4 Zusammengesetzte *SVG*-Objekte

Nachdem Sie sich mit einfachen *SVG*-Elementen befasst haben, soll alles komplizierter werden. Die Kenntnisse aus Kapitel 3 werden in diesem Kapitel angewendet und festigen somit die bisher gewonnen Kenntnisse über Klassen-Strukturen. Desweiteren werden die Konvertierungen mit *parse* angewendet und Sie lernen u. a. das wichtige *array*-Element kennen.

Learning by doing ist auch hier das Motto zum Erlernen der neuen Begriffe und Strukturen. Auch wenn schwierige Themen wie der *trend* mit einem *path* nicht so schnell vollständig verstanden werden, sollten Sie wissen, dass in den folgenden Kapiteln jeweils immer automatisch eine Wiederholung der wichtigen Algorithmen stattfindet. Sie können so beruhigt an das nächste Kapitel denken, wenn Sie dieses Kapitel so einigermaßen verstanden haben.

4.1 Das *Polygon und* der Text

Mit den Grundkenntnissen aus dem vorangegangenen Kapitel sind wir in der Lage *SVG-Objekte* zu zeichnen. Die erstellten *SVG-Basic*-Elemente können zu komplexen *SVG*-Objekten zusammengesetzt werden. Das soll nun in diesem Kapitel vorgestellt werden, um damit auch weitere *JS*-Kenntnisse zu sammeln. Die *JS*-Dateien sind nun in drei verschiedene Dateien aufgeteilt und befinden sich unter folgenden Bezeichnung:

..//Buch_DVD/Kapitel_4.1_SVG_Basic.js

..//Buch_DVD/Kapitel_4.1_SVG_Multi.js

..//Buch_DVD/Kapitel_4.1_SVG_main.js

Die Startdatei für den *Browser:*

..//Buch_DVD/Kapitel_4.1_Index.html

Unsere Startdatei *..index.html* beinhaltet nun drei Dateien. Die Zeile 11(**Bild 4.1**) ist uns aus den vorangegangenen Kapiteln bereits bekannt. Aus der Datei *Kapitel_4.1_Basic.js* habe ich die Klasse *C_Main* entfernt und dafür eine eigene Datei *Kapitel_4.1_SVG_Main.js* angelegt (Zeile 13). Die Datei in Zeile 12 ist neu und soll nun in diesem Kapitel behandelt werden. Der *Browser* muss nun drei Dateien laden und geht dabei in der angegebenen Reihenfolge entsprechend der Zeilen 11 bis 13 vor.

Alle **Konstanten**, globale **Variablen** und **Klassen** aus der ersten Datei sind demnach in den darauffolgenden Dateien bekannt und können ohne weitere Maßnahmen von den folgenden Dateien angewendet werden.

```
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">

<head>
    <meta charset="utf-8" />
    <title> Web-PCE </title>
</head>

<body>
    <script src="Kapitel_4.1_SVG_Basic.js"> </script>
    <script src="Kapitel_4.1_SVG_Multi.js"> </script>
    <script src="Kapitel_4.1_main.js"> </script>
</body>

</html>
```

Bild 4.1: Die Startdatei für den Browser hat nun drei JavaScript-Dateien zu laden

Man muss bei dieser Vorgehensweise nicht mit *include*-Anweisungen arbeiten, so wie das bei einigen Programmiersprachen der Fall ist. Unser erstes, zusammengesetztes *SVG*-Objekt soll ein Ventil darstellen, so wie man es in der Visualisierung für chemische Prozesse zu sehen bekommt. Das ist ein schönes Beispiel, denn hier kommen die uns bereits bekannten Basis-Objekte*, SVG-Text*, *SVG_Line* und ein *SVG_Rect* zur Anwendung. Neu ist das *SVG-Polygon* für die Darstellung des Ventils.

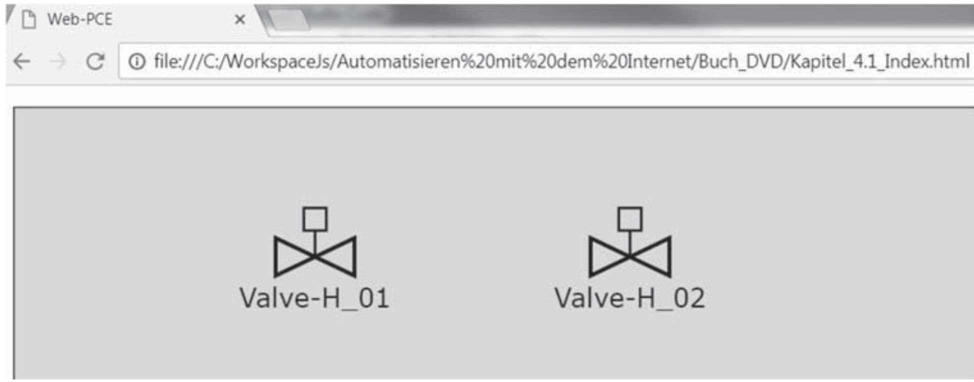

Bild 4.2: Das Ventil

In der Datei *Kapitel_4.1_SVG_Basic.js* wurde die Konstante *const VALVE = 5;* definiert (nicht im Bild).

Zudem werden wir uns ein wenig mit der *string*-Verarbeitung beschäftigen müssen, denn das *Polygon* verwendet als Parameter eines *string* mit den Koordinaten der einzelnen Punkte.

Das Ergebnis ist in **Bild 4.2** (stark vergrößert) zu sehen. Was so einfach aussieht, muss jedoch zur Erklärung in einzelne Abschnitte zerlegt werden. Deswegen beginnen wir mit dem *main* (**Bild 4.3**), das aus Sicht des Anwenders meist der erste Bezugspunkt ist.

```
1   // main zu SVG-Elemente
2   'use strict'
3   window.onerror = main_OnError;
4
5   class C_Main {
6
7     constructor() {
8
9       this.layer = new C_InitLayer(LAYER_STROKE, LAYER_BACK_GROUND).GetLayer();
10      this.valve_01 = new C_Valve(
11        this.layer,
12        100, 50,
13        30,
14        "Valve-H_01");
15
16      this.valve_02 = new C_Valve(
17        this.layer,
18        this.valve_01.Get_XPos() + this.valve_01.Get_Dx() * 4, 50,
19        this.valve_01.Get_Dx(),
20        "Valve-H_02");
21    }
22  }
23
24  var main = new C_Main();
25
26  // ***** OnError
27  function main_OnError(p1, p2, p3) {
28    alert("Sorry: Error on exception in ...main.js\n"
29      + p1 + "\n" + p2 + "\n" + p3);
30    return true;
31  }
```

Bild 4.3: Das main

In den Kopfzeilen (Zeile 1 bis 4) sehen wir die *Error*-Behandlung, welche im Falle eines Fehlers an das Ende der Datei zur Funktion *main_OnError* verzweigen soll, so wie wir das bereits in den gezeigten Dateien umgesetzt haben. Die Klasse *C_Main* erzeugt in Zeile 9 den Layer und soll danach zwei Ventile abbilden.

In Zeile 9 wird die Klasse *C_InitLayer* instantiiert und danach mit dem Punktoperator die *Methode* ***GetLayer()*** angefügt. Es ist darauf zu achten, dass diese *Methode* nicht vergessen werden darf (ein häufiger Fehler). Darauf soll noch einmal hingewiesen werden.

In Zeile 10 wird das erste Ventil (*Valve-H_01*) kreiert. Der Konstruktor wird mit den Daten versorgt,damit das Ventil mit 30 Pixel Länge an der Position X=100, Y=50 Pixel gezeichnet wird. Warum das so vorgegeben werden muss, sehen wir etwas später am entsprechenden Konstruktor zur Ventil-Klasse.

Das zweite Ventil dient zur Kontrolle der abgeleiteten *Methoden* (*Get_XPos* etc.), welche vom ersten Ventil genutzt werden. Damit haben wir ein erstes, praktisches Beispiel für die Anwendung der abgeleiteten *Methoden* aus ***Kapitel 3.3.***

In Zeile 18 errechnet sich der zweite Parameter für die X-Position zum Ventil *Valve-H_02*, welches mit einem Abstand der 4-fachen Länge vom ersten Ventil gezeichnet werden soll. Die Länge des Ventilkörpers für den dritten Parameter wird in Zeile 19 vom ersten Ventil abgeleitet. Wie im **Bild 4.2** ersichtlich, funktioniert das schon einmal wie geplant.

In Zeile 24 wird schließlich die Instanz zum *main* gebildet, damit das Ganze auch auf dem *Browser* sichtbar wird. Aus Sicht der Anwendung (*main*) für Programmierer also relativ einfach, da Klassen mit den durch den Konstruktor vorgegebenen Parametern angewendet werden. Jetzt wird auch ein wenig klar, warum Konstruktoren mit vielen Parametern nicht so sinnvoll sind und man sich das Nachschlagen im Konstruktor ersparen kann.

Wie entsteht nun tatsächlich dieses zusammengesetzte *SVG*-Element?

```
 5  // *****
 6  class C_Valve extends C_SVG_Object {
 7
 8    constructor(layer, x, y, width, text) {
 9      super();
10      this.layer = layer;
11      this.width = width;
12      this.x = x;
13      this.y = y;
14      this.heightPolygon = this.width/2.1;
15      this.frameHeight = this.width/3.5;
16      this.frameWidth = this.frameHeight;
17      this.xPosFrame = this.x + this.width/2 - this.frameWidth/2;
18      this.yPosFrame = this.y - this.heightPolygon/2 - this.frameHeight/2;
19      this.statusColor = STATUS_COLOR;
20      this.valveName = text;
21      this.textHeight = 10;
22      this.stroke = LAYER_STROKE;
23      this.fill = LAYER_BACK_GROUND;
24
25      // polygon
26      this.doc = document.createElementNS(SVG_SPEC, 'polygon');
27      this.doc.setAttribute(
28        "points",
29        this.x.toString() + ',' + this.y.toString() + ' ' +
30        (this.x + this.width).toString() + ',' +
31        (this.y + this.heightPolygon).toString() + ' ' +
32        (this.x + this.width).toString() + ',' + this.y.toString() + ' ' +
33        this.x.toString() + ',' + (this.y + this.heightPolygon).toString()
34      );
35      this.AppendObject();
36      // alert(this.doc.getAttribute("points"));
```

Bild 4.4: Der Konstruktor mit dem Polygon

In **Bild 4.4** ist der erste Abschnitt der Datei *Kapitel_4.1_SVG_Multi.js* mit der Klassenbildung für das Ventil (Zeile 6) zu sehen. Die Parameter zum Konstruktor (Zeile 8) erklären jetzt die Aufruf-Optionen aus dem *main*-Abschnitt für die beiden Ventile. Zur Darstellung des Ventils werden alle Parameter aus der Ventilbreite *this.witdh* abgeleitet, damit bei einer Vergrößerung oder Reduzierung des Ventilkörpers die Verhältnisse gleich bleiben (Zeilen 14 bis 18). Die Textgröße ist hier nicht berücksichtigt, da dieser immer gut lesbar sein soll, wird diese in Zeile 21 mit 10 *hardcodiert* und muss bei einer Größenänderung gegebenenfalls noch korrigiert werden.

Hardcodierte Ausdrücke verstoßen oft gegen einen festgelegten *Code Style*. In diesem Fall könnte man die Texthöhe 10 über eine Konstante festlegen. Das ist für viele Programmierer auch eine Art *Hardcodierung*, denn auch diese müsste man bei einer Änderung neu definieren. Sie können auch ein *Attribute* festlegen, wie *this.textHoehe = 10;* So kann das *Attribute* dann in der Programmanwendung verändert werden.

Die Position des Ventils wird mit *this.x* und *this.y* (Zeilen 12, 13) auf den ersten Parameter des Polygons bezogen.

Aufbauend auf den Ventilkörper über ein *Polygon* gezeichnet, wird danach die senkrechte Linie und das Rechteck darüber erzeugt. Zum Schluss erfolgt der Text, dessen Textparameter in Bezug auf die Ventilbreite und Ventilhöhe nicht berücksichtigt sind. Das ist allgemein bei Texten so, welche sich im *Browser* durch die Eingabe ändern können.

Außerdem ist es praktischer, da der Text nicht zur aktiven Visualisierung gehört. Der Ventilkörper dagegen ja, denn der soll entsprechend seines Betriebszustands später einmal die Farbe ändern. Der Text bleibt dabei vorerst üblicherweise unberücksichtigt.

Betrachten wir zuerst die Erstellung des Ventilkörpers über das *Polygon* und später im Verlauf dieses Kapitels erst den Rest mit der Linie, dem Rechteck und dem Text.

In Zeile 26 wird mit *createElementNS* das *Polygon* mit dem Parameter *'polygon'* gebildet und in *this.doc* gespeichert.

Man könnte den Ventilkörper auch durch vier Linien darstellen. Allerdings wäre es dann im späteren Verlauf sehr kompliziert, die Innenflächen des Ventilkörpers farblich an den Betriebszustand des Ventils anzupassen.

Das *Polygon* dagegen besitzt die Eigenschaft ***fill*** und kann so den Betriebszustand elegant anzeigen. Danach erfolgt die Zuweisung der *Attribute* für das *Polygon* (Zeile 27) und durch den Aufruf der abgeleiteten *Methode AppendObject* werden die *fill*- und *stroke-Attribute* vorerst als *default*-Werte gesetzt. Die Daten für das Polygon werden als ***string*** in das *Attribute "**points**"* übergeben (Zeilen 29 bis 33).

Was bedeutet nun „points“ in Zeile 28?

Das Format für ein *Polygon* zum *Attribute points* ist ein *string* und beinhaltet die Koordinatenpaare (X,Y) durch ein Komma getrennt, wobei jedes Koordinatenpaar durch eine Leerstelle getrennt wird. Nach der letzten Koordinate schließt das *Polygon* automatisch zum Startpunkt.

Ein Beispiel dazu: ***"100,100 200,100 200,200 100,200"*** würde als *Polygon* ein Quadrat mit jeweils 100 Pixel Kantenlänge zeichnen. Das Quadrat beginnt an der Position X=100 und Y=100. Der nächste Punkt (waagerechte Linie nach rechts) ist X=200 und gleicher Y-Achse mit Y=100. Dann folgt die Senkrechte (rechte Linie) mit X=200 und Y=200. Die waagerechte Linie nach links folgt dann mit X=100 und Y=200. Automatisch schließt dann das *Polygon* zum Startpunkt mit X=100 und Y=100.

Da wir nun für die Koordinaten keine Texte (*string*) sondern Nummern zur Verfügung haben, müssen wir die Nummern in Texte wandeln. In Zeile 29 wird die Startposition des *Polygons* mit *this.x**.toString()*** gesetzt.

Der *Zahlenwert* von *this.x* kann durch die *Methode* **toString()** und dem Punkt-Operator in einen ***string*** gewandelt werden. Nicht vergessen, die *Methode toString()* mit der runden Klammer anzugeben, sonst wird diese nicht aufgerufen!

So wird z. B. aus der Zahl 125 mit ***125.toString()***, der *string* “125“ erzeugt. Danach wird ein Komma zur Trennung der Koordinaten benötigt. Das erfolgt mit dem Plus-Operator folgendermaßen: ***125.toString()* + *“,“*** ergibt den String **“125,“**.

Ein String kann mit dem „+“-Operator erweitert werden. Beispiel: “Name“ + “ Vorname“ + “ Alter: “ + *alter.toString()* ergibt den String „“Name Vorname Alter: 53“. Mit der Annahme, dass die Variable *alter* den numerischen Wert 53 beinhaltet.

In Zeile29 wird mit ***+this.y.toString()*** die Y-Koordinate in einen *string* gewandelt und mit dem „plus“ zum bestehenden *String* addiert. Das erste Koordinatenpaar ist somit fertig und es kann durch die Addition der Leerstelle mit **+“ “** an den bestehenden *string* das nächste Koordinatenpaar angehängt werden.

Der Ventilkörper besteht also aus dem Startpunkt (Zeile 29), dann folgend die erste Linie nach rechts-unten (Zeile 30 und 31), danach die senkrechte Linie nach oben (Zeile 32) und dann die Linie nach links-unten (Zeile 33). Fertig ist das *Polygon*, denn es schließt sich selbstständig zum Startpunkt.

Zum besseren Verständnis habe ich in Zeile 36 den Kommentar eingegeben. Wenn Sie den freigeben, dann erhalten Sie das Ergebnis des *Attributes “points“* am Browser für beide Ventile ausgegeben. Testen Sie das mal!

Jetzt kommen wir zum nächsten Punkt, der noch geklärt werden muss. Sie erinnern sich an das *switch-case-Statement* aus dem vorherigen **Kapitel 3.3**?

Wir werden noch mehrere *SVG*-Objekte entwickeln, welche in der Ableitung überwiegend das gleiche Ergebnis mittels der *case*-Anweisung haben werden. Es wäre deswegen besser die *switch-case*-Struktur so zu ändern, dass nur die Ausnahme eines Objekts gesondert behandelt wird. Dazu betrachten wir das Listing aus *Kapitel_4.1_SVG_Basis.js*, welches nun angepasst wurde. In **Bild 4.5** ist die abgeleitete *Methode AppendObject* noch einmal zu sehen (Zeile 61).

```
60    // Methods
61    AppendObject() {
62      this.doc.setAttribute("stroke", this.stroke);
63      this.doc.setAttribute("stroke-width", 1);
64      this.doc.setAttribute("stroke-opacity", 1.0);
65      switch (this.type) {
66        case LINE:
67          break;
68        default:
69          this.doc.setAttribute("fill", this.fill);
70          this.doc.setAttribute("fill-opacity", 1.0);
71      }
72      this.layer.appendChild(this.doc);
73    }
```

Bild 4.5: Die Methode AppendObject mit verbesserter switch-case-Struktur

Neu hinzugekommen ist das Schlüsselwort ***default*** in Zeile 68. Nun verzweigt das Programm in Zeile 65 mit der Anweisung *switch(this.type)* bei *type=LINE* nur noch in den *case*-Abschnitt *LINE,* ansonsten bei allen anderen Typen nach *default* (Zeile 68). Das hat