Ausarbeitung

Fachhochschule Lippe und Höxter

Mechanismenmodellierung mittels SVG und SMIL

Im Zuge des Moduls Multimedia und Webtechnologien,
FB Produktion und Wirtschaft der FH- Lippe und Höxter bei
Prof. Dr.-Ing. Stefan Gössner

von
Stefan Oluschinsky (Matr. 15146088)
und
André Heckers (Matr. 15145065)
und
Tim Quisbrock (Matr. 15150054)

Gliederung:

1 SVG Aufbau, SMIL und Zusammenhänge

Im Rahmen der Prüfungsform Präsentation, für das Fach Multimedia- und Webtechnologien entstand diese Ausarbeitung von Stefan Oluschinsky, André Heckers und Tim Quisbrock über das Thema „Mechanismenmodellierung mittels SVG und SMIL“. Die Ausarbeitung beschäftigt sich mit den Grundlagen der Animation von SVG- Grafiken, erläutert dabei die Vorgehensweise und gibt einige Beispiele dazu an.

1.1 Was ist SVG

SVG (Scalable Vector Graphics) ist eine XML-Sprache zur Beschreibung und Integration von zweidimensionalen Vektorgraphiken und ein offizieller Standard vom W3C (World Wide Web Consortium), der im Jahr 2001 vom W3C verabschiedet wurde und sich momentan in der Version 1.1 befindet. SVG wird von der nächsten Generation der gebräuchlichen Webbrowser größtenteils nativ unterstützt. Zurzeit wird noch oft ein Plug-In wie z. B. der SVG-Viewer von Adobe benötigt.
Mit SVG lassen sich textbasiert, d.h. per Texteditor, Grafiken erzeugen, welche einfache Grundformen, aber auch Füllungen mit Farbverläufen oder sogar aufwendige Filtereffekte beinhalten können.
Außerdem kann man mit Hilfe von SVG animierte Grafiken erstellen. Durch beliebige Ereignisse oder an bestimmten Zeitpunkten kann eine Animation gestartet werden, die z.B. die Farbe, Größe oder Position eines Objekts verändert. Ein weiterer Vorteil von SVG besteht in seiner Interaktivität. Mit einer Skriptsprache, wie etwa JavaScript, lässt sich jedes Element der Grafik bequem und einfach manipulieren.
SVG unterstützt also andere Sprachen wie z.B. Javascript (Scripting), CSS (Styling) und SMIL (Multimedia und Animation). Dies liegt nicht zuletzt daran, dass alle eine Teilmenge von XML sind. Warum im Fortlauf des ersten Kapitels noch mal speziell auf SMIL eingegangen wird, hat mit dem Animations-Modul von SVG zu tun, was sich wiederum zum Ende dieses ersten Kapitels erklären wird.

1.2 Der Aufbau von SVG

Der Kopf eines SVG-Dokument sollte immer folgende Elemente beinhalten:

Beispiel anhand eines Listing:

```
<?xml version="1.0"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
"http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg version="1.1"
     xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink">
  <image x="10" y="10" width="200" height="185"
   xlink:href="http://www.mona.de/lisa.png" />
</svg>
```

Laut XML-Deklaration ist die Versionsnummer der XML-Spezifikation 1.0. Die Dokument-Typ-Deklaration von SVG (Public Identifier) lautet PUBLIC "-//W3C//DTD SVG 1.1//EN" "http://www.w3.org/Graphics/SVG/1.1/DTD/-svg11.dtd". Durch sie wird im Allgemeinen die Grammatik einer Klasse von Dokumenten festgelegt, d.h. sie enthält die Informationen darüber, welche *Elementtypen* es gibt, welchen *Inhalt* sie haben dürfen, welche *Attribute* erlaubt sind und welche *Werte* sie annehmen dürfen. . Nach dem Prolog folgt das Wurzelelement. Die Versionsnummer von SVG ist 1.1. Der Bezeichner für den eindeutigen SVG-Namensraum heißt http://www.w3.org/2000/svg. Das Attribut xmlns definiert den Namensraum des Dokuments, sprich die Menge von Namen, die durch eine URI referenziert werden und die in einem XML-Dokument für die Namen von Elementtypen und Attributen verwendet werden können. Namensräume finden Verwendung, wenn in einem Dokument mehrere XML-Sprachen verwendet werden sollen. Beispielsweise gäbe es in einem XML-Dokument, das XHTML- und SVG-Anteile besitzt, Namenskonflikte, da beide Sprachen ein Anker-Element des Namens <a> besitzen.
Möchte man nun eine zusätzliche Sprache verwenden, muss ein weiterer Namensraum festgelegt werden. SVG selbst verwendet einige Attribute der XML Linking Language (XLink). Darum muß in diesem Fall ein zusätzlicher Namensraum angegeben werden. Dies geschieht durch das Attribut-Wert-Paar xmlns:xlink="http://www.w3.org/1999/xlink". Möchte man nun ein XLink-Element oder XLink-Attribut verwenden, so muß es mit dem Präfix xlink beginnen. Das <image>-Tag benutzt beispielsweise das Attribut xlink:href, um eine Referenz auf eine Grafik zu setzen.

Beispiel anhand eines Listing:

```
<?xml version="1.0"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
"http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg version="1.1"
     xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="100%" height="100%" viewBox="0 0 800 600">

    <defs>
      <circle id="kreis" cx="200" cy="200" r="30" fill="red"/>
    </defs>

    <use xlink:href="#kreis"/>

</svg>
```

Der Definitionsabschnitt ist eines der wichtigsten Elemente in SVG. Er dient dazu nicht sichtbare Elemente aufzunehmen. Jegliche SVG-Elemente, welche im Hauptteil zulässig sind, lassen sich im Definitionsabschnitt unterbringen. Dies eignet sich vor allem, wenn man ein Objekt mehrmals aufrufen möchte, denn es muss nur einmal im Definitionsbereich definiert werden. Die Elemente werden zwischen die Tags <defs> und </defs> geschrieben und müssen mit einem einmalig vergebenen id – Attribut versehen werden. Nur dann ist es im Hauptteil möglich ein Element mittels use xlink:href oder url(#element_id) zu referenzieren.

1.3 SMIL

SMIL (Synchronized Multimedia Integration Language) ist eine vom W3C standardisierte, textbasierte Sprache auf XML-Basis zur kombinierten Präsentation multimedialer Daten wie Audio, Video, Text und Graphik. Die Objekte können dabei zeitlich aufeinander abgestimmt werden (synchronized), verschiedenste Medienformate werden unterstützt (multimedia) und alle Objekte werden in einer Präsentation (integration) ausgegeben. Das W3C veröffentlichte im Juni 1998 die erste offizielle Empfehlung für SMIL 1. Die offizielle Empfehlung für SMIL 2 wurde im August 2001 freigegeben. SMIL 2 führte neben einigen Erweiterungen mit einer Modularisierung vor allem grundlegende strukturelle Änderungen ein. So können nun Module aus SMIL besser in andere XML Formate eingebunden werden und diese so erweitern. Beispielsweise wird das Vektorgrafik-Format SVG um das Animation-Modul erweitert. Während SMIL1 ein für sich stehendes Dateiformat ist, kann SMIL2 auch in andere Datei-Formate wie XHTML oder SVG integriert werden. Hierzu wird der Namespace-Mechanismus des XML-Formates verwendet, auf dem alle genannten Formate aufbauen.

1.4 Zusammenhänge

SMIL-Befehle wurden in SVG eingefügt. In der SMILANIM (SMIL Animation specification), die von der SYMM Working Group mit der SVG Working Group des W3C entwickelt wurde, sind diverse Animationselemente auf Basis von SMIL und einige Erweiterungen enthalten, die im Folgenden aufgelistet und später im Kapitel Animationsmöglichkeiten erklärt werden.
Zuerst sind die Elemente aufgeführt, die aus dem SMIL-Wortschatz stammen:

- animate, set, animateMotion, animateColor

Folgende Erweiterungen sind von der Arbeitsgruppe entwickelt worden:

- animateTransform, path Attribut, mpath, keyPoints, rotate Attribute

Auch wenn manche Elemente keine Animation auslösen können, sind sie dennoch nützlich für den Ablauf oder die Darstellung der Animation. SMIL kann aber auch über den Namespace-Mechanismus in andere Datenformate wie SVG eingefügt werden. Und SVG kann wiederum als Komponente in SMIL eingefügt werden.

2 Animationen in SVG

Nachdem die Grundlagen kurz umrissen wurden, möchten wir nun auf den Aufbau von Animations-Tags, die verschiedenen Elemente und Attribute und die Eingliederung der Animations-Tags in den Quellcode näher eingehen.

2.1 Was ist eine Animation

Eine Animation besteht aus einer Serie von Einzelbildern, die räumlich und von Bild zu Bild zeitlich korrellieren. Gewöhnlich ist der zeitliche Abstand so gering, dass der Eindruck einer flüssigen Bewegung entsteht.
Eine Animation in SVG verwendet zwar auch Einzelbilder, jedoch werden diese nicht alle von Hand erstellt, wie etwa bei einem klassischen Zeichentrickfilm (Bsp. Daumenkino). Hier genügt es einen Start- und einen Endzustand einer animierbaren Eigenschaft anzugeben. Die benötigten Zwischenbilder werden mit Hilfe eines Interpolationsverfahrens automatisch vom Computer berechnet.

2.2 Animationsmöglichkeiten und Elemente

Animationselement :		**Beschreibung:**
1.	**animate**	Verändert Attributswerte über die Zeit ; Bsp.: den Radius eines Kreises, oder Breite eines Rechtecks Kann beinahe für alle Animationen verwandt werde, mit Ausnahme der Animation entlang eines Pfades.
2.	**set**	Verändert Attributswerte sprungartig zu einem festgelegten Zeitpunkt innerhalb der Animation. Ist also im Prinzip eine Kurzform des <animate> – Tags, welches jedoch keine Animation im eigentlichen Sinne generiert. Besonders nützlich ist das <set> – Tag, wenn man nichtnumerische Attribute verändern möchte, wie beispielsweise das Attribut „visibility“. Das Tag <set> benötigt das Attribut **to = "<value>"**, um den Attributszustand des zu animierenden Attributes im Zielelement festzulegen.
3.	**animateMotion**	Dient dazu Bewegungen entlang eines Pfades zu generieren. Dazu kann man auf im <desfs> - Bereich vordefinierte Pfade referenzieren, oder dem <animateMotion> - Tag direkt als Attribut „path“ einen Pfad übergeben. (Siehe Animationselement mpath unten)
4.	**animateColor**	Farben lassen sich genauso gut mit dem <animate> – Tag animieren. Zur Übersichtlichkeit und Verständlichkeit des Codes kann/darf man gerne auch auf das animateColor – Element zurückgreifen.

Darüber hinaus können in SVG folgende Animationselemente und Attribute verwandt werden:

Animationselement		Beschreibung:
5.	**animateTransform**	Wird benötigt um Transformationen animieren zu können. Eine kurze Abhandlung über Transformationen finden sie unter 1.2 Transformationen. Grundsätzlich muss dem <animateTransform> - Tag der Transformationstyp mittels type=“ translate| scale	rotate	skewX	skewY“ übergeben werden.
6.	**mpath**	Ist im <defs> -Bereich eine path – Element definiert worden, so kann man dem <animateMotion> -Tag ein child – Element <mpath> zuordnen, welches auf das path – Element referenziert.

2.3 Animationsattribute

Jedes der 6 Animationstags kann durch Übergabe verschiedener Attribute konzipiert werden.
Die Attribute dienen dazu die Animation zu gestalten.

Identifikation des Zielelements:
Zunächst muss dem Animationstag mitgeteilt werden welches Element animiert werden soll. Das wird mittel der Referenzierung über **xlink:href = "<uri>"** realisiert.

Identifikation des Zielattributes:
Nachdem dem Animationstag das Zieltag der auszuführenden Animation mitgeteilt wurde, muss nun das Attribut bestimmt werden welches beim Zieltag animiert werden soll. Über **attributeName = <attributeName>** wird das zu animierende Attribut eindeutig bestimmt.
Mit dem Attribut **attributeType = "CSS | XML | auto"** definieren sie den Namespace des Attributes. Beim voreingestellten Wert „auto“ ermittelt der user agent selbst ob eine CSS-Eigenschaft oder ein XML-Attribut vorliegt.

2.4 Attribute, die die Eigenschaft der Animation beeinflussen

Die folgenden Attribute gelten für die Animtionstags <animate>, <animateMotion>, <animateColor> und <animateTransform>.
Nachdem nun festgelegt worden ist, welches Attribut welchen Elements animiert werden soll, müssen Werte definiert werden, die das Attribut zu Beginn und zum Ende der Animation besitzen soll. Das lässt sich auf verschiede Weise realisieren. Grundlegend sind die folgenden vier Attribute:

from = "**<value>**" Legt den Startwert des zu animierenden Attributes fest
to = "**<value>**" Legt den Endwert des zu animierenden Attributes fest

by = "**<value>**" Angenommen A sei der Startwert, dann liefert by=“B“ den Endwert A+B
values = "**<list>**" Eine durch Semikolon getrennte Liste von Werten, die je nach Interpolationsverfahren animiert werden.

Wenn eine Liste mit Werten spezifiziert worden ist, so werden alle Angaben wie „from“, „by“ oder „to“ ignoriert. Eine „from“- Angabe ist optional, da im Allgemeinen bereits ein Wert des zu verändernden Attributs zugewiesen ist. Jede „from“- Angabe benötigt entweder eine „to“- Angabe oder eine „by“- Angabe. Sind irrtümlicherweise sowohl „to“ als auch „by“ verwandt worden, so wird die „by“ – Angabe ignoriert.
Daraus ergeben sich folgende Animationsmöglichkeiten:

1. From-to-Animation
2. From-by-Animation
3. by-Animation
4. to-Animation

2.5 Was sind Eigenschaften in SVG und wie werden sie animiert?

Beispiel anhand eines Listing:

```
<circle cx=“300“ cy=“100“ r=“100“ fill=“red“ />
```

Das Element circle hat die Eigenschaften cx, cy, r und fill, diese Eigenschaften lassen sich in SVG annimieren. Es gibt in SVG sehr viele Attribute für die einzelnen Elemente und die meisten lassen sich animieren (Referenz unter http://www.w3.org/TR/SVG11/). Wichtig hierbei: eine Animation kann immer nur eine Eigenschaft animieren. In der Animation wird mit „*attributeName*“ die Eigenschaft festgelegt die animiert werden soll.

1. Beispiel anhand eines Listing, wo die Eigenschaft „*fill*“ animiert wird:

```
<svg width=“300“ height=“300“>
<circle cx=“150“ cy=“150“ r=“100“ >
     <animate attributeName=”fill” begin=”0s” dur=”3s”
     from=”black” to=”red”  />
</circle>
</svg>
```

2. Beispiel anhand eines Listing, wo die Eigenschaft „*r*“ animiert wird

```
<svg width="300" height="300">
<circle cx="150" cy="150" >
     <animate attributeName="r" begin="0s" dur="10s" from="50"
     to="130"  />
</circle>
</svg>
```

Der Radius ändert sich von 50 auf 130 Pixel. Es gibt in SVG 2 Möglichkeiten die Eigenschaften festzulegen, in den oberen Beispielen wird die XML Variante benutzt. Es ist aber auch möglich CSS-Stil zu benutzen. Welches Sie benutzen können Sie mit „*attributeType*“ festlegen. Wenn das Attribut nicht festgelegt ist, ist der Standardwert „*auto*“.

Zustände der Eigenschaften festlegen:

Es gibt, wie so oft, in SVG mehrere Möglichkeiten die Eigenschaften zu ändern.

1. Beispiel anhand eines Listing (Elemente: from, to):

```
<svg width="300" height="300">
<circle cx="150" cy="150" >
     <animate attributeName="r" begin="0s" dur="10s" from="50"
     to="130"  />
</circle>
</svg>
```

Der Radius ändert sich vom Startelement „*from*“ mit dem Wert 50 zum Endelement „*to*“ mit dem Wert 130. Der Wert muss dabei dem Typ entsprechen, der auch für die veränderte Eigenschaft gilt. Man kann also wie oben im Beispiel beschrieben, statt des Radius keine Farbe eingeben.

2. Beispiel anhand eines Listing (Elemente: from, by):

```
<svg width="300" height="300">
<circle cx="150" cy="150" r=50″>
     <animate attributeName="r" begin="0s" dur="10s"  by="80"/>
</circle>
</svg>
```

Die Animation ist die gleiche wie im ersten Beispiel, allerdings beträgt hier der Wert für „*by*“ nur 80, in diesem Beispiel ist das Element „from“ die Ausgangsposition und der Wert von „by“ wird dazu addiert, also 50+80=130.

Values:

Bis jetzt konnten wir nur 2 Zustände festlegen, Start und Endpunkt bzw. die Veränderung. Mit dem Attribut „*values*“ lassen Sich mehr wie 2 Zustände festlegen.

Beispiel anhand eines Listing :

```
<svg width="400" height="400">
  <circle cx="100" cy="100" r="25">
  <animate attributeName="cx" values="100; 200; 50"
  begin="0s" dur="5s" />
  <animate attributeName="cy" values="100; 200; 50"
  begin="0s" dur="5s" />
  <animate attributeName="fill" values="black; red; green"
  begin="0s" dur="5s" />
  </circle>
</svg>
```

Der Schreibaufwand verringert sich im Gegensatz zu from/to und bei mehr wie 2 Zuständen muss das Attribut „*values*“ verwendet werden.

2.6 Addition in Animationen

Es gibt noch eine dritte ergänzende Möglichkeit die Zustände zu animieren, über das Attribut „*additive*“. Dessen mögliche Werte sind:

- sum= Der jeweils zu setzende Wert wird zum Ursprungswert hinzugefügt
- replace= Der Ursprungswert wird durch den zu setzenden ausgewechselt

1. Beispiel anhand eines Listing :

```
<svg width="300" height="300">
<circle cx="150" cy="150" r=”50” >
     <animate attributeName="r" begin="0s" dur="10s" from="0"
     to="80"  additive=”sum” />
</circle>
</svg>
```

Das Ergebnis ist das gleiche wie beim „*by*“ Attribut, sollen allerdings mehrere Zustände dargestellt werden, kann dies nur über das Attribut „additive“ erreicht werden.

2. Beispiel anhand eines Listing (Vergleich von 3 Varianten) :

```
<svg width=“800“ height=”600”>
<circle cx="120" cy="160" r="40" fill="orange">
<animate attributeName="r" begin="0s" dur="2s" values="0; 50;
100" fill="freeze"/>
</circle>
<circle cx="420" cy="160" r="40" fill="orange">
     <animate attributeName="r" begin="0s" dur="2s" values="0;
     50; 100" additive="sum"
          fill="freeze"/>
</circle>
<circle cx="720" cy="160" r="40" fill="orange">
     <animate attributeName="r" begin="0s" dur="2s" values="0;
     50; 100" additive="replace"
          fill="freeze"/>
</circle>
</svg>
```

Die Animation des ersten Kreises ohne „*additive*“ Attribut und des dritten Kreises mit der Angabe additive=“replace“ verlaufen absolut gleich. Und wie man in der Abbildung erkennen kann, besitzen sie am Schluss auch denselben Radius, da der Wert von r immer zuerst auf 0, dann auf 50 und zuletzt auf 100 gesetzt wird. Die Angabe additive="sum" sorgt hingegen dafür, dass der ursprüngliche r-Wert 40 zuerst mit 0, danach mit 50 und zum Schluss mit 100 addiert wird. Wird „*additive*“ nämlich nicht benutzt, springt die zu animierende Eigenschaft nach jedem Zustand wieder auf ihren ursprünglichen Wert zurück. Fazit hierbei: Bei einfachen Animationen ist das „by“ Attribut der Favorit, sobald aber mehrere Zustände dargestellt werden sollen, wird das Attribut „*additive*“ verwendet.

2.7 Akkumulation

Möchte man eine Animation mehr als nur ein einziges Mal ablaufen lassen, kann man mit Hilfe des Attributs „*repeatCount*“ die Anzahl der Wiederholungen festlegen. Das Problem dabei ist jedoch, dass nach jeder Wiederholung die zu animierende Eigenschaft auf Ihren Ursprungswert gesetzt wird. Um dies zu vermeiden, setzt man das Attribut „*accumulate*“ ein, welches die folgenden Werte besitzen kann:

- sum= Baut auf den vorherigen Zustand auf
- none= Ersetzt den vorherigen Zustand (Standard)

Beispiel anhand eines Listing :

```
<svg width=“800“ height=”600”>
<circle cx="100" cy="50" r="40" fill="red">
   <animate attributeName="cx" begin="0" dur="1s" by="100"
   repeatCount="3" fill="freeze" />
</circle>
<circle cx="100" cy="190" r="40" fill="red">
     <animate attributeName="cx" begin="0" dur="1s" by="100"
     repeatCount="3" accumulate="sum"
     fill="freeze" />
</circle>
</svg>
```

Die beiden Kreise, die in diesem Beispiel definiert wurden, unterscheiden sich nur durch das „accumulate“ Attribut. Die Animation im ersten Kreis sorgt dafür, dass dieser innerhalb einer Sekunde um 100 Einheiten nach rechts verschoben wird. Wegen der Eigenschaft „repeatCount“, wird diese Animation drei Mal hintereinander durchlaufen, wobei das Objekt nach jedem Animationsdurchgang immer wieder an seinen Ursprungsort zurückspringt. Im Gegensatz dazu bewirkt accumulate="sum" im Animationselement des zweiten Kreises, dass dieser nicht nach jedem Animationslauf zur Ursprungsposition zurückfällt. Stattdessen baut eine Animation auf den jeweils letzten Zustand der vorherigen auf. In diesem Beispiel verschiebt sich der zweite Kreis also nicht nur um 100, sondern um drei mal 100 Pixel.

2.8 Zeitattribute

Die einzelnen Animationselemente in SVG kennen eine Vielzahl an Attributen, die Zeitangaben aufnehmen können. Die folgenden Tabellen beinhalteten die Namen dieser Attribute, eine kurze Beschreibung, Beispiele und die Werte, die die Attribute annehmen können. Letztere werden im nächsten Abschnitt aber noch mal detailliert erläutert.

Attribut	**Beschreibung**	**Beispiele**
begin	Beginn der Animation	<animate begin = "19s" … />
dur	Dauer der Animation	<animate ... dur="42s"/>
end	Ende der Animation	<animate … end="5"/>
restart	Verhalten der Animation bei Wiederholungen	<animate ... restart="never"/>
repeatCount	Anzahl der Wiederholungen	<animate ... repeatCount="3" />
repeatDur	Gesamtzeit der Animation inkl. Wiederholung	<animate ... repeatDur="3"/>
fill	Verhalten des animierten Objekts nach erfolgreichen Animationsdurchlauf	<animate … fill="freeze"/>

Attribut	Werte
begin	offset-value, syncbase-value, event-value, repeat-value, accessKey-value, wallclock-sync-value:wallclock, indefinite
dur	clock-value, media, indefinite
end	offset-value, syncbase-value, event-value, repeat-value, accessKey-value, wallclock-sync-value:wallclock, indefinite
restart	always, whenNotActive, never
repeatCount	repeat-value, indefinite
repeatDur	clock-value, indefinite
fill	freeze, remove

2.9 Spezielle Werte für Zeitattribute

offset-value:
Hier muß ein negativer oder positiver Zeitwert eingegeben werden.
Bsp.: <animate begin="19s" ... />

syncbase-value:
Hiermit wird eine Animation in Abhängigkeit einer anderen Animation ausgelöst oder beendet. Über eine Referenz werden diese miteinander verknüpft.
Bsp.: <animate begin="referenz.end-3s" ... />

event-value:
Damit wird die Animation mittels eines Ereignisses beeinflußt. Dabei wird ein Ereignis ausgelöst, wenn...

- focusin - das Element den Fokus erhält
- focusout - das Element den Fokus verliert
- activate - das Element aktiviert wird (z.B. durch eine Tastatureingabe)
- click - ein Klick auf die klickbare Fläche des Elements erfolgt (hier werden ebenfalls die Ereignisse mousedown und mouseup ausgelöst)
- mousedown - eine Maustaste gedrückt wird, während sich der Mauszeiger zum Zeitpunkt des Klickens über dem Element befand
- mouseup - die gedrückte Maustaste wieder losgelassen wird (wird immer in Verbindung mit mousedown ausgelöst)
- mouseover - sich der Mauszeiger über dem Element befindet
- mousemove - der Mauszeiger über dem Element bewegt wird
- mouseout - der Mauszeiger das Element wieder verläßt

Bsp.: <animate begin="mouseout" ... />

repeat-value:
Hier muss die Anzahl an Wiederholungen eingegeben werden. Außerdem kann festgelegt werden, nach wie vielen Wiederholungen die Animation enden soll.
Bsp.: <animate ... repeatCount="3" end="repeat(2)" />

accessKey-value:
Hier wird die Animation erst gestartet oder beendet, wenn eine spezielle Taste gedrückt wird. Es ist zu beachten, dass nicht alle Tasten, wie etwa STRG oder ALT, unterstützt werden.
Bsp.: <animate begin="accessKey(B)" ... />

wallclock-sync-value:
Durch diese Angabe lässt sich eine Animation zu einer bestimmten Uhrzeit starten bzw. beenden. Dabei kann eine einfache Zeitangabe, wie z.B. 11:47, oder eine komplette Weltzeitangabe (2004-03-17T22:35+01:00) benutzt werden.
Bsp.: <animate begin="wallclock-sync-value:wallclock(07:30)" ... />

indefinite:
Dies bedeutet, dass eine Animation zu einem unbestimmten Zeitpunkt startet oder endet. Bei den Attributen repeatCount oder repeatDur bedeutet indefinite auch unendliche Wiederholungen bis die Animation etwa durch ein Skript beendet wird. Wird indefinite in das begin-Attribut geschrieben, so lässt sich die Animation nur durch einen Hyperlink oder mit Hilfe eines Skripts und dem Aufruf beginElement() starten. Ähnlich verhält es sich beim end-Attribut, hier kann die Animation nur durch den Aufruf endElement() gestoppt werden. Tritt der Wert indefinite im dur-Attribut auf, so wird keine Interpolation betrieben, sondern der Wert der zu animierenden Eigenschaft wird sofort gesetzt.
Bsp.: <animate id="referenz" begin="indefinite" ... />

clock-value:
Dieser Wert kann verschiedene Zeitformate aufnehmen.
Bsp.: <animate ... dur="2:00" /> <animate ... dur="33s" />

media:
media sorgt dafür, dass die Animation genau so lange läuft wie ein zusätzlich eingebundenes Medium.
Bsp.: <animate ... dur="media" />

always, whenNotActive, never:
Diese Werte tauchen nur im Attribut restart auf und haben folgende Bedeutung:

- always - die Animation läßt sich zu jedem Zeitpunkt immer wieder starten
- whenNotActive - die Animation kann nur neu gestartet werden, wenn sie nicht läuft, d.h. sie lässt sich nicht unterbrechen.

- never - die Animation kann nach einem einzigen Durchlauf nicht wieder gestartet werden.

Bsp.: <animate ... restart="never" />

freeze, remove:
Diese beiden Werte können nur dem Attribut fill zugewiesen werden. Die Werte bedeuten dabei folgendes:

- remove - nach Ablauf der Animation wird das animierte Objekt wieder an den ursprünglichen Ort zurückgesetzt.
- freeze - das Objekt verbleibt nach Ablauf der Animation in seiner neuen Position.

Bsp.: <animate ... fill="freeze" />

2.10 Übergänge in Animationen steuern

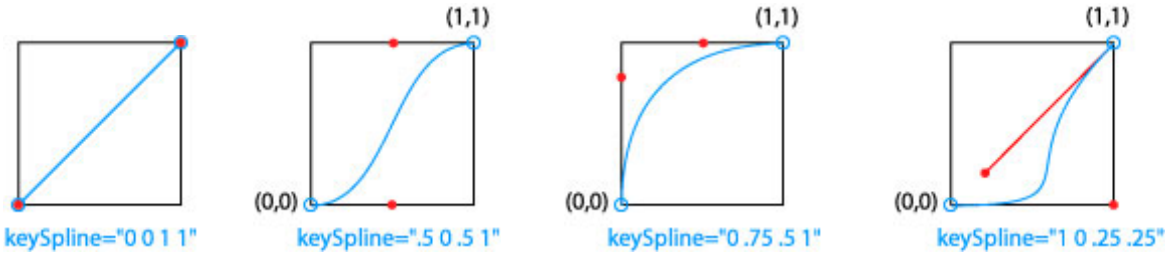
Sind nun Werte für das Attribut vergeben worden, kann der Animation über **calcMode = "discrete | linear | paced | spline"** die Interpolationsmethode übergeben, mit dem der Viewer die Zustände zwischen dem Start und Endzustand der Animation errechnet. Eine detaillierte Beschreibung der Interpolationsverfahren findet sich in Abschnitt 2.12 Interpolationsverfahren. Mit Ausnahme von <**animateMotion>**, bei dem **„paced“** voreingestellt ist, ist **„linear“** für **calcMode** voreingestellt. Damit sind die Grundlegenden Animationsattribute beschrieben. Darüber hinaus besteht die Möglichkeit über **keyTimes = "<list>"** eine Liste von Zeitwerten zu übergeben, welche die Schrittsteuerung der Animation bestimmt. Die Zeitwerte werden durch sich relativ auf die Dauer der Animation beziehende Werte aus dem Intervall [0,1] festgelegt. Vorraussetzung für eine „keyTimes“ - Liste ist eine „value –Liste“, die Zustandswerte für das Attribut enthält, welche jeweils zu einem Listenelement der „keyTimes“ – Liste zugehörig sind. Die Liste der „keyTimes“ muss dabei genau so viele Elemente enthalten, wie die „value“ – Liste.

Dazu eine kleine Erläuterung:

Angenommen die Animation startet bei 0 Sekunden, und endet zum Zeitpunkt 10 Sekunden. Zunächst definieren wir eine „value“ – Liste derart: **values="0;20;10;30".** Die Dazugehörige Liste der „keyTimes“ sei: **keyTimes="0;0.25;0.5;1".** Zu guter Letzt nehmen wir an, wir wollten das Attribut „r“ eines beliebigen Kreises animieren. Dann liefert uns die Animation:

```
<animate attributeName=“r“ begin=“0s“ dur=“10s“
         values="0;20;10;30" keyTimes="0;0.25;0.5;1">
```

zum Zeitpunkt 0 gar keinen Kreis, oder einen solchen mit keinem Radius, zum Zeitpunkt 2,5 Sekunden (0,25 * 10 Sekunden) einen Kreis mit Radius 20, zum Zeitpunkt 5 Sekunden einen mit Radius 10 und schließlich zum Zeitpunkt 10 Sekunden besitzt der Kreis einen Radius von 30.
Im Bezug auf die Werte der „keyTimes“ – Liste ist darauf zu achten, dass für eine lineare - oder eine spline- Animation der erste Wert der Liste eine 0 und der letzte Wert der Liste eine 1 ist.
Einen ähnlichen Effekt kann man mittels „keySplines“ erreichen. Über das Attribut **keySplines = "<list>"** kann man eine kubische Bézier Funktion generieren, welche die Schrittsteuerung innerhalb des Intervals zwischen zwei Werten einer „value“ – Liste festlegt. Somit sind beschleunigte Bewegungsabläufe realisierbar.
Eine „keySplines“ – Liste besteht immer aus 4 Werten. Wobei das erste Wertepaar den ersten und das zweite Wertepaar den zweiten Stützpunkt in Koordinatenform festlegt. Die kubische Bézier Funktion hat den Definitions – und Wertebereich [0,1]. Startpunkt ist (0|0) und Endpunkt (1|1). Hier Zwei Beispiele solche „keySplines“:



Zu 1: Die Kontrollpunkte der Kurve sind Rot gekennzeichnet. Sie liegen auf den Punkten (0,0) und (1,1). Es entsteht eine konstante Geschwindigkeit. Die Werte für das Attribut „keySpline“ sind die Koordinaten der Stützpunkte. Es ist jeder Stützpunkt (x,y) mit x,y aus dem Intervall [0,1] denkbar.
Zu2: Es entsteht eine Geschwindigkeitskurve, welche typisch für Bewegungen mit beschleunigtem Start und verzögerten Ende ist.
Zu3: Schneller Start; Geschwindigkeit wird direkt verzögert.
Zu4: Langsamer Start geht in relativ konstante Geschwindigkeit über; ohne Verzögerungsphase am Schluss (abruptes Stehen bleiben).

Möchte man dies nutzen, so ist neben der Attributsübergabe „keySpline“ die Interpolationsmethode über das Attribut **calcMode** auf “spline“ zu setzen. (Siehe auch Abschnitt 2.12 Interpolationsverfahren).

2.11 Spezielle Attribute für <animateMotion>

Animationsattribute		Beschreibung:
1.	**calcMode**	Bestimmt die Interpolationsmethode. Siehe dazu auch Abschnitt 2.12 Interpolationsverfahren.
2.	**path**	Über das Attribute path kann man dem <animateMotion> -Tag auch direkt übergeben, entlag welchen Pfades es animieren soll.
3.	**keyPoints**	Sind „keyTimes“ definiert, kann man mittels „keyPoints“ (Werte aus [0,1]) der Animation mitteilen, zu welchen Zeitpunkten wie viel des Pfades realisiert sein soll.
4.	**rotate = "<angle> | auto | auto-reverse"**	Mit rotate=”auto” orientiert sich das Objekt relativ zum Pfad. Über „auto-reverse“ generiert man eine um 180 Grad gedrehte Orientierung des Objekts am Pfad, oder man gibt einen Winkel an.

2.12 Interpolationsverfahren

Eine SVG Animation besitz grundsätzlich einen Start- und Endzustand, sowie eine Dauer, die sich eindeutig aus diesen ergibt. Die restlichen Zustände, also alle Zustände zwischen dem Start- und Endzustand werden vom Viewer interpoliert. Durch das Attribut „calcMode“ kann man dem Viewer eine von vier verschieden Interpolationsverfahren übergeben. Folgende Methoden können benutzt werden; voreingestellt für ist das lineare Interpolationsverfahren (mit Ausnahme des <animateMotion> -Tags, bei dem „paced“ voreingestellt ist):

calcMode = "**discrete | linear | paced | spline**"

Discrete
Beim Berechnungsverfahren “discrete“ werden keine Interpolationen zwischen den Werten über value=“a,b,c“ errechnet. Die Animation läuft sprungartig über die angegebenen Zwischenwerte a, b und c ab. Einsetzen kann man dieses Verfahren beispielsweise bei einem Countdown oder dem Sekundenzeiger einer Uhr.

Linear
Lineare Interpolation zwischen den Werten über „value“.
Die Zeit , welche benötigt wird, um die Werte über „value“ zu animieren ist immer gleich, was zur Folge hat, dass die Geschwindigkeit der Animation bei größeren Differenzen zwischen zwei Werten über „value“ größer ist, als bei kleinen Differenzen.

Paced
Lineare Interpolation zwischen den Werten über „value“.
Allerdings ist hier die Geschwindigkeit der Animation zwischen den „values“ immer gleich, und die benötigte Zeit zwischen den Zuständen kann variieren.

Spline
Beim Interpolationsverfahren „spline“ wird die Animation anhand einer Bézier – Kurve interpoliert. Ist calcMode=“spline“ eingestellt, so muss zusätzlich über das Attribut „keySpline“ (siehe 1.2.3) eine Liste mit den Koordinaten der Stützpunkte einer solchen Bézier – Kurve über dem Intervall $[0,1]^2$ angegeben werden.

3 Transformationen

Transformationen in SVG beruhen auf Translation, Skalierung, Drehung und Scherung (oder Neigung) des Koordinatensystems. Mittels Multiplikation mit einer (3x3) – Matrix werden die Koordinatenpunkte eines transformierten Elements errechnet. Näheres zum Berechnungsverfahren findet sich in Abschnitt 3.4 Eigene Matrizen. In SVG besteht die Möglichkeit auf vier vordefinierte Transformationen zuzugreifen, womit ein Großteil der Anwendungsgebiete abgedeckt ist. Transformationen werden in SVG dem Attribut **„transform“** zugewiesen. Die vier vordefinierten Transformationen sind:

transform=“translate | scale | rotate | skewX | skewY“

Translate

Eine Translation ist eine Verschiebung eines Objektes. Übergeben werden zwei Parameter:
translate(x-Verschiebung [y-Verschiebung]). Die Übergabe des zweiten Parameters kann, muss aber nicht erfolgen. Die Verschiebungen folgen dabei dem SVG internen Koordinatensystem: x⟶ ↓y .

Scale

Skalierungen dienen dazu Objekte zu vergrößern oder zu verkleinern. Analog zu „translate“ werden auch hier 2 Parameter Übergeben: **scale(x-Skalierung [y-Skalierung])**. Wird kein Wert für die y-Skalierung übergeben, so erfolgt eine proportionale Skalierung.

Rotate

Drehungen um einen Drehpunkt lassen sich mit: **rotate(Drehwinkel [x-Drehpunkt y-Drehpunkt])** erzeugen. Werden keine Drehpunktkoordinaten angegeben, erfolgt die Rotation um den Koordinatenursprung. Analog zum SVG eigenen Koordinatensystem entspricht ein positiver Winkel einer Rotation im Uhrzeigersinn.

SkewX; SkewY

Das SVG eigene Koordinatensystem hat einen Scherungswinkel von 90 Grad. Mit den Transformationen skewX und skewY lässt sich dieser Winkel für einzelne Objekte ändern. Das erfolgt mittels: **skewX(Winkel der x-Achsen-Neigung)** und **skewY(Winkel der y-Achsen-Neigung).** Allerdings ist darauf zu achten (falls das Element seinen Startpunkt nicht in (0,0) hat), dass sich durch

eine Transformation des Scherungswinkels nicht bloß das Erscheinungsbild des Elements verändert, sonder auch seine Position selbst. Deshalb ist es manchmal notwendig mehrere Transformationen hintereinander durchzuführen.
Das geschieht, indem man dem dem Attribute „transform“ eine Befehlskette übergibt. Bsp.:**transform="translate(10, 10) scale(2)"**. (Hierbei würde zunächst vergrößert, und anschließend verschoben) Dabei ist unbedingt auf die Reihenfolge zu achten, welche das Ergebnis beeinflusst.

Eigene Matrizen

Neben den vier vordefinierten Transformationen besteht auch die Möglichkeit eigene Transformationsmatrizen zu verwenden.
Da es sich bei SVG um Vektorgraphiken handelt, ist jedem Punkt eines Elements ein Ortsvektor zugeordnet. Transformationen erfolgen durch Multiplikation mit einer (3x3)-Matrix der folgenden Form:

$$\begin{pmatrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{pmatrix}$$

Eine Transformation in SVG ließe sich dann mit **transform=“matrix(a, b, c, d, e, f)“** realisieren.

Sei also $a = \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$

der Ortsvektor zum Punkt (x,y). Dann errechnet sich der Ortsvektor zum transformierten Punkt folgendermaßen:

$$\begin{pmatrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} a*x + c*y + e \\ b*x + d*y + f \\ 1 \end{pmatrix}$$

SVG ist ein zweidimensionales Graphikformat, warum also eine Multiplikation in drei Dimensionen? Grund ist, dass man so auch die einfache Translation, also Verschiebung mit einer solchen Matrixtransformation realisieren kann.
Aus obiger Formel ergeben sich die Berechnungsschemata für die vier Standardtransformationen derart:

1. Translation:

$$\begin{pmatrix} 1 & 0 & \Delta x \\ 0 & 1 & \Delta y \\ 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} x + \Delta x \\ y + \Delta y \\ 1 \end{pmatrix}$$

2. Skalierung:

$$\begin{pmatrix} sx & 0 & 0 \\ 0 & sy & 0 \\ 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} sx*x \\ sy*y \\ 1 \end{pmatrix}$$

3. Rotation um den Winkel α:

$$\begin{pmatrix} \cos(\alpha) & -\sin(\alpha) & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} sx*x \\ sy*y \\ 1 \end{pmatrix}$$

4. Scherung der x-Achse um den Winkel α

$$\begin{pmatrix} 1 & \tan(\alpha) & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} x + y*\tan(\alpha) \\ y \\ 1 \end{pmatrix}$$

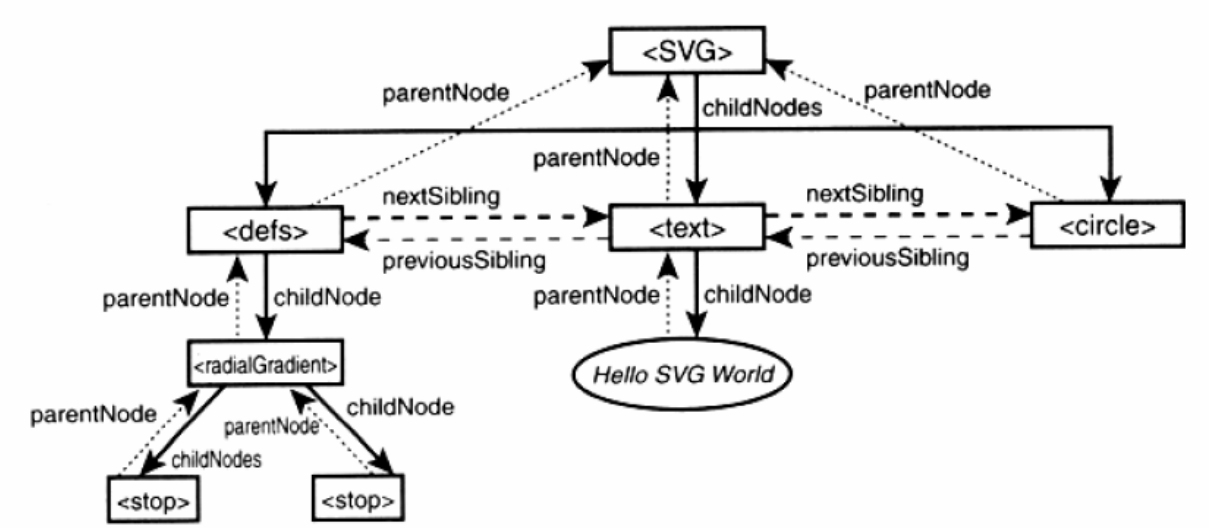
5. Scherung der y-Achse um den Winkel α:

$$\begin{pmatrix} 1 & 0 & 0 \\ \tan(\alpha) & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y + x* \tan(\alpha) \\ 1 \end{pmatrix}$$

4 SVG und Javascript

4.1 SVG-Dom

Dass SVG ein XML-Dialekt ist, merkt man schon am SVG-DOM. Die Elemente des SVG-Dokuments sind in einer baumähnlichen Struktur angeordnet.



Der Baum besteht aus Knoten, die verschiedene Elemente, Attribute, Events etc. repräsentieren. Es gibt 12 verschiedene Knotentypen. Der Knoten an der Spitze des Baumes ist das <svg>-Tag, auch Wurzelelement genannt, da sich alle anderen Elemente von diesem Element ableiten. Das <svg>-Tag kann nun beliebig viele Kind-Elemente besitzen. Diese Kind-Elemente, die auf dem 2ten Level des Baumes angeordnet sind, haben zum <svg>-Element eine Eltern-Kind-Beziehung. Alle Level 2 Elemente haben untereinander eine Geschwister-Beziehung. Im DOM hat jeder Knoten Eigenschaften, die mittels JavaScript abgefragt und geändert werden können. Die Beziehungen zwischen den Knoten ermöglichen es, im DOM zu navigieren. Dafür gibt es verschiedene Methoden wie firstChild, nextSibling oder nodeName etc.

4.2 Interaktivität

Interaktivität bedeutet, dass man ein Dokument bzw. sein Inhalt, der in einem Programm (z.B. Browser) dargestellt wird, manipulieren kann, ohne das Programm zu schließen. Zur Realisierung der Interaktivität sind Sprachen wie Javascript nicht mehr wegzudenken. So gut wie jede Benutzerinteraktion kann erkannt und auf verschiedenste Art und Weise beantwortet werden. Durch das DOM (Document Object Model) ist es möglich, auf die Elemente und Attribute eines Dokuments zuzugreifen und diese zu modifizieren. SVG bietet neben dem SVG-DOM auch ein Event-Modell an. Wer also Javascript in HTML schon programmiert hat, wird viele Ähnlichkeiten zum SVG-DOM und Event-Modell feststellen.

4.3 Skripte

Javascript wird ähnlich wie CSS in den Skript-Bereich eines SVG-Dokument eingefügt. Bei beiden handelt es sich ja um gänzlich andere "Sprachen". Im <defs>-Bereich bindet man das Skript über das <script>-Tag ein, wobei man über das Attribut type mit dem Wert "text/javascript" die Skriptsprache dem Interpreter bekannt gibt. Der eigentliche Skript-Code befindet sich im <![CDATA[...]]>-Tag, wie bei CSS. Das sieht dann so aus:

```
<defs>
  <script type="text/javascript"><![CDATA[
    hier dann der JavaScript-Code!
  ]]></script>
</defs>
```

CDATA steht für "character data". Liegt ein Bereich innerhalb eines solchen CDATA-Blocks, wird alles, was sich darin befindet, vom Parser ignoriert. Nur die Zeichenfolge]]>, die das Ende des CDATA-Blocks bekannt gibt, wird von ihm interpretiert.
Im Folgenden soll der Aufbau eines Skriptes in SVG anhand eines Beispiels genauer erläutert werden. Das Beispiel stellt einen Kreis dar, der die Farbe ändert, wenn man mit dem Cursor über ihn fährt, und seine Ursprungsfarbe zurückerhält, wenn der Cursor die Kreisfläche wieder verlässt:

```
<svg>
  <script type="text/javascript">
  <![CDATA[
  function toggleColor(evt)
  {
    var svgdoc = evt.getTarget().getOwnerDocument();
    var change = svgdoc.getElementById("farbe");
    var color = change.getAttribute("fill");
    if(color == "red")
    {
      change.setAttribute("fill","green");
    }
    else
    {
      change.setAttribute("fill","red");
    }
  }
  ]]>
  </script>
  <circle id="farbe" cx="200" cy="200" r="100" fill="red"
   onmouseover="toggleColor(evt)"
   onmouseout="toggleColor(evt)"/>
</svg>
```

Zugriff auf das SVG-Dokument:
Am Anfang jedes Skriptes in SVG steht der Event, dass das Skript ausgelöst hat. Das Event-Objekt heißt **evt**. Die verschiedenen Event-Objekte (MouseEvent etc.) entsprechen übrigens völlig den DOM-Level-2 Event-Objekten. **evt** besitzt zudem die Funktion **getTarget()**, die das Element liefert, welches den Event ausgelöst hat. Und jedes Element besitzt wiederum die Funktion **getOwnerDocument()**, die das SVG-Dokument liefert.
Die Zeile **var svgdoc = evt.getTarget().getOwnerDocument();** speichert das SVG-Dokument in der Variable svgdoc. Dies ist meistens am Anfang einer Funktion sinnvoll.

Zugriff auf die Elemente des SVG-Dokumentes:
Um auf einen Knoten zuzugreifen, stehen zwei Möglichkeiten offen:
1. Das Benutzen von getNextSibling.
2. Das Benutzen von getElementById() oder getElementsByTagName().
Im Beispiel benutzen wir **getElementById()**, das uns aufgrund der id den gewünschten Knoten liefert. Diesen speichern wir in der Variable change. Im Zusammenhang mit getElementsByTagName() sollten auch noch die Methoden des Listenobjekts getLength() und item() aufgeführt werden. Mit getLength() kann die Länge der Liste ermittelt werden. item() hingegen liefert eine Referenz auf ein Objekt in der Liste. Dazu erhält die Methode als Parameter den Index des gewünschten Elements.

Attribute auslesen und neu zu setzen:
Dazu werden vorrangig zwei Methoden verwendet:
1. getAttribute("attributeName"): Attribute wird ausgelesen
2. setAttribute("attributeName";"wert"): Attribut erhält neuen Wert
Mit getAttribute() können wir auf das gewünschte Attribut des Elements (hier: fill) zugreifen und den Wert auslesen, der im Beispiel in der Variable color gespeichert wird. Mit setAttribute() ordnen wir dem Attribut fill einen neuen Wert zu.

Die beiden Methoden getAttribute() und setAttribute() funktionieren nur bei XML-Attributen. Sind die Eigenschaften im Stil enthalten, funktionieren diese beiden Methoden nicht mehr. Mit getStyle() kann der Stil eines Elements abgefragt werden. Folgende Methoden können nun eingesetzt werden:
1. getPropertyValue() hat dieselbe Wirkung wie getAttribute(), gilt allerdings nur für Stilattribute
2. setProperty() hat dieselbe Wirkung wie setAttribute(), gilt allerdings nur für Stilattribute

Natürlich kann das DOM auch modifiziert werden. Hier wird dann mit Methoden wie removeChild(), replaceChild() oder createElement() gearbeitet.

4.4 Animationen in Javascript

Javascript kann aber nicht nur Interaktivität ermöglichen, sondern Objekte können in Javascript auch animiert werden. Grundlage dafür sind zwei Funktionen:

- setTimeout (function,duration): rekursiver Aufruf der Funktion nach einer Dauer:

```
function animate(evt) //function wird bei durch Bedingung
beendet
{
  if (condition) {}; //SVG-Objekte werden verändert
  setTimeout("animate()",duration) //rekursiver
  Aufruf der function nach einer Dauer
}
```

- setInterval(function,duration): die Funktion wird nach einer Dauer neu aufgerufen:

```
function  animate(evt)
{
  setInterval("rebuild()",duration) // function wird nach
einer Dauer neu
  aufgerufen
}
function rebuild(evt) {} //SVG-Objekte werden verändert
```

Beispiel: Der Radius eines Kreises wächst von 100 auf 200 an:

```
<svg>
  <defs>
  <script type="text/javascript">
  <![CDATA[
     function tim(evt)
    {
      svgdoc = evt.getTarget().getOwnerDocument();
      action();
    }
        function action()
    {
      kreis=svgdoc.getElementById("hans");
      var radius = kreis.getAttribute('r');
      if(radius < 200)
      {
        kreis.setAttribute('r', parseFloat(radius) + 0.2);
     setInterval
     setTimeout("action()",5);
      }
    }
        ]]>
  </script>
```

```
  </defs>
    <circle id="hans" cx="300" cy="300" r="100" fill="green"
     onload="tim(evt)"/>
</svg>
```

Schwierig wird es, wenn mit SVG komplexe Mechanismen dargestellt werden sollen. Denn dahinter stecken komplexe mathematische Formeln, die nur mit Javascript realisiert werden können. Wenn man jetzt Vor- und Nachteile des Animierens in SVG und in Javascript aufzeigen wollte, kann man sagen, dass einfache Animationen schnell und einfach in SVG zu programmieren sind. In Javascript wird für solche Animationen eine erheblich höhere Rechnerleistung benötigt, was zu ungleichmäßigen Abläufen führen kann. Haben wir jedoch ein komplexe Animation vorliegen, die mehrere Bestandteile hat, die wiederum mehrere verschiedene Bewegungsrichtungen haben und die sich trotzdem logisch zueinander bewegen sollen (Bsp. Dreigelenkgetriebe oder ähnliches), dann nimmt das Programmieren schon ganz andere Dimensionen an. Und hier ist Javascript unausweichlich, nicht zuletzt, weil für die einzelnen Zustände der Elemente immer wieder neue Koordinatenpunkte berechnet werden müssen.

5 Vorgehensweise bei der Programmierung einer SVG-Animation

Im Folgenden wird eine beispielhafte Vorgehensweise zur Programmierung einer SVG-Animation vorgestellt. Sie besteht aus drei Teilen: Dokumentenkopf, Definitionsbereich und Hauptteil. Das Beispiel besteht aus einem Gabelstapler mit Fahrer, der eine Palette aus einem Regal holen und sie anschließend auf dem Boden absetzten soll. Doch zuvor ein paar Grundlagen hinsichtlich Syntax und Koordinatensystem.
Syntaktische Besonderheiten von SVG gegenüber HTML:

- SVG unterschiedet zwischen Groß- und Kleinschreibung; Tags, Attribute und deren vordefinierten (!) Werte werden vollständig klein geschrieben.
- Zu jedem öffnenden Tag muß ein schließendes gesetzt werden. Tags, die keinen Body haben, sprich keinen textuellen Inhalt zwischen öffnenden und schließenden, können auch mit einem / am Ende geschlossen werden (Bsp: <rect/>)
- Attributwerte werden generell von Anführungszeichen umschlossen.
- Kommentare sind genau wie in HTML mit <!-- --> umfasst.
- Positionierungen durch Attribute werden nicht durch top und left definiert, sondern x und y

Der Nullpunkt des Koordinatensystem liegt wie bei den meisten Grafikformaten links oben. Nun zum Beispiel:

1.)

```
 <!-- Zuerst muss die Sprache des Dokuments (XML), die Version
(1.0) und, ob es sich um ein eigenständiges Dokument oder um
eingebetteten Code handelt. -->
<?xml version="1.0" standalone="yes"?>
<!-- Angabe der korrekten DTD (hier SVG Version 1.1)-->
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
"http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<!-- Das alles umschließende SVG Tag mit Versions- und
Namespace-Definition -->
<svg width="500px" height="400px" version="1.1"
xmlns="http://www.w3.org/2000/svg">
  <!-- Hier erfolgt dann die SVG-Definition -->
</svg>
```

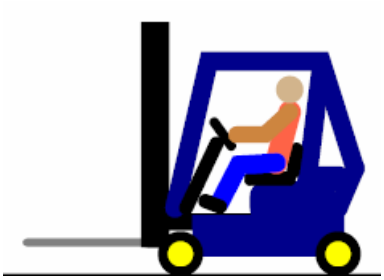
2.)
Als nächstes werden im Definitionsbereich Objekte definiert und zu Gruppen zusammengefasst. Dies hat den Vorteil, dass die zu einer Gruppe zusammengefassten Objekte alle die gleiche Animation ausführen, daher nur einmal aufgerufen werden müssen und der Code dadurch übersichtlicher wird. Beispielhaft wird im Folgenden die Gruppe “Palette“ über das Tag “g“ beschrieben:

```
<g id="palett">
<line x1="0" y1="0" x2="90" y2="0" stroke-width="5"
stroke="darkkhaki"/>
<rect x="0" y="0" width="15" height="11" fill="darkkhaki"/>
```

```
<rect x="36.5" y="0" width="15" height="11" fill="darkkhaki"/>
<rect x="75" y="0" width="15" height="11" fill="darkkhaki"/>
</g>
```

Als erstes wird im <g>-Tag dem Objekt eine id zugeordnet, die es identifiziert. In diesem Beispiel besteht die Palette aus 1 Linie und 3 Rechtecken.

Im Definitionsbereich werden darüber hinaus der Gabelstapler und das Regal definiert.

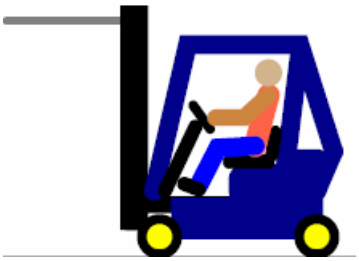


3.) Außerhalb des Definitionsbereichs rufen wir nun über das <use> - Tag die oben definierten Objekte auf und animieren diese oder ihre Untergruppen, wenn sie dazu bestimmt sind. Wie man beim Ablauf der Animation erkennen kann, setzt sich zuerst der Gabelstapler in Bewegung. Dies geschieht mit folgenden Zeilen:

```
<use xlink:href="#forklift_truck" x="800" y="503">
<animate attributeName="x" begin="0.1s" dur="3s"
 calcMode="spline" values="800;280"
 keySplines="0.28 0.3 0.46 1" fill="freeze"/>
```

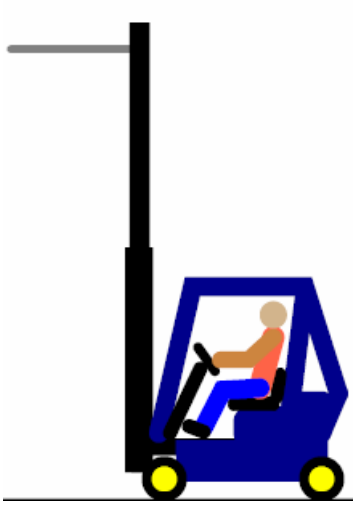
Nachdem wir über use xlink:href unseren Gabelstapler aufgerufen haben, referenzieren wir im <animate>-Tag mit attributeName auf die x-Position des Staplers und verändern diesen Wert auf 280. Nun kommt der Begriff Interpolation ins Spiel, d.h. die Bewegung von einem zu einem anderen Zustand kann manipuliert. Wie wir schon erfahren haben, gibt es dazu verschiedene Modi. Im Modus spline kann man mit Eingabe von keysplines sogar eine Beschleunigung und Verzögerung erzeugen, da die Punkte eine Bézierkurve festlegen und die Animation anhand dieser interpoliert wird. Anschließend möchten wir ja wie im Realen, das der Mast mit der Gabel ausfährt und die Palette aus dem Regal holt. Dazu werden zwei Untergruppen des Staplers, der mehrteilige Mast und die Gabel, animiert:

```
<animate xlink:href="#fork" attributeName="y" from="200"
 to="10" begin="0.1s" dur="3s" fill="freeze"/>
```



Da wir über use den Gruppenkopf schon aufgerufen haben, müssen wir nun die gewünschte Untergruppe über deren id aufrufen und die gewünschte Variable animieren. Hier im Beispiel wird natürlich y angesprochen, dass wir uns ja die Höhe bewegen wollen.

```
 <animate xlink:href="#mast_2" attributeName="y" from="0"
to="-200" begin="3.8s" dur="3s" fill="freeze"/>
<animate xlink:href="#fork" attributeName="y" from="10" to="-
180" begin="3.8s" dur="3s" fill="freeze"/>
```



Nun müssen die einzelnen Elemente noch mit begin und dur zeitlich aufeinander abgestimmt werden. Dazu kommt natürlich auch später die Palette, die aus dem Regal entnommen wird, und die wir als eigenständige Gruppe separat über ein <use>-Tag aufrufen.

Der Code:

```
<?xml version="1.0"?>
<svg xmlns="http://www.w3.org/2000/svg"
xmlns:xlink="http://www.w3.org/1999/xlink"
width="100%" height="900">
<defs>
<g id="forklift_truck">
<rect y="170" x="10" width="140" height="15" fill="black"/>
<rect id="fork" y="200" x="-115" width="115" height="7"
fill="gray" rx="5"/>
<rect id="mast" width="25" height="200" y="0" x="-10"
fill="black"/>
<rect id="mast_1" width="14" height="200" y="0" x="-4"
fill="black"/>
<rect id="mast_2" width="18" height="200" y="0" x="-6"
fill="black"/>
<g id="frame" stroke="darkblue" stroke-linecap="round" stroke-
width="17">
<rect y="180" x="44" width="118" height="20" fill="darkblue"/>
<rect x="95" y="150" fill="darkblue" width="60" height="20"
rx="5"/>
<line x1="165" y1="165" x2="180" y2="130" />
<line x1="140" y1="185" x2="165" y2="165"/>
<line x1="132" y1="185" x2="150" y2="40"/>
<line x1="140" y1="139" x2="165" y2="139" stroke-width="22"/>
<polyline points="180 130, 150 35, 50 35, 20 165"
fill="none"/>
</g>
<g id="seat" stroke="black" fill="black">
<rect x="82" y="133" width="45" height="12" rx="5"/>
<line x1="125" y1="133" x2="128" y2="113" stroke-width="12"
stroke-linecap="round"/>
</g>
<g id="steering_wheel" stroke="black" stroke-linecap="round"
stroke-width="13">
<line x1="30" y1="165" x2="57" y2="108" />
<line x1="55" y1="89" x2="70" y2="110" stroke-width="9"/>
</g>
<g id="frontwheel">
<circle r="20" cx="25" cy="206"/>
<circle r="11" cx="25" cy="206" fill="yellow"/>
</g>
<g id="backwheel" fill="yellow">
<circle r="20"  fill="black" cx="165" cy="206"/>
<circle r="11"  cx="165" cy="206"/>
</g>
<g id="pilot" stroke-linecap="round" stroke-width="20"
stroke="peru">
<polyline id="body" points="110 125, 122 80" fill="none"
stroke="tomato"/>
```

```
<polyline id="arms" points="122 80,100 100,75 100" fill="none"
stroke-width="15"/>
<polyline id="trousers" points="110 125, 77 127,60 160"
fill="none" stroke-width="17" stroke="blue"/>
<polyline id="shoes" points="60 164,47 159" fill="none"
stroke="black" stroke-width="13"/>
<circle id="head" cx="122" cy="60" r="12" fill="tan" stroke-
width="0"/>
</g>
</g>
<g id="depot">
<line x1="0" y1="0" x2="0" y2="400" stroke-width="13"
stroke="gray"/>
<line x1="85" y1="0" x2="85" y2="400" stroke-width="13"
stroke="gray"/>
<line id="lagerplatz_4" x1="0" y1="35" x2="85" y2="35" stroke-
width="6" stroke="darkgray"/>
<line id="lagerplatz_3" x1="0" y1="120" x2="85" y2="120"
stroke-width="6" stroke="darkgray"/>
<line id="lagerplatz_2" x1="0" y1="220" x2="85" y2="220"
stroke-width="6" stroke="darkgray"/>
<line id="lagerplatz_1" x1="0" y1="320" x2="85" y2="320"
stroke-width="6" stroke="darkgray"/>
<polyline id="strutting" points="85 0,0 35, 85 85,0 120,80
170,0 220,80 270,0 320,80 370,0 399"
fill="none" stroke-width="2" stroke="dimgray"/>
<line id="boden" x1="-10" y1="400" x2="1000" y2="400" stroke-
width="3" stroke="black"/>
</g>
<g id="palett">
<line x1="0" y1="0" x2="90" y2="0" stroke-width="5"
stroke="darkkhaki"/>
<rect x="0" y="0" width="15" height="11" fill="darkkhaki"/>
<rect x="36.5" y="0" width="15" height="11" fill="darkkhaki"/>
<rect x="75" y="0" width="15" height="11" fill="darkkhaki"/>
</g>
</defs>
<use xlink:href="#forklift_truck" x="800" y="503"
transform="scale(.7)">
<animate attributeName="x" begin="0.1s" dur="3s"
calcMode="spline" values="800;280"
keySplines="0.28 0.3 0.46 1" fill="freeze"/>
<animate xlink:href="#fork" attributeName="y" from="200"
to="10" begin="0.1s" dur="3s" fill="freeze"/>
<animate xlink:href="#mast_2" attributeName="y" from="0" to="-
200" begin="3.8s" dur="3s"
fill="freeze"/>
<animate xlink:href="#fork" attributeName="y" from="10" to="-
180" begin="3.8s" dur="3s" fill="freeze"/>
<animate xlink:href="#mast_1" attributeName="y" from="-200"
to="-380" begin="7s" dur="3s"
fill="freeze"/>
```

```
<animate xlink:href="#fork" attributeName="y" from="-180"
to="-370" begin="7s" dur="3s" fill="freeze"/>
<animate attributeName="x" begin="9s" dur="2s" from="280"
to="340" fill="freeze"/>
<animate xlink:href="#mast_1" attributeName="y" begin="11s"
dur="2s" from="-380" to="-326"
fill="freeze"/>
<animate xlink:href="#fork" attributeName="y" from="-370"
to="-306" begin="11s" dur="2s"
fill="freeze"/>
<animate attributeName="x" begin="13s" dur="4s"
calcMode="spline" values="340;150"
keySplines="0.28 0.3 0.46 1" fill="freeze"/>
<animate xlink:href="#mast_1" attributeName="y" begin="18s"
dur="1s" from="-326" to="-332"
fill="freeze"/>
<animate xlink:href="#fork" attributeName="y" from="-306"
to="-312" begin="18s" dur="1s"
fill="freeze"/>
<animate xlink:href="#mast_1" attributeName="y" begin="20.2s"
dur="1.5s" from="-332" to="-338"
fill="freeze"/>
<animate xlink:href="#fork" attributeName="y" from="-312"
to="-318" begin="20.2s" dur="1.5s"
fill="freeze"/>
<animate attributeName="x" begin="22s" dur="3s"
calcMode="spline" values="150;300"
keySplines="0.66 0 0.67 1" fill="freeze"/>
<animate xlink:href="#mast_1" attributeName="y" begin="25s"
dur="3s" from="-338" to="-200"
fill="freeze"/>
<animate xlink:href="#fork" attributeName="y" from="-318"
to="-190" begin="25s" dur="3s"
fill="freeze"/>
<animate xlink:href="#mast_2" attributeName="y" from="-200"
to="0" begin="28.2s" dur="3s"
fill="freeze"/>
<set xlink:href="#mast_1" attributeName="y" to="0" begin="28s"
fill="freeze"/>
<animate xlink:href="#fork" attributeName="y" from="-190"
to="10" begin="28.2s" dur="3s"
fill="freeze"/>
<animate xlink:href="#fork" attributeName="y" from="10"
to="180" begin="31.2s" dur="2s" fill="freeze"/>
<animate attributeName="x" begin="33.6s" dur="7s"
calcMode="spline" values="300;800"
keySplines="0.66 0 0.67 1" fill="freeze"/>
</use>
<use xlink:href="#palett" x="7" y="132">
<animate attributeName="y" from="132" to="128" begin="20.2s"
dur="1.5s" fill="freeze"/>
<animate attributeName="x" from="7" to="112" begin="22"
dur="3s" calcMode="spline" values="7;112"
```

```
keySplines="0.66 0 0.67 1" fill="freeze"/>
<animate attributeName="y" from="128" to="217.6" begin="25s"
dur="3s" fill="freeze"/>
<animate attributeName="y" from="217.6" to="357.6"
begin="28.2s" dur="3s" fill="freeze"/>
<animate attributeName="y" from="357.6" to="476.6"
begin="31.2s" dur="2s" fill="freeze"/>
<animate attributeName="x" begin="33.6s" dur="7s"
calcMode="spline" values="112;462"
keySplines="0.66 0 0.67 1" fill="freeze"/>
</use>
<use xlink:href="#depot" x="10" y="110"/>
</svg>
```

Quellen:

http://svglbc.datenverdrahten.de/
http://www.fh-fedel.de/~si/praktika/MultimediaProjekte/SVG/SVG_Tutorial_mi3794/index.htm
http://www.selfsvg.info
http://svg.tutorial.aptico.de
http://www.w3.org
http://www.schumacher-netz.de/TR/1999/REC-xml-names-19990114-de.html
http://www.linkwerk.com/pub/xmlidp/2000/quick-start-dtd.html
http://www.svgopen.org/2002/material/ws_animation_filters/diapos_animation/title_0.svg
http://svglbc.datenverdrahten.de/