darstellen oder sie beherrschen nur eigene, proprietäre Formate welche nicht vorgesehen waren.

Die Lösung des Problems schien es zu sein, eine Rastergrafik zu exportieren und diese mit verstecktem SVG-Code zu versehen.

Da es anfangs nicht gelang, Rastergrafiken mit Metainformationen abzuspeichern, wurde die Export Funktionalität um das PDF-Format erweitert.
Mit Hilfe der freien API *iText* war es ein Leichtes, den SVG-Code einzubetten. Auch das Kopieren über die Zwischenablage nach Microsoft Word war kein Problem.

Nur das Zurückkopieren war – trotz vorhandener Metadaten – nicht mehr möglich.
Nach eingehender Analyse der Zwischenablage konnte festgestellt werden, dass Word das Bild in vielen Formaten liefert und in manchen auch der eingebettete SVG-Code steckt. Über das Java Clipboard war es aber nicht möglich diesen auszulesen, da Java offensichtlich keinen Zugriff auf die gesamte Zwischenablage hat.

Nach vielen Versuchen gelang es letztendlich doch noch, Rastergrafiken inklusive Metadaten abzuspeichern. Beim Testen der Zwischenablage mit diesen Rastergrafiken, stellte sich aber heraus, dass gängige Textverarbeitungsprogramme die Metadaten verwerfen.

Da die meisten zukünftigen Nutzer unter Microsoft Windows mit Word arbeiten, wird auch hier der Schwerpunkt auf MS Word gelegt.
Word unterstützt Vektorgrafiken in den proprietären Formaten

- Windows Metafile (WMF) und
- Windows Enhanced Metafile (EMF).

Die Idee war nun, auch eines dieser Formate in das Repertoire der Software aufzunehmen.
Der Versuch mit einer bestehenden EMF-Datei welche in MS Word eingefügt und von dort aus in die Zwischenablage kopiert wurde, schien vielversprechend.
Zumindest dahingehend, dass eine EMF-Datei vorhanden war.
Auch diesmal scheiterte der Import der Grafik jedoch wieder am Java Clipboard welches keines der brauchbaren Formate lieferte.

Möglicherweise könnten C/C++ Bibliotheken über das Java Native Interface (JNI) angebunden werden um die Kommunikation mit der Zwischenablage darüber abzuwickeln. Bislang jedoch wurde für dieses Problem keine Lösung entwickelt und der Datenaustausch mit anderen Programmen funktioniert vorerst über PNG-Bilder inklusive Metainformationen.

7.10. Grafiken importieren

Das Importieren von gespeicherten Grafiken übernimmt die Klasse *SVGImporter*.
Diese Klasse nimmt SVG-/PDF- oder PNG-Dateien mit vorhandenen Metainformationen entgegen.

```
public static Vector<SVGElement> Import(File file);
```

Falls ein entsprechender SVG-Code in XML Format gefunden wird, können die vorhandenen Elemente daraus extrahiert werden.

```
DocumentBuilder db = DocumentBuilderFactory.newInstance().newDocumentBuilder();
Document doc = db.parse(file);
doc.getDocumentElement().normalize();
NodeList nList = doc.getChildNodes();
```

Die extrahierten Elemente werden dann einzeln überprüft und je nach Typ geparst.

```
private static SVGElement importSVGElement(Element elem) throws XPathExpressionException {
 SVGElement impElem = null;
 String g_id = elem.getAttribute("id");

  if (g_id.equalsIgnoreCase("SVGComposite")) {
    SVGComposite comp = new SVGComposite();
    comp.setElements(importElements(elem.getChildNodes()));
    if(!comp.getElements().isEmpty())
      impElem = comp;
  } else if(g_id.equalsIgnoreCase("SVGLine")) {
    impElem = importSVGLine(elem);
  } else if(g_id.equalsIgnoreCase("SVGRoundSideRectangle")) {
    impElem = importSVGRoundSidedRectangle(elem);
  } else if(g_id.equalsIgnoreCase("SVGPolygone")) {
    impElem = importSVGPolygone(elem);
  } else if(g_id.equalsIgnoreCase("SVGText")) {
    impElem = importSVGText(elem);
  } else if(g_id.equalsIgnoreCase("SVGRectangle")) {
    impElem = importSVGRectangle(elem);
  } else if(g_id.equalsIgnoreCase("SVGEllipse")) {
    impElem = importSVGEllipse(elem);
  } else if(g_id.equalsIgnoreCase("SVGTriangle")) {
    impElem = importSVGTriangle(elem);
  }
  return impElem;
}
```

Für nahezu jedes *SVGElement* gibt es eine eigene Methode, um aus dem XML-Knoten dieses Element zu erzeugen.

```
private static SVGElement importSVGLine(Element element);
private static SVGElement importSVGTriangle(Element elem);
private static SVGElement importSVGPolygone(Element element);
private static SVGElement importSVGRectangle(Element element);
private static SVGElement importSVGRoundSidedRectangle(Element element);
private static SVGElement importSVGEllipse(Element element);
private static SVGElement importSVGText(Element element);
```

Da alle Elemente ein paar identische Eigenschaften – wie zum Beispiel die Linienfarbe – besitzen, verwenden sie die gleichen Methoden um diese Attribute in einem Knoten identifizieren und in ein brauchbares Format umzuwandeln. Die *getPathPointsForAttributeName()*-Methodeliefert für einen Knoten mit einem Pfad-Attribut[8], sämtliche Punkte und auch deren entsprechende Bedeutung.

```
private static void setArrow(SVGArrow elem, Point start, Point end, Node line);
private static boolean isRoundedRect(Node rect);
private static Point getPointForAttributeNames(String xAttribute, String yAttribute, Node node);
private static int getWidth(Node node);
private static int getHeight(Node node);
private static int getLineStrength(Node node);
private static Color getRGBColorForAttributeName(String attribute, Node node);
private static int getIntValueForAttributeName(String attribute, Node node);
private static String getStringValueForAttributeName(String attribute, Node node);

private static Vector<PathPoint> getPathPointsForAttributeName(String attribute, Node node);
private static class PathPoint {
 public enum Type {MoveToAbsolute, MoveToRelative, LineToAbsolute, LineToRelative,
                   CurveToAbsolute, CurveToRelative, SmoothCurveToAbsolute, SmoothCurveToRelative,
                   QuadCurveToAbsolute, QuadCurveToRelative, SmoothQuadCurveToAbsolute,
                   SmoothQuadCurveToRelative, ArcCurveToAbsolute, ArcCurveToRelative,
                   ClosePath
 };

 public Type type = null;
 public Point[] points = null;
 public PathPoint(Type t, Point[] pts) {
  this.type = t;
  this.points = pts;
 }
}
```

[8] SVG-Element bestehend aus einer Menge an Punkten welche einen Pfad bilden und damit die Form eines Elementes darstellen.

8. Erweiterung

Im folgenden Kapitel soll gezeigt werden, wie die Software adaptiert werden kann.
Im Detail wird behandelt wie ein neues grafisches Grundelement samt Import/Export und neue Objekteigenschaften/-aktionen samt ToolBar-Erweiterung implementiert werden können.

8.1. Neues Element – Dreieck

Jedes grafische Element leitet sich von dem Basis-Typ *SVGElement* ab und implementiert die wichtigsten Methoden um erstellt, verändert und dargestellt werden zu können.

```
public class SVGTriangle extends SVGElement {
  public SVGTriangle();
  public SVGElement getInstance();
  public void paint(Graphics g);
  public void paint(Graphics g, int zoomFactor);
  public boolean hit(Rectangle rect);
  public Icon getIcon();
  public Point getTranslationPointForHandle(int handleIndex);
  public void moveHandle(int handleIndex, Point p, MouseEvent e);
  public boolean validate();
  public String exportToSVG();
  public void handleMousePressed(MouseEvent e);
  public void handleMouseDragged(MouseEvent e);
  public void handleMouseReleased(MouseEvent e);
  public void handleMouseClicked(MouseEvent e);
}
```

8.1.1. Konstruktor

Der Konstruktor ist kurz und übersichtlich gehalten. Hier werden lediglich die benötigte Anzahl an Bearbeitungspunkten hinzugefügt und einige Standardoptionen definiert welche die Möglichkeiten des Elementes spezifizieren.

```
public SVGTriangle() {
  this.getHandles().add(new Point(-1,-1));
  this.getHandles().add(new Point(-1,-1));
  this.getHandles().add(new Point(-1,-1));
  this.setFillcolorSetable(true);
  this.setTextSetable(false);
}
```

8.1.2. Neue Instanz

Die Methode *getInstance()* wird beim Duplizieren aufgerufen und bietet die Möglichkeit elementspezifische Eigenschaften zu setzen (z.B.: Pfeilspitze).

```
@Override
public SVGElement getInstance() {
 return new SVGTriangle();
}
```

8.1.3. Rendern

Wie bereits erwähnt weiß jedes Element für sich, wie es sich darzustellen hat.

In der *paint()*-Methode kann man unterschiedliche Zustände des Elementes, Zoom-Modi usw. berücksichtigen. Auch das Polygon unterscheidet hier zwischen seinen Bearbeitungsmodi.

```
@Override
public void paint(Graphics g) {
 this.paint(g, 1);
}

@Override
public void paint(Graphics g, int zoomFactor) {
 this.setGraphics((Graphics2D) g);
 int[] xa = new int[3];
 int[] ya = new int[3];
 for(int i = 0; i<this.getHandles().size(); i++) {
  Point p = this.calcZoom(this.getHandles().get(i), zoomFactor);
  xa[i] = p.x;
  ya[i] = p.y;
 }
 this.getGraphics().setStroke(getStroke(zoomFactor));
 if(this.isFillcolorSetable()) {
  this.getGraphics().setColor(this.getFillColor());
  this.getGraphics().fillPolygon(xa, ya, this.getHandles().size());
 }
 if(this.getLineStrength()!=-1) {
  this.getGraphics().setColor(this.getLineColor());
  this.getGraphics().drawPolygon(xa, ya, this.getHandles().size());
 }
}
```

8.1.4. Hit-Test

Alle individuell implementierten Elemente können unterschiedliche äußere Formen annehmen. Deshalb ist es wichtig, dass die Überprüfung – ob ein Element bei einem Mausklick getroffen wurde – für jedes Element einzeln gemacht wird. Hierfür ist es notwendig den Hit-Test individuell anzupassen.

```
@Override
public boolean hit(Rectangle rect) {
 int[] xa = new int[3];
 int[] ya = new int[3];
 for(int i = 0; i<this.getHandles().size(); i++) {
  xa[i] = this.getHandles().get(i).x;
  ya[i] = this.getHandles().get(i).y;
 }
 Polygon poly = new Polygon(xa,ya,this.getHandles().size());
 return poly.intersects(rect);
}
```

8.1.5. Icon

Um ein Objekt zeichnen zu können, muss zuerst das entsprechende Werkzeug ausgewählt werden. Um hier Klarheit zu schaffen, erzeugt jedes *SVGElement* sein eigenes Icon welches angezeigt werden kann. Zusätzlich kann ein Text hinterlegt werden, welcher als ToolTip-Text[9] genutzt wird.

```
@Override
public Icon getIcon() {
 return new Icon() {
  public int getIconHeight() { return 16; }
  public int getIconWidth()  { return 16; }

  @Override
  public void paintIcon(Component c, Graphics g, int x, int y) {
   Graphics2D g2 = (Graphics2D) g;
   g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);
   g2.setColor(Color.BLACK);
   g2.setStroke(new BasicStroke(1, BasicStroke.CAP_BUTT, BasicStroke.JOIN_MITER));
   int[] xa = {x + 8, x + 3,  x + 13};
   int[] ya = {y + 2, y + 13, y + 13};
   g2.drawPolygon(xa, ya, 3);
  }
 };
}
```

[9] Wird eingeblendet wenn sich der Mauszeiger über einer bestimmten Position befindet.

8.1.6. Bearbeiten – Handle bewegen

Die Elemente werden intern mit einer Menge von Punkten beschrieben. Um sie zu verändern, verschiebt man diese Eckpunkte nach Belieben.

Hier kann man individuelles Verhalten implementieren falls sich durch das Verschieben eines Eckpunktes auch andere Punkte verändern müssen.

```
@Override
public void moveHandle(int handleIndex, Point p, MouseEvent e) {
 if (this.isNormalMode()) return;

 this.getHandles().get(handleIndex).translate(p.x, p.y);
 this.updateBoundingRect(this.getHandles());
 System.out.println(exportToSVG());
}
```

8.1.7. Validieren des Elementes

Um zu gewährleisten, dass die gespeicherten Elemente richtig dargestellt werden können, müssen diese nach dem Erstellen und Bearbeiten validiert werden.

Ist das Element nach dem Erstellen fehlerhaft, wird es verworfen.

Ein Element wäre in einem fehlerhaften Zustand, wenn es nicht mehr korrekt dargestellt werden könnte. Dies wäre z. B. der fall wenn bei einem Rechteck zwei der vier Eckpunkte dieselben Koordinaten hätten.

Ist das Element nach dem Bearbeiten fehlerhaft, werden die Änderungen verworfen und der ursprüngliche Zustand wiederhergestellt.

```
@Override
public boolean validate() {
 for(int i = 0; i<this.getHandles().size(); i++) {
  if(this.getHandles().get(i).distance(this.getHandles().get((i+1)%3)) == 0 ||
    this.getHandles().get(i).distance(this.getHandles().get((i+2)%3)) == 0)
   return false;
 }
 return true;
}
```

8.1.8. *Interaktion mit der Maus*

In diesem Beispiel wird das Maus-Klick Ereignis nicht verwendet. Bei den meisten Elementen kann der Nutzer mit einem Doppelklick in den Text-Bearbeitungsmodus gelangen welcher beim Dreieck jedoch nicht unterstützt wird.

```
@Override
public void handleMouseClicked(MouseEvent e) { }
```

Meistens beginnt das Erstellen mit dem Drücken der Maustaste im initialen Erstellmodus. Hier werden die ersten Eigenschaften gesetzt.

```
@Override
public void handleMousePressed(MouseEvent e) {
 if(!this.isInitialCreateMode()) return;

 this.getHandles().set(0, e.getPoint());
 this.getHandles().set(1, e.getPoint());
 this.getHandles().set(2, e.getPoint());
 this.updateBoundingRect(this.getHandles());
}
```

Durch das Ziehen der Maus bei gedrückter Maustaste wird das Dreieck aufgespannt.

```
@Override
public void handleMouseDragged(MouseEvent e) {
 if(!this.isInitialCreateMode()) return;

 this.getHandles().set(1, e.getPoint());
 Point p2 = (Point)e.getPoint().clone();
 p2.x -= (e.getPoint().x - this.getHandles().get(0).x)*2;
 this.getHandles().set(2, p2);
 this.updateBoundingRect(this.getHandles());
}
```

Der Erstellmodus wird beendet, sobald die Maustaste losgelassen wurde. Gleichzeitig wechselt das Element automatisch in den Bearbeitungsmodus.

```
@Override
public void handleMouseReleased(MouseEvent e) {
 if(!this.isInitialCreateMode()) return;

 this.updateBoundingRect(this.getHandles());
 this.setMode(Mode.Edit);
}
```

8.1.9. Exportieren des Elementes

Jedes abgeleitete *SVGElement* muss die Methode *exportToSVG()* überschreiben um die Möglichkeit des Speicherns zur Verfügung zu stellen.

Generiert wird W3C konformer SVG Code welcher auch von anderen Programmen interpretiert werden kann.

```
@Override
public String exportToSVG() {
 Vector<Point> elems = this.getHandles();
 StringBuilder sb = new StringBuilder();
 Color col = this.getLineColor();
 sb.append("\t");
 sb.append("<g id=\"" + this.getClass().getSimpleName() + "\">\n");
 sb.append("\t\t");
 sb.append("<path ");
 sb.append("d=\"M");
 sb.append(elems.firstElement().x + "," + elems.firstElement().y);
 for(int i = 1; i<elems.size(); i++)
  sb.append("L" + elems.get(i).x + "," + elems.get(i).y);
 sb.append("Z");

 sb.append("\" ");
 sb.append("stroke=\"rgb(" + col.getRed() + ","
                   + col.getGreen() + ","
                   + col.getBlue() + ")\" ");
 Color fc = this.getFillColor();
 if (fc.getAlpha() != 0)
  sb.append("fill=\"rgb(" + fc.getRed() + ","
                   + fc.getGreen() + ","
                   + fc.getBlue() + ")\" ");
 else
  sb.append("fill=\"none\" ");
 sb.append("stroke-width=\"" + this.getExportLineStrength() + "\"");
 sb.append(" />");
 sb.append("\n");
 sb.append("\t");
 sb.append("</g>\n");

 return sb.toString();
}
```

8.1.10. Erweitern der Werkzeugleiste

Damit das neue Element auch verwendet werden kann, benötigt es natürlich noch einen neuen Eintrag in der Werkzeugleiste. Wie einfach dies zu bewerkstelligen ist verdeutlichen die folgenden drei Schritte.

Neben dem Icon sind für die Werkzeuge auch ToolTip-Texte vorhanden welche aus einer Ressourcen-Datei gelesen werden.
Dazu wird die Datei *„svgEditorStrings_de.properties“* um folgenden Eintrag erweitet:

```
drawTriangle=Dreieck
```

Nachdem der Eintrag existiert, muss noch ein Wert in der *Reader.StringProperty* Enumeration angelegt werden um das Lesen zu vereinfachen:

```
public static enum StringProperty implements ConfigValue {
/* ... */,
drawTriangle;
}
```

Im dritten Schritt fehlt nur noch ein neuer Eintrag in der Werkzeugleiste. Hierzu wird die *init()* Methode im *ToolBarManager* wie folgt erweitert:

```
private void init() {
 /* ... */
 // Triangle
 drawBut = new ToolBarDrawingButton(
                        Reader.getPropertyValueAsString(Reader.StringProperty.drawTriangle),
                        this, this.optionManager,
                        new SVGTriangle()
            );
 this.toolBar.add(drawBut);
 /* ... */
}
```

Nach diesen drei simplen Schritten kann beim nächsten Start des Programms das Ergebnis begutachtet werden:

Abbildung 60: Werkzeugleiste mit Dreieck

8.1.11. Importieren

Nachdem das Dreieck gezeichnet und gespeichert werden kann, fehlt nur noch das Laden von Grafiken welche dieses neue Element enthalten. Hierzu muss die Klasse *SVGImporter* erweitert werden. Als Erstes wird dafür gesorgt, dass die *Import*-Klasse das Dreieck erkennt und die entsprechende Parsing-Methode aufruft.

```
private static SVGElement importSVGElement(Element elem) throws XPathExpressionException {
  if (/* ... */) {
  /* ... */
  } else if(g_id.equalsIgnoreCase("SVGTriangle")) {
    impElem = importSVGTriangle(elem);
  }
  return impElem;
}
```

Da das Dreieck in SVG als Pfad gespeichert wird, muss dieser Pfad gesucht und interpretiert werden um die Eckpunkte zu erhalten.

Für die weiteren Attribute wie Linienstärke, Linienfarbe und Füllfarbe kommen die Standardfunktionen des *SVGImporter*s zum Einsatz.

```
private static SVGElement importSVGTriangle(Element elem) {
  SVGTriangle tri = new SVGTriangle();
  Node n = null;
  /* Search for path node */
  for(int i = 0; i<elem.getChildNodes().getLength(); i++) {
    n = elem.getChildNodes().item(i);
    if(n.getNodeName().equalsIgnoreCase("path")) {  i++;   break;  }
  }
  if(n==null)     return tri;

  Vector<Point> pts = new Vector<Point>();
  /* if path node has been found, parse the points and generate Point-List */
  for(PathPoint pp : SVGImporter.getPathPointsForAttributeName("d", n)) {
    switch(pp.type) {
      case MoveToAbsolute:    pts.add(pp.points[0]);    break;
      case LineToAbsolute:    pts.add(pp.points[0]);    break;
    }
  }
  tri.setHandles(pts);   // Set the points as handles of the triangle

  /* Set further attributes */
  tri.setLineStrength(SVGImporter.getLineStrength(n));
  tri.setLineColor(SVGImporter.getRGBColorForAttributeName("stroke", n));
  tri.setFillColor(SVGImporter.getRGBColorForAttributeName("fill", n));
  tri.setMode(SVGElement.Mode.Normal);
  return tri;
}
```

Dies war der letzte notwendige Schritt, um das neue Element vollständig nutzen zu können.

8.2. Neue Aktion – Magnetisches Raster anzeigen

Um zu zeigen wie einfach es ist neue Aktionen hinzuzufügen, wird die Menüleiste auf den nächsten Seiten um einen Eintrag ergänzt, welcher es erlauben soll, das magnetische Raster ein- und auszublenden.
Diese Funktionalität ist in der Klasse *MagneticGrid* bereits vorgesehen. Was noch fehlt ist eine Möglichkeit für den Benutzer dies aus dem Menü heraus steuern zu können.

Also wird eine neue Aktion in der Klasse *ActionPerformedEvent* angelegt.

```
public enum Action {
/* ... */,
showMagneticGrid;
}
```

Nach erfolgreichem Auslösen des soeben angelegten Ereignisses, muss natürlich auch jemand darauf reagieren.
Da das Raster verändert werden soll und dieses in die Leinwand integriert ist, wird die Klasse *Canvas* dahingehend erweitert.

```
public void actionPerformed(ActionPerformedEvent e) {
 switch (e.getAction()) {
  case showMagneticGrid:
   this.magneticGrid.setMagneticGridDisplayed((boolean)e.getValue());
   this.repaint();
   break;
 /* ... */
 }
}
```

Nachdem die Aktion existiert und auch jemand darauf reagiert, fehlt nur noch eine Möglichkeit diese in der GUI auszulösen. Wie bei den Werkzeugen die ToolTip-Texte nötig sind, werden auch im Menü Texte benötigt, welche aus der Ressourcen-Datei *„svgEditorStrings_de.properties“* gelesen werden können.

```
labelMagGrid=Magnetisches Raster
labelShowMagGrid=anzeigen
labelHideMagGrid=verbergen
```

Wie bei den ToolTip-Texten werden auch noch entsprechende Einträge in der *Reader.StringProperty* Enumeration benötigt:

```
public static enum StringProperty implements ConfigValue {
/* ... */,
labelMagGrid;
labelShowMagGrid;
labelHideMagGrid;
}
```

Im letzten Schritt wird noch der *MenuBarManager* erweitert, um die Aktionen tatsächlich auslösen zu können.

```
private JMenuItem /* ... */ , showMagGrid, hideMagGrid;
private JMenu /* ... */ , magGrid;


private void init() {
 /* ... */

/**
 * MagneticGrid Menu Item *
 */
magGrid = new JMenu(Reader.getPropertyValueAsString(Reader.StringProperty.labelMagGrid));
showMagGrid = new JMenuItem(Reader.getPropertyValueAsString(
                                        Reader.StringProperty.labelShowMagGrid
                    ));
showMagGrid.addActionListener(new ActionListener() {
 @Override
 public void actionPerformed(ActionEvent e) {
  ActionManager.fireActionPerformedEvent(new ActionPerformedEvent(
      ActionPerformedEvent.Action.showMagneticGrid,true));
 }
});
magGrid.add(showMagGrid);
hideMagGrid = new JMenuItem(Reader.getPropertyValueAsString(
                                        Reader.StringProperty.labelHideMagGrid
                    ));
hideMagGrid.addActionListener(new ActionListener() {
 @Override
 public void actionPerformed(ActionEvent e) {
  ActionManager.fireActionPerformedEvent(new ActionPerformedEvent(
      ActionPerformedEvent.Action.showMagneticGrid,false));
 }
});
magGrid.add(hideMagGrid);
menu.add(magGrid);
```

9. Sinnvolle Erweiterungen

An dieser Stelle werden ein paar Ideen für zukünftige Weiterentwicklungen festgehalten. Hierbei handelt es sich sowohl um neue Funktionen als auch um Verbesserungsvorschläge.

SVGElemente:

- **SVGImage**
  In manchen Situationen kann es sinnvoll sein auch Rastergrafiken einzubetten. Da Vektorgrafiken nur Beschreibungen enthalten, müssten diese Rastergrafiken in die Beschreibung hineinkodiert werden.
- **SVGText**
  Das Textelement ist definitiv verbesserungsfähig (z. B. hinsichtlich der Cursorpositionierung).

Aktionen:

- **Snapshot**
  Wie mit Hilfe des *„Snipping Tools“* aus Microsofts Windows 7, könnte auch hier mit einem Rechteck oder einem anderen Element ein Schnappschuss eines Teilausschnittes gemacht werden.
  Java und SVG unterstützen das sogenannte *Clipping* mit dem Teile eines Bildes mit sogenannten Maskierungen ausgeblendet werden können.

Sonstiges:

- **Zwischenablage**
  Das Kopieren aus einem Textverarbeitungsprogramm in die Grafiksoftware wäre für den Benutzer ein komfortables Extra.
- **Zusätzliche Import/Export Formate**
  WMF, EMF, OLE, TIF, …

10. Ähnliche Programme

Um eine Gegenüberstellung mit anderen Programmen zu bieten, werden hier einige ausgewählte Punkte vergleichen:

	SVGEditor	**Inkscape**	**Dia**
Kosten	-	-	-
Werkzeuge			
Selektion	x	x	x
Pfeil	x	-	x
Linie	x	x	x
Dreieck	x	x	-
Polygone	x (inkl. Pfeilspitzen)	x	x
Rechteck	x	x	x
Ellipse	x	x	x
Abgerundetes Rechteck	x	-	x
Rechteck mit runden Seiten	x	-	-
Text	x	x	x
Vieleck	-	x	x
Spirale	-	x	-
3D-Box	-	x	-
Zoom	x	x	x

	SVGEditor	Inkscape	Dia
Funktionen			
Linienstärke	x	x	x
Haarlinie	x	x	-
Linienfarbe	x	x	x
Transparenz	-	x	-
Füllfarbe	x	x	x
Transparenz	--	x	-
Textoptionen			
Größe	x	x	x
Schriftart	x	x	x
Fett	x	x	x
Kursiv	x	x	x
Unterstrichen	x	-	-
Durchgestrichen	x	-	-
Ausrichtung	x	x	x
Farbe	x	x	x
Magn. Raster	x		x
Gruppierung	x	x	x
Duplizieren	x	x	x
Ausrichtung	x	x	x
Drehen	-	x	-
Spiegeln	-	x	-
Drucken	x	x	x

	SVGEditor	Inkscape	Dia
Importformate	SVG, PNG (inkl. Metadaten), PDF (inkl. Metadaten)	SVG, SVGZ, PDF, PNG, PS, EPS, EMF, POV, FX, ODG, TEX, DXF, GPL, HPGL, PLT, SK1, XAML, WMF	DIA, DXF, BMP, EMF, GIF, ICNS, JPEG, PNG, QIF, TGA, WMF, XPM, SVG, VDX, WPG, FIG
Exportformate	SVG, PNG *(inkl. Metadaten)*, PDF *(inkl. Metadaten)*	SVG, SVGZ, PDF, PNG, PS, EPS, EMF, POV, FX, ODG, TEX, DXF, GPL, HPGL, PLT, SK1, XAML, WMF	SVG, EMF, PNG, PDF, PS, WMF, CGM, SHAPE, DIA, DXF, EPS, TEX, GIF, BMP, JPEG, TIF, MP, WPG, FIG, CODE
Einsatzgebiet	Diagramme, Schaltbilder, …	Diagramme, Schaltbilder, …	Vektorkunst, Logos, Landkarten, Stadtpläne, Flugblätter, Diagramme, Poster, Schriftzüge, …

11. Technische Daten

Um ein Gefühl für die Komplexität und den Ressourcenverbrauch der Software zu bekommen, wurden einige Softwaremetriken und Laufzeitanalysen auf verschiedenen Testsystemen erstellt.

11.1. Softwaremetrik

Die folgenden Metriken wurden mit Hilfe eines NetBeans-Plugins namens Source-Code-Metrics von Krystian Warzocha angefertigt [8].

11.1.1. Pakete

	A	I	NCP	LOC
SVGEditor	8%	51%	4	13281
at.hoertler.svgeditor	14%	64%	3	518
at.hoertler.svgeditor.action	0%	14%	1	48
at.hoertler.svgeditor.common	0%	54%	4	1312
at.hoertler.svgeditor.data	0%	66%	4	1705
at.hoertler.svgeditor.display.controls	0%	75%	2	877
at.hoertler.svgeditor.display.menubar	0%	80%	2	1109
at.hoertler.svgeditor.display.toolbar	13%	55%	15	2095
at.hoertler.svgeditor.event	40%	25%	5	554
at.hoertler.svgeditor.options	0%	33%	1	394
at.hoertler.svgeditor.svg.elements	15%	41%	9	4669

A ..... Abstraktheit
I..... Instabilität
NCP..... Number of Classes in Package
LOC..... Lines Of Code

Wie der Tabelle zu entnehmen ist, besitzt die Software in ihrem aktuellen Umfang 13.281 Zeilen Code und durchschnittlich vier Klassen pro Paket.

Die Abstraktheit wird aus dem Verhältnis von abstrakten Klassen und Schnittstellen zur Gesamtsumme aller Klassen gebildet. Eine hohe Instabilität weist auf hohe Abhängigkeit zu anderen Komponenten hin.

Es ist gut erkennbar, dass die für Erweiterungen interessanten Pakete eine höhere Abstraktheit und daher mehr abstrakte Klassen und Schnittstellen enthalten.

11.1.2. Klassen

	LOC	NOM	VG
SVGElement	1428	102	2
ElementManager	1281	52	3
SVGImporter	850	34	4
SVGPolygone	713	39	2
MenuBarManager	587	8	2
Canvas	532	28	3

LOC ..... Lines Of Code
NOM ..... Number Of Methods
VG ..... McCabe Cyclomatic Complexity

In der obigen Tabelle werden die sechs Klassen aufgelistet welche die meisten Codezeilen beinhalten.

An erster Stelle steht die Klasse *SVGElement*. Da es sich um eine Grafiksoftware handelt und die zu zeichnenden Elemente von größter Bedeutung sind, ist dieses Ergebnis nicht verwunderlich.

SVGElement enthält neben vielen abstrakten- auch viele bereits implementierte Methoden welche zum Beispiel den Wechsel zwischen verschiedenen Modi, das Setzen und Speichern von Eigenschaften sowie oft benötigte Hilfsaufgaben übernehmen.

11.2. Laufzeitanalyse

Um einen Eindruck des Ressourcenbedarfs zu bekommen, wurden mit Hilfe des NetBeans Profilers[10] Performancetests auf drei Computersystemen durchgeführt.

11.2.1. System 1

Prozessor:	*Intel® Core™ i7 @3,40GHz*
Arbeitsspeicher:	*8,00 GB*
Grafik:	*NVIDIA GeForce GTX 560 Ti*
Betriebssystem:	*Windows 7 64-Bit*

Rendering Aufwand

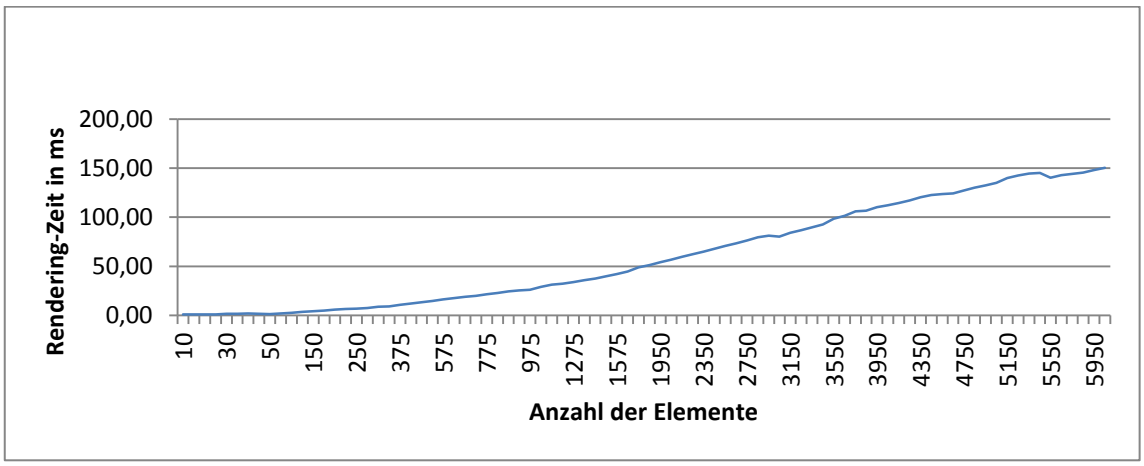


Abbildung 61: Rendering Aufwand Intel Core i7

Speicherbedarf

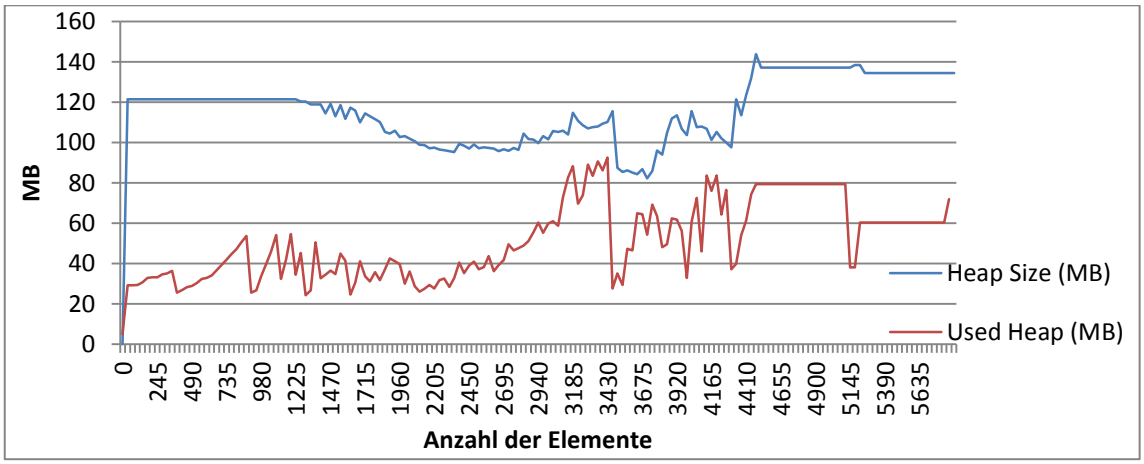


Abbildung 62: Speicherbedarf

[10] Werkzeug der Softwareentwicklung um eine dynamische Analyse des Laufzeitverhaltens durchzuführen und zum Beispiel ineffiziente Codesegmente aufzuspüren.

11.2.1. System 2

Prozessor:	*Intel® Core™ i3 @2,20GHz*
Arbeitsspeicher:	*4,00 GB*
Grafik:	*Intel® HD Graphics Family*
Betriebssystem:	*Windows 7 64-Bit*

Rendering Aufwand

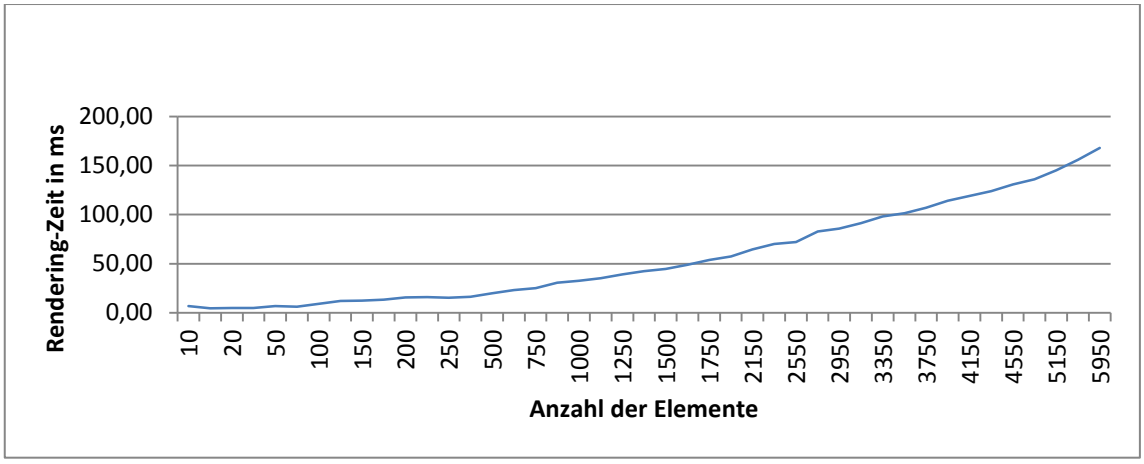


Abbildung 63: Rendering Aufwand Intel Core i3

11.2.1. System 3

Prozessor:	*Intel® Core2Duo @2,26GHz*
Arbeitsspeicher:	*2,00 GB*
Grafik:	*Intel® GMA X4500*
Betriebssystem:	*Windows 7 32-Bit*

Rendering Aufwand

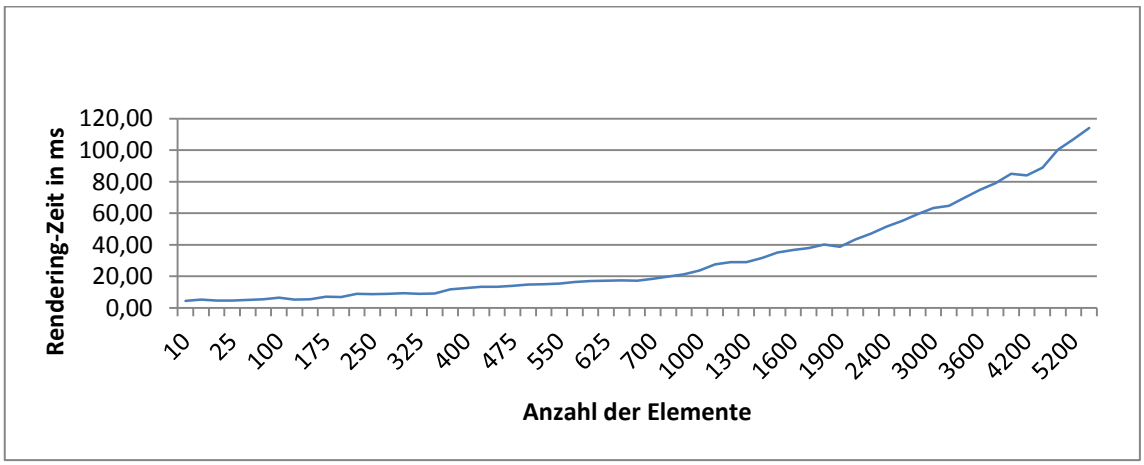


Abbildung 64: Rendering Aufwand Intel Core2Duo

11.2.2. Fazit

Das Zeichnen der Elemente nimmt wesentlich mehr Rechenleistung in Anspruch als erwartet.
Die Ergebnisse bezüglich des Rendering Aufwandes sehen trotz Leistungsunterschieden zwischen den Testsystemen ähnlich aus.
Dies ist möglicherweise darauf zurückzuführen, dass das High-End Testsystem 1 auch eine viel höhere Auflösung von *1680 x 1050 Pixel* hat. Beim Testsystem 3 hingegen handelt es sich um ein Lenovo Thinkpad x200 mit einer Auflösung von nur *1280 x 800 Pixel*.
Die Analyse der Speicherbedarfsanalyse sieht bereits besser aus. Bei 5600 Elementen werden knapp 60 MB Speicher benötigt.

Die aktuelle Implementierung zeichnet mit jedem Repaint alle Elemente neu.
Es gäbe ein paar Möglichkeiten um die Performance zu verbessern. Zum Beispiel:

- Jedes Element in ein *BufferedImage* zwischenspeichern und nur bei Bedarf neu erstellen
- Leinwand nach Veränderung nur partiell neu zeichnen

12. Persönlicher Rückblick

Wie zu Beginn dieser Arbeit berichtet, wurde mir die Aufgabe gestellt, eine Grafiksoftware zu entwickeln, welche den bereits am Anfang dieser Arbeit beschriebenen Kriterien entspricht.

12.1. Erkenntnisse und Verbesserungen

Nachdem ich mich für die Technologie *Java* entschieden hatte, startete ich das Projekt mit dem Entwicklungswerkzeug *Eclipse*.
Mein erstes Ziel war es, ein paar Grundformen zeichnen zu können. Nach und nach wurde das Programm um immer mehr Funktionen erweitert.
Während der Entwicklung stellte sich heraus, dass manche – der zu Beginn entwickelten – Architektur Ideen, bei einigen noch geplanten Erweiterungen des Programms zu Problemen geführt hätten. Aus diesem Grund entschied ich mich für ein umfassendes Überarbeiten und wechselte auch gleich auf das Entwicklungswerkzeug *NetBeans*, da ich mit *Eclipse* – aufgrund von Problemen mit etwaigen Plugins – unzufrieden geworden war.Mit der neuen überarbeiteten Architektur, war das Zusammenspiel der einzelnen Komponenten viel besser und auch überschaubarer.

Im Laufe dieser Arbeit habe ich viele Aspekte entdeckt, welche ich bei einer erneuten Entwicklung einer vergleichbaren Grafiksoftware berücksichtigen würde. Mit der gewonnenen Erfahrung wäre es jetzt einfacher, die Software in mehr Komponenten aufzuspalten und die Kopplung zwischen ihnen deutlich zu verringern.
Bei der Entwicklung des Rasters ist mir durch eine Analyse mit dem *NetBeans Profiler* aufgefallen, dass ich das komplette Raster jedes Mal neu zeichnen lasse, wenn sich das Bild geändert hatte. Da dies eine sehr aufwändige und unnötig oft ausgeführte Prozedur ist, wurde dieses so verändert, dass das Raster zwischengespeichert und nur bei Bedarf erneuert wird. Bei den Laufzeitanalysen am Ende der Entwicklung, erkannte ich die Grenzen der Software bezüglich der Anzahl der gezeichneten Elemente.
Auch hier könnten Verbesserungen vorgenommen werden, welche ich bei einer Neuentwicklung von Anfang an mit einplanen würde.

12.2. Besonderheiten

Neben der einfachen Bedienung besitzt der Grafikeditor verglichen mit anderen Vektorgrafikprogrammen äußerst nützliche Besonderheiten.
Gängige Editoren besitzen alle die Möglichkeit Bilder in verschiedenen Rastergrafikformaten abzuspeichern. Bisher mussten die Bilder dann jedoch in einem Vektor- und einem Pixelformat gespeichert und archiviert werden, um diese in Textverarbeitungsprogrammen verwenden und auch wieder bearbeiten zu können.

Die mit dieser Arbeit entstandene Software ermöglicht dem Anwender neben dem Erstellen von Pixelgrafiken auch das Laden dieser entworfenen Bilder und das Bearbeiten jedes einzelnen Elementes. Eine exportierte Pixelgrafik vereint somit die Vorteile beider Formate.

13. Literaturverzeichnis

[1] World Wide Web Consortium (W3C), „SVG - Scalable Vector Graphics (SVG) 1.1 (Second Edition),“ 2011.

[2] M. B. Michael Bender, Computergrafik - Ein anwendungsorientiertes Lehrbuch, München: Carl Hanser Verlag, 2005.

[3] J. Knudsen, Java 2D Graphics, O'Reilly Media, 1999.

[4] G. Blaschek, AppleDraw - A Desk Accessory for Object Oriented Drawing, 1989.

[5] G. Randers-Pehrson, „World Wide Web Consortium (W3C),“ 10 11 2003. [Online]. Available: http://www.w3.org/TR/2003/REC-PNG-20031110/. [Zugriff am 12 Februar 2013].

[6] F. S. Foundation, „GNU AFFERO GENERAL PUBLIC LICENSE,“ 19 November 2007. [Online]. Available: http://www.gnu.org/licenses/agpl-3.0.de.html. [Zugriff am 9 März 2013].

[7] B. Lowagie, iText in Action, Second Edition, Manning Publications Co., 2010.

[8] K. Warzocha, „Google Code,“ Google, 14 Mai 2012. [Online]. Available: http://code.google.com/p/source-code-metrics/. [Zugriff am 1 März 2013].

14. Bedienungsanleitung – SVGEditor

SVGEditor ist eine praktische Anwendung zur Erstellung von technischen Diagrammen. Durch die einfache und intuitive Bedienung kann der Benutzer ohne viel Lernaufwand qualitativ hochwertige Grafiken erzeugen.

Der *SVGEditor* wird vorzugsweise dazu verwendet, Vektorgrafiken für wissenschaftliche Arbeiten anzufertigen.

In dieser Bedienungsanleitung werden nur jene Funktionen näher behandelt, welche für den Benutzer nicht einfach verständlich sind.

14.1. Funktionsüberblick

- ✓ Zeichnen von grafischen Elementen (Linie, Pfeil, Rechteck, abgerundetes Rechteck, Rechteck mit runden Seiten, Dreieck, Ellipse, Text, Polygon)
- ✓ Verschieben, Kopieren, Skalieren, Verzerren und Duplizieren von Elementen
- ✓ Text mit verschiedenen Formatierungen (Schriftart, -größe, -farbe, -stil, …)
- ✓ Text Ausrichtung (linksbündig, zentriert, rechtsbündig)
- ✓ Text mit Rahmen und Hintergrundfarbe
- ✓ Verschiedene Linienstärken (1 bis 20 Pixel + Haarlinie)
- ✓ Verschiedene Linien- und Füllfarben
- ✓ Raster (sichtbar/magnetisch)
- ✓ Zoom
- ✓ Nachträgliches Bearbeiten von Polygonen
- ✓ Polygone mit Pfeilspitzen
- ✓ Gruppieren und Bearbeitung von Gruppen
- ✓ Einfügen vorhandener SVGEditor-Grafiken
- ✓ Importieren von SVGEditor-Grafiken (SVG, PNG, PDF)
- ✓ Exportieren als SVG, PNG, PDF
- ✓ Pixelgenaues Anpassen mit den Pfeiltasten
- ✓ Ausrichten von (selektierten) Elementen
- ✓ Verschiebung in der Z-Achse
- ✓ Drucken

14.2. Mauszeiger

Während des Arbeitens mit dem Programm verändert sich je nach Situation der Mauszeiger.
Mögliche Situationen sehen Sie hier:

	Selektionswerkzeug
	Selektionswerkzeug über nicht selektiertem Element zum Selektieren
	Selektionswerkzeug über selektiertem Element zum Verschieben
	Zeichenwerkzeug
	Textwerkzeug

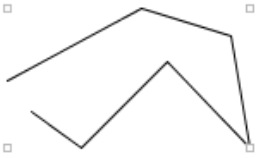
14.3. Elemente zeichnen

Die meisten Elemente werden gezeichnet indem man zuerst das gewünschte Werkzeug wählt und dann auf der Leinwand die linke Maustaste drückt und das Element durch Ziehen der Maus auf die entsprechende Größe aufspannt.
Bei Elementen wie dem Rechteck oder der Ellipse kann man durch gleichzeitiges gedrückt halten der *Strg*-Taste ein Quadrat oder einen Kreis erstellen.
Bei Linien oder Pfeilen können mit der *Shift*-Taste die X- oder Y-Achsen fixiert und somit horizontale oder vertikale Linien/Pfeile erstellt werden.
Einige Elemente wie zum Beispiel Rechtecke können mit Text versehen werden. Um Schreiben zu können, wird ein Doppelklick auf das Element ausgeführt, wodurch ein Cursorsymbol erscheint. Mittels *Shift+Return* kann eine neue Textzeile hinzugefügt werden.

14.3.1. Polygone

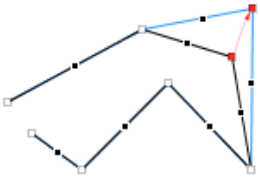
Um ein Polygon zu zeichnen, wird die erste Kante wie eine ganz normale Linie erstellt. Dann wird die Maus an den nächsten gewünschten Punkt bewegt und ein Klick mit der linken Maustaste ausgeführt. Dies wird fortgeführt bis alle gewünschten Punkte hinzugefügt wurden.

Der letzte Punkt eines Polygons wird mit einem Doppelklick gesetzt. Beim Klicken auf den Ausgangspunkt, wird das Polygon geschlossen.

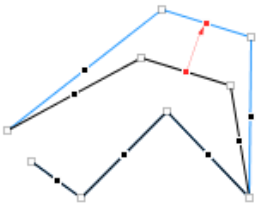


Um ein Polygon nachträglich zu bearbeiten, wird dieses selektier und mit einem Doppelklick in den Bearbeitungsmodus versetzt. Jetzt werden die einzelnen Eckpunkte – welche zum Verändern benötigt werden – sichtbar.

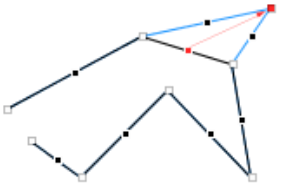
Im Bearbeitungsmodus können die bereits definierten Eckpunkte verschoben werden.



Eine ganze Kante kann verschoben werden, indem die Position ihres Mittelpunktes verändert wird.



Wird der Mittelpunkt einer Kante mit gedrückter *Strg*-Taste verschoben, wird ein neuer Eckpunkt eingefügt.



Durch Anklicken der Endpunkte, können Pfeilspitzen hinzugefügt oder entfernt werden.

14.3.2. Text

Um eine Beschriftung einzufügen wird mit Hilfe des Textwerkzeuges der Cursor an der gewünschten Stelle auf der Leinwand platziert. Nachdem die Position für den Text gewählt wurde, kann ein Text eingetippt, oder aus der Zwischenablage eingefügt werden (*Strg+V*).

Mit der *Return*-Taste wird der Erstellvorgang abgeschlossen. Mit *Shift+Return* wird eine neue Textzeile hinzugefügt.

Wird das Erstellen nicht mit der *Return*-Taste beendet, sondern mit dem Textwerkzeug erneut auf die Leinwand geklickt, wird ein neues Text-Element an dieser Stelle erzeugt.

14.4. Elemente formatieren

Um die einzelnen Elemente weiter zu verfeinern, bietet die Werkzeugleiste entsprechende Hilfsmittel.

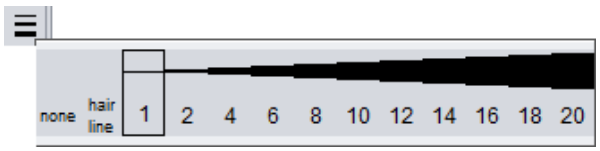
14.4.1. Linienstärke

Um die Dicke einer bestehenden Linie zu verändern, wird diese selektiert und über die Werkzeugleiste die gewünschte Dicke ausgewählt.

Bei gefüllten Elementen kann es auch Sinn machen die Liniendicke auf 0 („*none*“) zu setzen.

Haarlinien entsprechen einer Dicke von 0,25 Pixeln. Der Unterschied zu 1-Pixel-Linien, wird nur in der Vergrößerung, beim Exportieren oder im Druck sichtbar.

Mit Haarlinien können auf entsprechenden Druckern, besonders filigrane Linien und Pfeile erzeugt werden.



14.4.2. Linien- und Füllfarbe

Beinahe jedes Element besitzt eine veränderbare Linien- oder Füllfarbe.

Mit den – aus der Werkzeugleiste – erreichbaren Farbpaletten, kann die farbliche Repräsentation von selektierten oder zukünftig gezeichneten Elementen ganz einfach verändert werden. Standardmäßig sind Linien schwarz und die Füllflächen transparent.

14.4.3. Textoptionen

Texte besitzen ein paar mehr Eigenschaften als normale Elemente.

Mit den bisher bekannten Hilfsmitteln können Texte mit einem Rahmen (Liniendicke/Linienfarbe) sowie einer Hintergrundfarbe (Füllfarbe) versehen werden.

Zusätzlich gibt es die Möglichkeit, die Schriftgröße, Schriftart und die Schriftfarbe anzupassen sowie den Text mit einer Kombination aus fett, kursiv, unterstrichen und durchgestrichen zu versehen.

Bei Texten innerhalb von Elementen oder mehrzeiligen Textfeldern ist auch die Ausrichtung ein wichtiger Punkt.

Es ist nicht möglich in einem Textfeld, Textteile unterschiedlich zu gestalten.

14.5. Elemente bearbeiten

Zusätzlich zur Bearbeitung der einzelnen Elemente mit dem Selektionswerkzeug, bietet der *SVGEditor* noch ein paar sehr nützliche Funktionen.

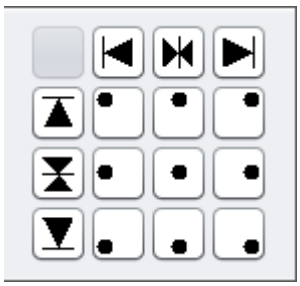
14.5.1. Pixelgenaues Verschieben

Selektierte Elemente können mit den Pfeiltasten pixelgenau ausgerichtet werden. Durch das pixelweise Verschieben, wird das Element möglicherweise aus dem magnetischen Raster der anderen Elemente hinausgeschoben.
Wird beim Ausrichten mit den Pfeiltasten die *ALT*-Taste gedrückt gehalten, bewegen sich die Elemente nicht in Pixel-Schritten sondern in Raster-Schritten über die Leinwand.

Werden die Elemente mit der Maus verschoben, bewegen sich diese automatisch im Raster. Um mit der Maus auf Pixel-Schritte umzuschalten, wird die *ALT*-Taste gedrückt gehalten während das Element bewegt wird.

14.5.2. Ausrichten

Über die Menüleiste kann über den Eintrag „*Ausrichtung*“ die Anordnung der Elemente auf der Leinwand verändert werden.



Wird eine bestimmte Ausrichtung ausgewählt, ohne Elemente selektiert zu haben, wird diese Ausrichtung für die gesamte Leinwand übernommen.

14.5.3. Duplizieren

Es gibt mehrere Möglichkeiten Elemente zu duplizieren.

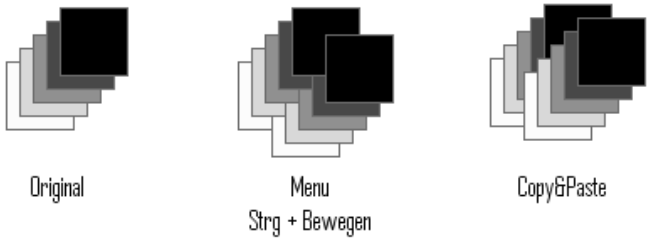
Die naheliegende Variante ist über den Menüpunkt „Duplizieren“. Dies führt zu einer exakten Kopie der selektierten Elemente und selektiert zugleich die Kopien welche etwas nach rechts unten versetzt werden.

Eine weitere, weit verbreitete Variante ist das Duplizieren per *Copy&Paste*.

Der gravierende Unterschied zwischen diesen beiden Varianten ist, dass mittels *Copy&Paste* die Elemente in der obersten Ebene eingefügt werden.

Beim Duplizieren über das Menü werden die einzelnen Elemente in der Z-Achse lediglich eine Stufe über ihr Original gelegt.

Eine Variante welche die gleiche Funktion wie das Menü erfüllt ist, die Elemente bei gedrückter Strg-Taste zu bewegen. Dadurch wird eine exakte Kopie direkt von den Originalen abgezogen.



14.5.4. Gruppierungen

Der *SVGEditor* bietet die Möglichkeit Elemente zu Gruppen zusammenzufassen um eine Grafik strukturieren zu können oder neue Grundelemente anzufertigen.

Um Elemente gruppieren zu können, müssen diese zuerst selektiert werden. Gruppierte Elemente werden als Einheit behandelt. Sie werden zusammen verschoben, verzerrt und skaliert.

Um eine Gruppe bearbeiten zu können, ist es nicht notwendig diese vorher aufzulösen. Der *SVGEditor* unterstützt einen Bearbeitungsmodus für Gruppen, welcher mit einem Doppelklick auf die Gruppierung betreten werden kann.
Im Bearbeitungsmodus werden alle Elemente außerhalb der Gruppe ausgeblendet und die enthaltenen werden in ihrem Originalzustand (also dem Zustand bei der Gruppierung) dargestellt.
Diese Elemente können bearbeitet oder gelöscht werden oder die Gruppe kann auch durch neue Elemente erweitert werden.

Der Bearbeitungsmodus kann mit einem Doppelklick auf eine freie Fläche der Leinwand oder über den eingeblendeten Menü-Eintrag verlassen werden.
Gruppen können wie alle anderen Elemente dupliziert werden.
Gruppen welche Textelemente enthalten können nicht verzerrt werden.

14.6. Vorgefertigte Elemente verwenden

Es gibt einige Elemente welche sehr häufig verwendet werden. Viele dieser oft benötigten Elemente können aus mehreren Grundformen bestehen.
Um diese nicht bei jeder Grafik neu anfertigen zu müssen, können kombinierte Elemente abgespeichert und später in neue Zeichnungen über den Menüeintrag „*Datei -> Grafik importieren*“ einfügt werden.
Die ausgewählte Grafik wird vor dem Einfügen in eine Gruppe verpackt und kann sofort als Ganzes verwendet werden.

Persönliche Daten

Name	Philipp Hörtler
Anschrift	Leonfeldner Straße 99b A-4040 Linz
Geburtsdatum	11.04.1986
Geburtsort	Linz
Nationalität	Österreich

Ausbildung

1992 – 1996	Volksschule Altenfelden
1996 – 2000	BRG Rohrbach
2000 – 2005	HAK-IT Rohrbach
2006 – 2012	Bachelorstudium Informatik an der JKU Linz
Seit 2010	Masterstudium Informatik mit Kernbereich *„Software Engineering“* und Nebenfach *„Netzwerke und Sicherheit“* an der JKU Linz

Berufserfahrung

2006 – 2008	eworx Network & Internet GmbH
2008 – 2009	mangoART GmbH
2009 – 2012	Moviemento Programmkino Gemeinnützige GmbH
2012 – Heute	Keba AG

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Die vorliegende Masterarbeit ist mit dem elektronisch übermittelten Textdokument identisch.