Schnuppern Sie rein in die Webentwicklung der Zukunft

Durchstarten mit HTML5

Mark Pilgrim

Deutsche Übersetzung von Lars Schulten

O'REILLY®

KAPITEL 4

Zeichenstunde

Einstieg

HTML5 definiert das <canvas>-Element (*http://bit.ly/9JHzOf*) als »eine auflösungsabhängige Bitmap-Leinwand zur unmittelbaren Zeichnung von Diagrammen, Spielgrafiken und anderen visuellen Bildern«. Ein Canvas ist ein Rechteck in Ihrer Seite, in das Sie mit JavaScript nach Belieben zeichnen können. Die folgende Tabelle zeigt, welche Browser eine elementare Canvas-Unterstützung boten, als dies geschrieben wurde:

IE[1]	Firefox	Safari	Chrome	Opera	iPhone	Android
7.0+	3.0+	3.0+	3.0+	10.0+	1.0+	1.0+

Wie also sieht ein Canvas aus? Nach nichts eigentlich. Ein <canvas>-Element an sich hat keinen Inhalt oder Rahmen. Das Markup sieht so aus:

```
<canvas width="300" height="225"></canvas>
```

Abbildung 4-1 zeigt das Canvas mit einem gepunkteten Rahmen, damit Sie sehen können, wovon wir reden.

Eine Seite kann mehrere <canvas>-Elemente enthalten. Jedes Canvas erscheint im DOM, und jedes Canvas hat seinen eigenen Zustand. Wenn Sie jedem Canvas ein id-Attribut geben, können Sie darauf zugreifen wie auf jedes andere Element in einer Seite.

Erweitern wir das Markup um ein id-Attribut:

```
<canvas id="a" width="300" height="225"></canvas>
```

Jetzt ist es ein Kinderspiel, das <canvas>-Element im DOM zu finden:

```
var a_canvas = document.getElementById("a");
```

1 Der Internet Explorer benötigt die externe Bibliothek explorercanvas.

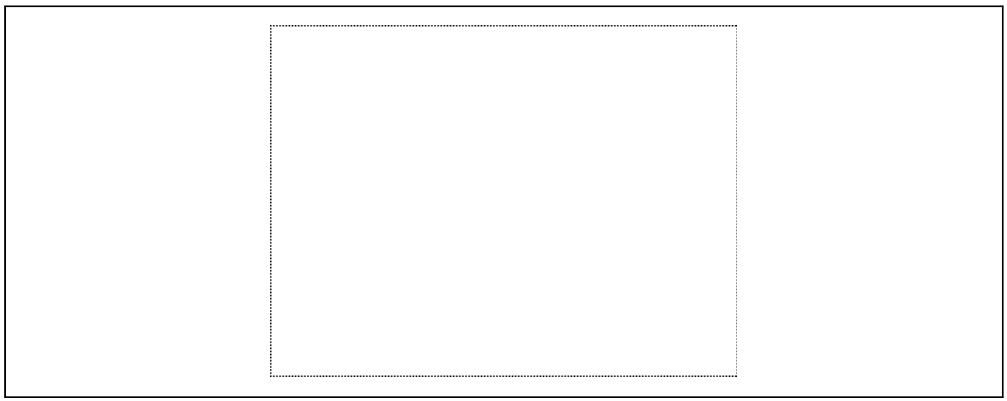

Abbildung 4-1: Canvas mit Rahmen

Einfache Figuren

IE[2]	Firefox	Safari	Chrome	Opera	iPhone	Android
7.0+	3.0+	3.0+	3.0+	10.0+	1.0+	1.0+

Zu Beginn ist ein Canvas leer. Wie langweilig! Schaffen wir Abhilfe. Zeichnen wir etwas. Sie können den `onclick`-Handler einsetzen, um eine Funktion aufzurufen, die ein Rechteck zeichnet (*http://diveintohtml5.org/canvas.html* präsentiert Ihnen ein interaktives Beispiel):

```
function draw_b() {
  var b_canvas = document.getElementById("b");
  var b_context = b_canvas.getContext("2d");
  b_context.fillRect(50, 25, 150, 100);
}
```

Die erste Zeile der Funktion ist nichts Besonderes. Sie sucht im DOM nach dem `<canvas>`-Element. In der zweiten Zeile wird die Angelegenheit schon interessanter. Ein Canvas hat einen Zeichenkontext (Drawing-Kontext), und das ist der Ort, an dem all die interessanten Dinge geschehen. Haben Sie das `<canvas>`-Element (über `document.getElementById()` oder eine beliebige andere Methode) im DOM gefunden, können Sie seine `getContext()`-Methode aufrufen. Sie müssen dabei den String `"2d"` an `getContext()` übergeben:

```
function draw_b() {
  var b_canvas = document.getElementById("b");
  var b_context = b_canvas.getContext("2d");
  b_context.fillRect(50, 25, 150, 100);
}
```

2 Der Internet Explorer benötigt die externe Bibliothek explorercanvas.

Fragen an Professor Markup

F: Gibt es ein 3-D-Canvas?

A: Noch nicht. Einzelne Hersteller haben mit eigenen dreidimensionalen Canvas-APIs experimentiert, aber keine wurde standardisiert. Die HTML5-Spezifikation sagt dazu: »Eine zukünftige Version dieser Spezifikation wird wahrscheinlich einen 3-D-Kontext definieren.«

Sie haben also ein `<canvas>`-Element und seinen Zeichenkontext. Der Zeichenkontext ist das, worin alle Zeichenmethoden und Eigenschaften definiert sind. Es gibt eine große Menge an Eigenschaften und Methoden, die dem Zeichnen von Rechtecken gewidmet sind:

- Die `fillStyle`-Eigenschaft kann eine CSS-Farbe, ein Muster oder ein Verlauf sein. (Mehr zu Verläufen in Kürze.) Der Standard für `fillStyle` ist Schwarz, aber Sie können die Eigenschaft auf einen beliebigen Wert setzen. Jeder Zeichenkontext hält seine Eigenschaften fest, solange die Seite offen ist, oder bis Sie etwas unternehmen, dass sie neu setzt.
- `fillRect(`*`x`*`, `*`y`*`, `*`width`*`, `*`height`*`)` zeichnet ein Rechteck, das mit dem aktuellen Füllstil gefüllt wird.
- Die `strokeStyle`-Eigenschaft gleicht `fillStyle`, d.h., sie kann eine CSS-Farbe, ein Muster oder einen Verlauf enthalten.
- `strokeRect(`*`x`*`, `*`y`*`, `*`width`*`, `*`height`*`)` zeichnet ein Rechteck mit dem aktuellen Strichstil. `strokeRect` füllt das Rechteck nicht, sondern zeichnet nur den Rahmen.
- `clearRect(`*`x`*`, `*`y`*`, `*`width`*`, `*`height`*`)` leert die Pixel im angegebenen Rechteck.

Fragen an Professor Markup

F: Kann ich ein Canvas »zurücksetzen«?

A: Ja. Das Setzen der Breite oder Höhe eines `<canvas>`-Elements löscht seinen Inhalt und setzt alle Eigenschaften auf seinem Zeichenkontext auf ihre Standardwerte zurück. Sie müssen die Breite nicht ändern, es reicht, wenn Sie sie folgendermaßen wieder auf ihren aktuellen Wert setzen:

```
var b_canvas = document.getElementById("b");
b_canvas.width = b_canvas.width;
```

Kehren wir zum Code aus unserem vorangegangenen Beispiel zurück:

```
var b_canvas = document.getElementById("b");
var b_context = b_canvas.getContext("2d");
b_context.fillRect(50, 25, 150, 100);
```

Der Aufruf von `fillRect()` zeichnet ein Rechteck und füllt es mit dem aktuellen Füllstil, d.h. mit Schwarz, da wir den Füllstil nicht geändert haben. Das Rechteck wird von seiner oberen linken Ecke (`50, 25`), seiner Breite (`150`) und seiner Höhe (`100`) definiert. Schauen wir uns das Canvas-Koordinatensystem an, damit wir uns die Sache besser vorstellen können.

Canvas-Koordinaten

Das Canvas ist ein zweidimensionales Raster. Die Koordinaten (0, 0) bilden die obere linke Ecke des Canvas. Entlang der x-Achse erhöhen sich die Werte in Richtung des rechten Rands des Canvas, entlang der y-Achse erhöhen sich die Werte in Richtung seines unteren Rands.

Das Koordinatendiagramm in Abbildung 4-2 wurde in einem `<canvas>`-Element gezeichnet. Sie besteht aus:

- einem Satz vertikaler cremefarbener Linien,
- einem Satz horizontaler cremefarbener Linien,
- zwei horizontalen schwarzen Linien,
- zwei kleinen diagonalen schwarzen Linien, die einen Pfeil bilden,
- zwei vertikalen schwarzen Linien,
- zwei kleinen diagonalen schwarzen Linien, die einen weiteren Pfeil bilden,
- dem Buchstaben »x«,
- dem Buchstaben »y«,
- dem Text »(0, 0)« nahe der oberen linken Ecke,
- dem Text »(500, 375)« nahe der unteren linken Ecke sowie
- einem Punkt in der oberen linken Ecke und einem anderen in der unteren rechten Ecke.

In den folgenden Abschnitten werden wir uns ansehen, wie man die Effekte erzeugt, die in dieser Abbildung gezeigt werden, aber erst müssen wir das `<canvas>`-Element selbst definieren. Folgendes `<canvas>`-Element definiert mit den Eigenschaften `width` und `height` die Breite der Zeichenfläche und die `id`, damit wir es später finden können:

```
<canvas id="c" width="500" height="375"></canvas>
```

Dann benötigen wir ein Skript, um das `<canvas>`-Element im DOM zu finden und seinen Zeichenkontext abzurufen:

```
var c_canvas = document.getElementById("c");
var context = c_canvas.getContext("2d");
```

Jetzt können wir damit beginnen, Linien zu zeichnen.

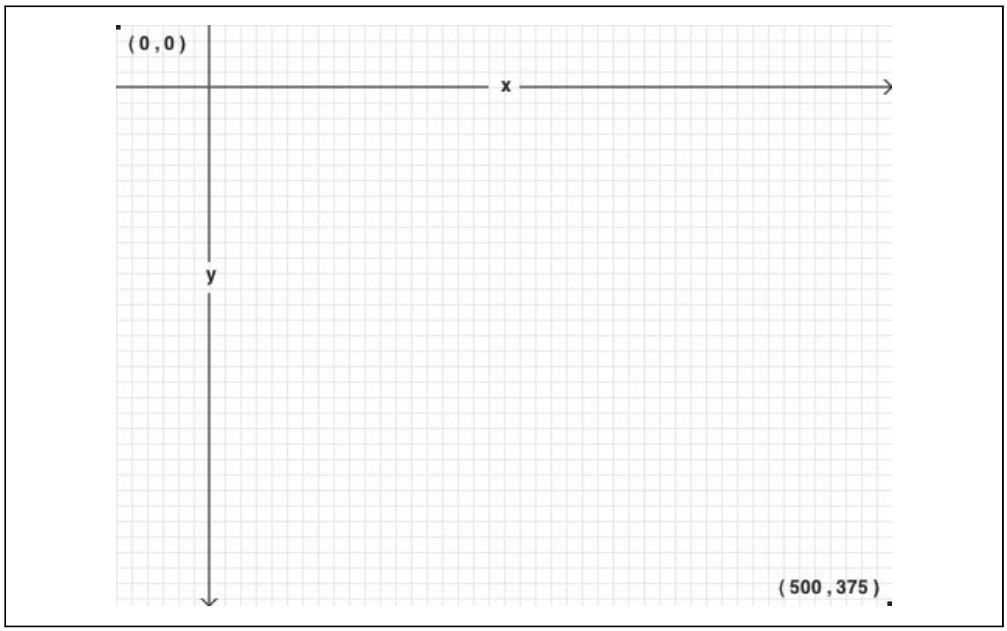

Abbildung 4-2: Ein Canvas-Koordinatendiagramm

Pfade

IE[3]	Firefox	Safari	Chrome	Opera	iPhone	Android
7.0+	3.0+	3.0+	3.0+	10.0+	1.0+	1.0+

Mal angenommen, Sie möchten eine Tintenzeichnung erstellen. Normalerweise fangen Sie nicht sofort an, mit Tinte zu zeichnen, weil Sie eventuelle Fehler dann nicht so leicht beheben können. Stattdessen werden Sie die Linien und Kurven zunächst mit einem Bleistift skizzieren und diese Skizze erst dann mit Tinte überzeichnen, wenn Sie mit Ihrem Entwurf zufrieden sind.

Jedes Canvas hat einen *Pfad*. Die Definition des Pfads ist wie das Zeichnen mit einem Bleistift. Sie können zeichnen, was Sie wollen, aber Teil des fertigen Produkts wird es erst, wenn Sie zur Feder greifen und Ihren Pfad mit Tinte überzeichnen.

Die folgenden beiden Methoden nutzen Sie, um zwei gerade Linien mit Bleistift zu zeichnen:

- `moveTo(x, y)` bewegt den Bleistift zum angegebenen Startpunkt.
- `lineTo(x, y)` zeichnet eine Linie zum angegebenen Endpunkt.

3 Der Internet Explorer benötigt die externe Bibliothek explorercanvas.

Je häufiger Sie moveTo() und lineTo()aufrufen, umso umfangreicher wird der Pfad. Diese beiden Methoden sind »Bleistiftmethoden«. Sie können sie so oft aufrufen, wie Sie wollen, aber Sie werden so lange nichts sehen, bis Sie eine der »Tintenmethoden« aufrufen.

Beginnen wir damit, das cremefarbene Raster zu zeichnen:

```
for (var x = 0.5; x < 500; x += 10) {
  context.moveTo(x, 0);
  context.lineTo(x, 375);
}

for (var y = 0.5; y < 375; y += 10) {
  context.moveTo(0, y);
  context.lineTo(500, y);
}
```

Da das alles »Bleistiftmethoden« waren, wurde noch nichts auf das Canvas gezeichnet. Wir müssen es erst mit einer »Tintenmethode« festigen:

```
context.strokeStyle = "#eee";
context.stroke();
```

stroke() ist eine der »Tintenmethoden«. Sie übernimmt den komplexen Pfad, den Sie mit diesen ganzen moveTo()- und lineTo()-Aufrufen erzeugt haben, und zeichnet ihn tatsächlich auf das Canvas. strokeStyle steuert die Farbe Ihrer Linien. Abbildung 4-3 zeigt das Ergebnis.

Abbildung 4-3: Ein auf ein Canvas gezeichnetes Raster

Fragen an Professor Markup

F: Warum haben Sie `x` und `y` bei `0.5` beginnen lassen? Warum nicht bei `0`?

A: Stellen Sie sich die einzelnen Pixel als große Quadrate vor. Die ganzzahligen Koordinaten (`0`, `1`, `2`, ...) sind die Ränder der Quadrate. Wenn Sie auf einer ganzzahligen Koordinate eine Linie zeichnen, die eine Einheit breit ist, reicht diese zur Hälfte jeweils in die Pixelquadrate auf beiden Seiten. Da der Rechner keine halben Pixel zeichnen kann, wird die Linie also mit zwei Pixeln Breite gezeichnet. Wollen Sie eine Linie zeichnen, die nur ein Pixel breit ist, müssen Sie die Koordinaten lotrecht zur Richtung der Linie um 0,5 verschieben.

Zeichnen wir jetzt den horizontalen Pfeil. Alle Linien und Kurven auf einem Pfad werden in der gleiche Farbe gezeichnet (stimmt, Verläufe, mit denen wollten wir uns auch noch befassen). Den Pfeil wollen wir in einer anderen Farbe zeichnen – Tintenschwarz statt cremefarben –, also müssen wir dazu einen neuen Pfad beginnen:

```
context.beginPath();
context.moveTo(0, 40);
context.lineTo(240, 40);
context.moveTo(260, 40);
context.lineTo(500, 40);
context.moveTo(495, 35);
context.lineTo(500, 40);
context.lineTo(495, 45);
```

Der vertikale Pfeil sieht ganz ähnlich aus. Da er die gleiche Farbe wie der horizontale Pfeil hat, müssen wir diesmal keinen neuen Pfad beginnen. Die beiden Pfeile werden Teil des gleichen Pfads sein:

```
context.moveTo(60, 0);
context.lineTo(60, 153);
context.moveTo(60, 173);
context.lineTo(60, 375);
context.moveTo(65, 370);
context.lineTo(60, 375);
context.lineTo(55, 370);
```

Ich sagte, dass diese Pfeil schwarz werden sollen, aber unser `strokeStyle` ist immer noch cremefarben. (`fillStyle` und `strokeStyle` werden nicht zurückgesetzt, wenn ein neuer Pfad begonnen wird.) Bislang war das kein Problem, da wir nur »Bleistiftmethoden« genutzt haben. Aber bevor wir den neuen Pfad tatsächlich, in »Tinte«, zeichnen lassen, müssen wir `strokeStyle` auf Schwarz setzen. Sonst werden die Pfeile cremefarben und folglich kaum erkennbar sein! Die folgenden Zeilen ändern die Farbe in Schwarz und zeichnen die Linien auf das Canvas:

```
context.strokeStyle = "#000";
context.stroke();
```

Abbildung 4-4 zeigt das Ergebnis.

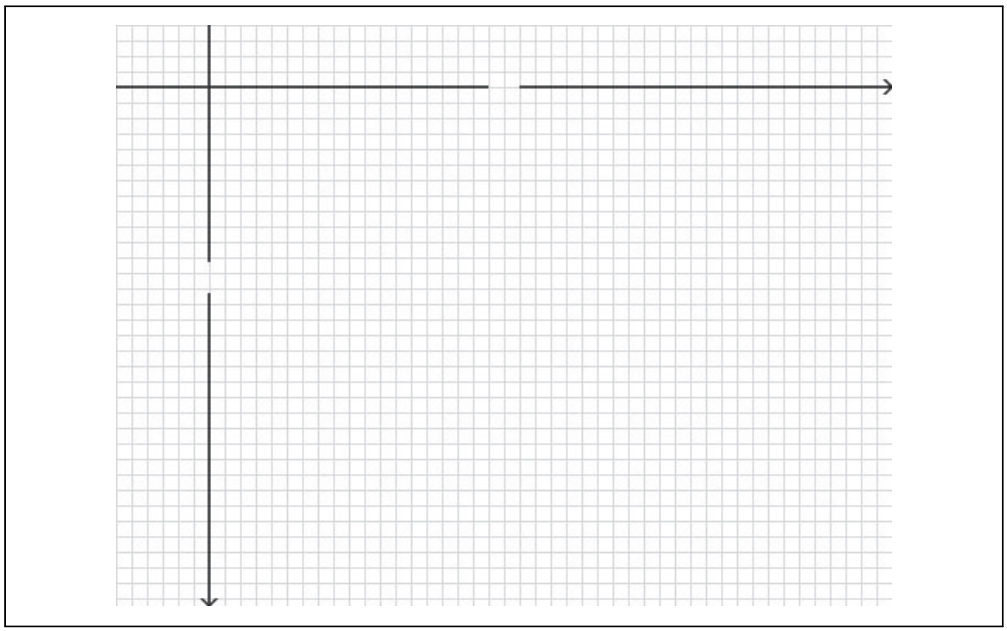

Abbildung 4-4: Unbeschriftete Achsen auf einem Canvas

Text

IE[4]	Firefox[5]	Safari	Chrome	Opera	iPhone	Android
7.0+	3.0+	3.0+	3.0+	10.0+	1.0+	1.0+

Auf ein Canvas können Sie nicht nur Linien, sondern auch Text zeichnen. Anders als beim Text in der umgebenden Webseite gibt es kein Box-Modell. Das heißt, dass keine der vertrauten CSS-Layouttechniken verfügbar sind: keine schwebenden Elemente, keine Außenabstände, keine Innenabstände, kein Zeilenumbruch. (Vielleicht sind Sie ja der Meinung, das sei gar nicht so verkehrt!) Sie können ein paar Schriftattribute setzen, und anschließend wählen Sie sich einen Punkt auf Ihrem Canvas, um dort Ihren Text zu zeichnen.

Die folgenden Schriftattribute sind auf dem Zeichenkontext verfügbar (siehe dazu den Abschnitt »Einfache Figuren« auf Seite 60):

- `font` kann ein beliebiger Wert sein, den Sie auch in einer CSS-`font`-Regel angeben. Das schließt den Schriftschnitt, die Schriftvariante, das Schriftgewicht, die Schriftgröße, die Zeilenhöhe und die Schriftfamilie ein.
- `textAlign` steuert die Textausrichtung. Es ähnelt der CSS-`text`-Regel `align` (ist mit ihr aber nicht identisch). Mögliche Werte sind `start`, `end`, `left`, `right` und `center`.

4 Der Internet Explorer benötigt die externe Bibliothek explorercanvas.
5 Mozilla Firefox 3.0 benötigt eine Kompatibilitätsschicht.

- `textBaseline` steuert, auf welcher Höhe im em-Quadrat der Text gezeichnet wird. Mögliche Werte sind `top`, `hanging`, `middle`, `alphabetic`, `ideographic` und `bottom`.

`textBaseline` ist verzwickt, weil Text verzwickt ist. (Na gut, englischer Text ist kein Problem, aber auf ein Canvas kann man beliebige Unicode-Zeichen zeichnen, und Unicode ist ein Problem.) Die HTML5-Spezifikation erläutert die verschiedenen Basislinien für Text:[6]

> Die obere Grundlinie des Gevierts entspricht ungefähr dem oberen Rand der Glyphen (d.h. der Schriftzeichen) einer Schrift. An der hängenden Grundlinie sind einige Glyphen wie आ verankert. Die Mittellinie bezeichnet die Mitte zwischen der oberen und der unteren Begrenzung des Gevierts. Die alphabetische Grundlinie gibt die Höhe an, auf der Zeichen wie Á, ÿ, f und Ω verankert sind. Auf der ideographischen Grundlinie sind Glyphen wie 私 und 達 verankert. Auf der unteren Grundlinie des Gevierts liegen Schriftzeichen mit Unterlängen wie p oder y in etwa auf. Der Abstand zwischen der unteren bzw. der oberen Begrenzung der Zeichen-Box und den Grundlinien des Gevierts kann recht groß sein, weil manche Schriftzeichen weit über das Geviert hinaus reichen (siehe Abbildung 4-5).

Abbildung 4-5: Text-Grundlinien

Bei einfachen Alphabeten wie dem englischen oder dem deutschen können Sie sich bei der Eigenschaft `textBaseline` problemlos auf die Werte `top`, `middle` oder `bottom` beschränken.

Zeichnen wir etwas Text! Text, der in ein Canvas gezeichnet wird, erbt die Schriftgröße und den Schriftschnitt des `<canvas>`-Elements. Aber Sie können diese Werte überschreiben, indem Sie die `font`-Eigenschaft des Zeichenkontexts nutzen:

```
context.font = "bold 12px sans-serif";
context.fillText("x", 248, 43);
context.fillText("y", 58, 165);
```

6 *http://bit.ly/aHCdDO*. Das Geviert (oder em-Quadrat) ist eine typografische Maßeinheit, mit der die Höhe und die Breite von Schriftzeichen festgelegt wird. Die Maßeinheit em wird verwendet, um die quadratische Fläche zu beschreiben, deren Breite gleich der Höhe des Schriftkegels ist. Das Geviert definiert also in der Höhe den Mindestzeilenabstand einer Schrift.

Die Methode `fillText()` zeichnet den eigentlichen Text:

```
context.font = "bold 12px sans-serif";
context.fillText("x", 248, 43);
context.fillText("y", 58, 165);
```

Fragen an Professor Markup

F: Kann ich relative Schriftgrößen nutzen, um Text in ein Canvas zu zeichnen?

A: Ja. Wie alle anderen HTML-Elemente in Ihrer Seite hat das `<canvas>`-Element eine auf Basis der CSS-Regeln der Seite berechnete Schriftgröße. Setzen Sie die Eigenschaft `context.font` auf eine relative Schriftgröße wie `1.5em` oder `150%`, multipliziert Ihr Browser diesen Wert mit der für das `<canvas>`-Element selbst berechneten Schriftgröße.

Nehmen wir an, wir wollen, dass der Text oben links bei `y=5` erscheint. Leider sind wir faul und haben absolut keine Lust, die Texthöhe zu messen und dann auf dieser Basis die Grundlinie zu berechnen. Stattdessen können wir `textBaseline` einfach auf `top` setzen und dann die Koordinate der Zeichen-Box übergeben:

```
context.textBaseline = "top";
context.fillText("( 0 , 0 )", 8, 5);
```

Jetzt zum Text unten rechts. Angenommen, das rechte Ende des Textes soll sich auf den Koordinaten `(492,370)` befinden – nur wenige Pixel von der unteren rechten Ecke des Canvas selbst entfernt. Wieder sind wir zu faul, die Breite oder Höhe des Texts zu messen. Stattdessen setzen wir `textAlign` auf `right` und `textBaseline` auf `bottom` und rufen mit `fillText()` den Text auf den Koordinaten an der unteren rechten Ecke der Zeichen-Box auf:

```
context.textAlign = "right";
context.textBaseline = "bottom";
context.fillText("( 500 , 375 )", 492, 370);
```

Abbildung 4-6 zeigt das Ergebnis.

Mist! Wir haben die Punkte in den Ecken vergessen. Wie man Kreise zeichnet, werden wir uns etwas später ansehen. Jetzt mogeln wir etwas und zeichnen sie einfach als Rechtecke (siehe dazu den Abschnitt »Einfache Figuren« auf Seite 60):

```
context.fillRect(0, 0, 3, 3);
context.fillRect(497, 372, 3, 3);
```

Und das war es! Abbildung 4-7 zeigt das endgültige Ergebnis.

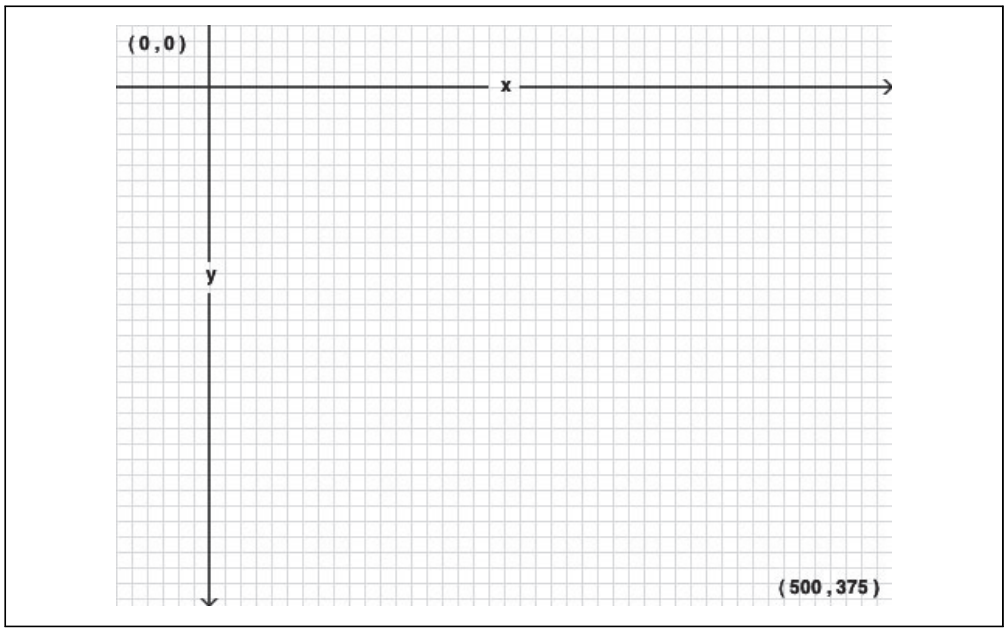

Abbildung 4-6: Beschriftete Achsen auf einem Canvas

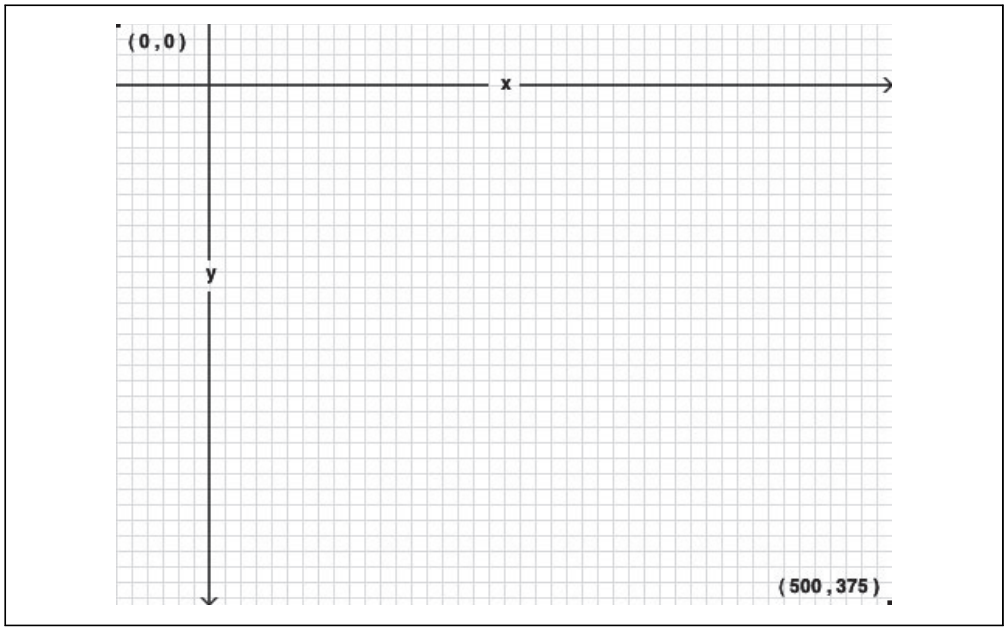

Abbildung 4-7: Ein Canvas-Koordinatensystem auf einem Canvas

Verläufe

	IE[7]	Firefox	Safari	Chrome	Opera	iPhone	Android
Lineare Verläufe	7.0+	3.0+	3.0+	3.0+	10.0+	1.0+	1.0+
Radiale Verläufe	·	3.0+	3.0+	3.0+	10.0+	1.0+	1.0+

Zu Beginn dieses Kapitels haben wir uns angesehen, wie man ein Rechteck zeichnet, das durchgängig mit einer Farbe gefüllt ist (siehe dazu den Abschnitt »Einfache Figuren« auf Seite 60), später haben wir eine ebenso gefüllte Linie gezeichnet (siehe Abschnitt »Pfade« auf Seite 63). Aber bei beidem sind wir nicht auf Einfarbigkeit beschränkt. Mit Verläufen können wir die vielfältigsten Zauberkunststücke vollführen. Abbildung 4-8 zeigt Ihnen ein Beispiel.

Das Markup sieht aus wie bei jedem anderen Canvas:

```
<canvas id="d" width="300" height="225"></canvas>
```

Erst müssen wir das <canvas>-Element und seinen Zeichenkontext abrufen:

```
var d_canvas = document.getElementById("d");
var context = d_canvas.getContext("2d");
```

Abbildung 4-8: Ein linearer Verlauf von links nach rechts

Sobald wir den Zeichenkontext haben, können wir mit der Definition des Verlaufs loslegen. Ein Verlauf ist ein sanfter Wechsel von zwei oder mehr Farben. Der Canvas-Zeichenkontext unterstützt zwei Arten von Verläufen:

- `createLinearGradient(`*x0*`,` *y0*`,` *x1*`,` *y1*`)` malt entlang einer Linie von (*x0*, *y0*) nach (*x1*, *y1*).
- `createRadialGradient(`*x0*`,` *y0*`,` *r0*`,` *x1*`,` *y1*`,` *r1*`)` malt entlang eines Kegels zwischen zwei Kreisen. Die ersten drei Parameter repräsentieren den Ausgangskreis mit dem Ursprung (*x0*, *y0*) und dem Radius *r0*. Die letzten drei Parameter repräsentieren den Zielkreis mit dem Ursprung (*x1*, *y1*) und dem Radius *r1*.

7 Der Internet Explorer benötigt die externe Bibliothek explorercanvas.

Erstellen wir einen linearen Verlauf. Verläufe können beliebige Größen haben. Diesen werden wir 300 Pixel breit machen, genauso breit wie das Canvas also:

```
var my_gradient = context.createLinearGradient(0, 0, 300, 0);
```

Da die y-Werte (der zweite und der vierte Parameter) beide `0` sind, fließt dieser Verlauf gleichförmig von links nach rechts.

Haben wir unseren Verlauf erzeugt, können wir die Farben definieren. Ein Verlauf hat zwei oder mehr Farbstopps. Diese können an beliebigen Punkten entlang des Verlaufs liegen. Wollen Sie einen Farbstopp einfügen, müssen Sie seine Position auf dem Verlauf angeben. Diese Position wird durch einen beliebigen Wert zwischen 0 und 1 bestimmt.

Definieren wir einen Verlauf von Schwarz nach Weiß:

```
my_gradient.addColorStop(0, "black");
my_gradient.addColorStop(1, "white");
```

Durch die Definition eines Verlaufs wird noch nichts auf das Canvas gezeichnet. Er ist einfach ein Objekt, das irgendwo im Speicher sitzt. Wollen Sie ihn zeichnen, setzen Sie `fillStyle` auf den Verlauf und zeichnen eine Figur wie ein Rechteck oder eine Linie:

```
context.fillStyle = my_gradient;
context.fillRect(0, 0, 300, 225);
```

Abbildung 4-9 zeigt das Ergebnis.

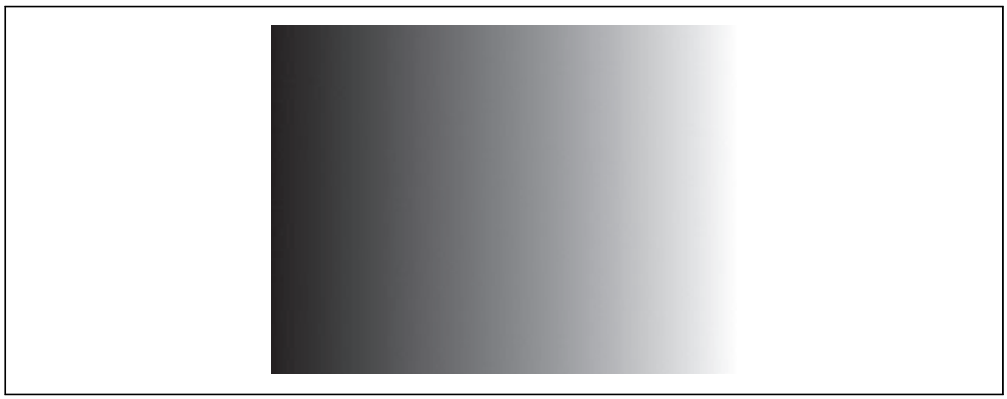

Abbildung 4-9: Ein linearer Verlauf von links nach rechts

Angenommen, Sie wollen aber einen Verlauf erzeugen, der von oben nach unten fließt. Halten Sie dazu bei der Erstellung des Verlaufs die x-Werte (den ersten und dritten Parameter) konstant und lassen Sie die y-Werte (den zweiten und vierten Parameter) von 0 bis zur Höhe des Canvas reichen:

```
var my_gradient = context.createLinearGradient(0, 0, 0, 225);
my_gradient.addColorStop(0, "black");
my_gradient.addColorStop(1, "white");
context.fillStyle = my_gradient;
context.fillRect(0, 0, 300, 225);
```

Abbildung 4-10 zeigt das Ergebnis.

Abbildung 4-10: Ein linearer Verlauf von oben nach unten

Sie können auch diagonal verlaufende Verläufe erstellen, so zum Beispiel:

```
var my_gradient = context.createLinearGradient(0, 0, 300, 225);
my_gradient.addColorStop(0, "black");
my_gradient.addColorStop(1, "white");
context.fillStyle = my_gradient;
context.fillRect(0, 0, 300, 225);
```

Abbildung 4-11 zeigt das Ergebnis.

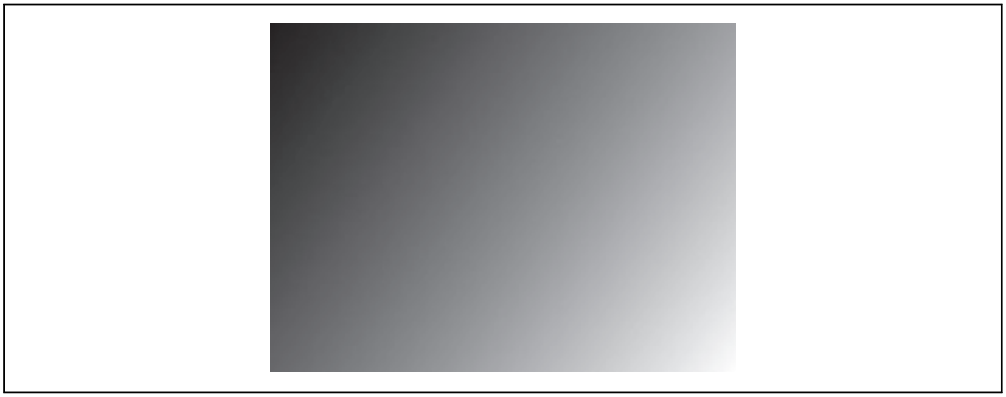

Abbildung 4-11: Ein diagonaler linearer Verlauf

Bilder

IE[8]	Firefox	Safari	Chrome	Opera	iPhone	Android
7.0+	3.0+	3.0+	3.0+	10.0+	1.0+	1.0+

Abbildung 4-12 zeigt das Bild einer Katze, das mit einem `<img>`-Element angezeigt wird.

Abbildung 4-12: Katze mit einem <img>-Element

Abbildung 4-13 zeigt die gleiche Katze, aber auf ein Canvas gezeichnet.

Abbildung 4-13: Katze mit einem <canvas>-Element

Der Canvas-Zeichenkontext definiert verschiedene Methoden, um Bilder auf ein Canvas zu zeichnen:

- `drawImage(`*`Bild, dx, dy`*`)` erwartet ein Bild und zeichnet es auf das Canvas. Die übergebenen Koordinaten (*dx, dy*) repräsentieren die obere linke Ecke des Bilds. Bei den Koordinaten (0, 0) würde das Bild in die obere linke Ecke des Canvas gezeichnet.
- `drawImage(`*`Bild, dx, dy, dw, dh`*`)` erwartet ein Bild, skaliert es auf eine Breite von *dw* sowie eine Höhe von *dh* und zeichnet es bei den Koordinaten(*dx, dy*) auf das Canvas.
- `drawImage(`*`Bild, sx, sy, sw, sh, dx, dy, dw, dh`*`)` nimmt ein Bild, schneidet es auf das Rechteck (*sx, sy, sw, sh*) zu, skaliert es auf die Größe (*dw, dh*) und zeichnet es bei den Koordinaten (*dx, dy*) auf das Canvas.

Die HTML5-Spezifikation erläutert die Parameter für `drawImage()` (*http://bit.ly/9WTZAp*) folgendermaßen:

> Das Quellrechteck ist das Rechteck (in der Bildquelle), dessen Ecken die Punkte (*sx, sy*), (*sx+sw, sy*), (*sx+sw, sy+sh*), (sx, sy+sh) sind.

8 Der Internet Explorer benötigt die externe Bibliothek explorercanvas.

Das Zielrechteck ist das Rechteck (im Canvas), dessen Ecken die vier Punkte (*dx, dy*), (*dx+dw, dy*), (*dx+dw, dy+dh*), (*dx, dy+dh*) sind.

Abbildung 4-14 zeigt eine visuelle Darstellung dieser Parameter.

Wenn Sie ein Bild auf ein Canvas zeichnen wollen, benötigen Sie zunächst einmal ein Bild. Das Bild kann ein vorhandenes `<img>`-Element oder ein mit JavaScript erstelltes `Image`-Objekt sein. In beiden Fällen müssen Sie sicherstellen, dass das Bild vollständig geladen ist, bevor Sie versuchen, es auf das Canvas zu zeichnen.

Nutzen Sie ein vorhandenes `<img>`-Element, können Sie es gefahrlos während des Events `window.onload` zeichnen:

```
<img id="cat" src="images/cat.png" alt="sleeping cat" width="177" height="113">
<canvas id="e" width="177" height="113"></canvas>
<script>
window.onload = function() {
  var canvas = document.getElementById("e");
  var context = canvas.getContext("2d");
  var cat = document.getElementById("cat");
  context.drawImage(cat, 0, 0);
};
</script>
```

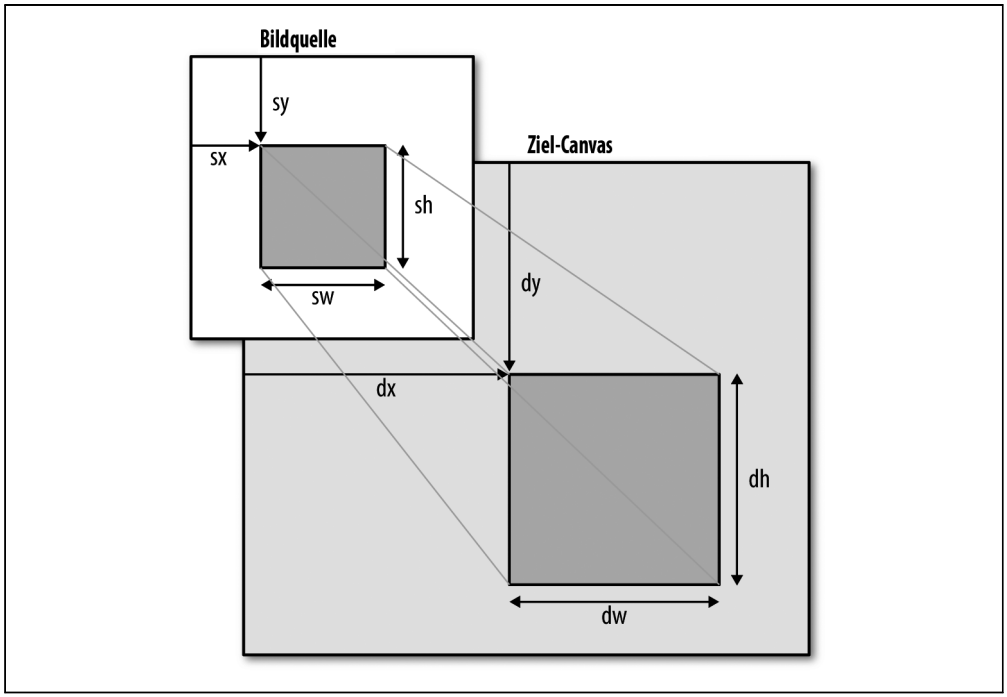

Abbildung 4-14: Wie drawImage() ein Bild in das Canvas einfügt

Möchten Sie ein Image-Objekt vollständig in JavaScript erstellen, können Sie das Bild gefahrlos während des Events `Image.onload` zeichnen:

```
<canvas id="e" width="177" height="113"></canvas>
<script>
  var canvas = document.getElementById("e");
  var context = canvas.getContext("2d");
  var cat = new Image();
  cat.src = "images/cat.png";
  cat.onload = function() {
    context.drawImage(cat, 0, 0);
  };
</script>
```

Die optionalen dritten und vierten Parameter für die Methode `drawImage()` steuern die Skalierung des Bilds. Abbildung 4-15 zeigt das gleiche Katzenbild auf die halbe Breite und Höhe skaliert und mehrfach an unterschiedlichen Koordinaten in ein einziges Canvas gezeichnet:

Hier ist das Skript, das diese »Katzenfamilie« zeichnet:

```
cat.onload = function() {
  for (var x = 0, y = 0;
       x < 500 && y < 375;
       x += 50, y += 37) {
    context.drawImage(cat, x, y, 88, 56);
  }
};
```

Die ganzen Mühen werfen natürlich eine legitime Frage auf: Warum sollte man überhaupt ein Bild in ein Canvas zeichnen wollen? Warum soll man die zusätzliche Komplexität im Vergleich zu einem gewöhnlichen `<img>`-Element und ein paar CSS-Regeln auf sich nehmen? Selbst die »Katzenfamilie« könnte man leicht mit zehn überlappenden `<img>`-Elementen nachbauen.

Die einfache Antwort ist, dass man das wohl aus demselben Grund macht, aus dem man auch Text in ein Canvas schreibt (siehe dazu den Abschnitt »Text« auf Seite 66). Unser Canvas-Koordinatendiagramm (siehe dazu den Abschnitt »Canvas-Koordinaten« auf Seite 62) schloss Text, Linien und Figuren ein. Der Text-auf-Canvas-Aspekt war nur ein Baustein in einem größeren Werk. Bei einem komplexeren Diagramm könnte man `drawImage()` beispielsweise einsetzen, um Symbole, Logos oder andere Grafiken zu zeichnen.

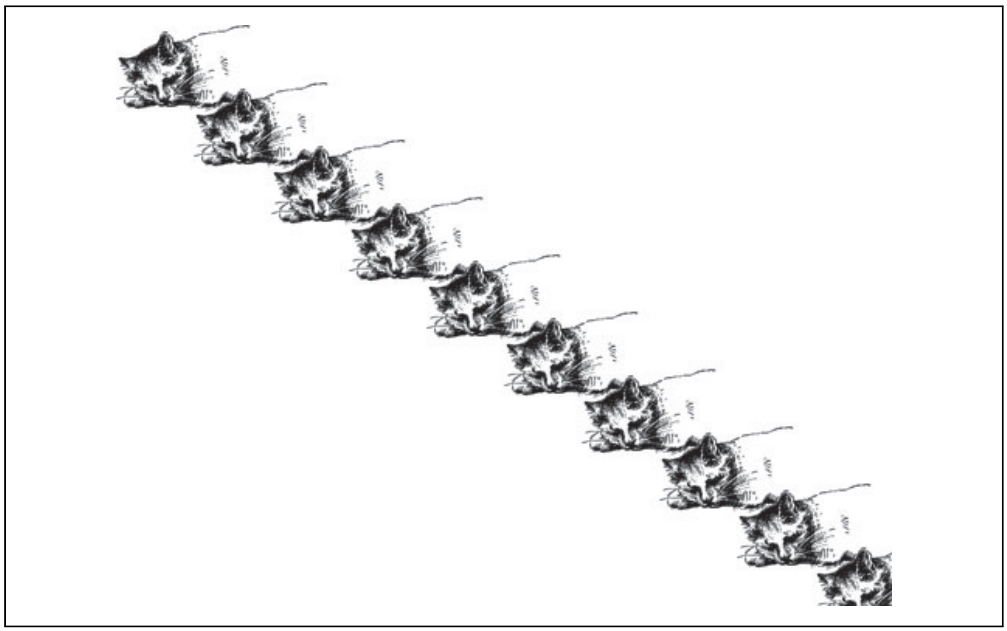

Abbildung 4-15: Eine Katzenfamilie

Was ist mit dem IE?

Microsofts Internet Explorer (bis einschließlich Version 8, der die aktuelle Version war, als dies geschrieben wurde) bietet keine Unterstützung der Canvas-API. Er unterstützt allerdings eine proprietäre Microsoft-Technologie namens VML, die ähnlich Dinge tun kann wie das `<canvas>`-Element. Und so wurde *excanvas.js* geboren.

ExplorerCanvas (*http://code.google.com/p/explorercanvas/*) – *excanvas.js* – ist eine Open Source-JavaScript-Bibliothek unter der Apache-Lizenz, die die Canvas-API im Internet Explorer nachbildet. Wenn Sie sie verwenden wollen, müssen Sie am Anfang Ihrer Seite folgendes `<script>`-Element einfügen:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Dive Into HTML5</title>
  <!--[if IE]>
    <script src="excanvas.js"></script>
  <![endif]-->
</head>
<body>
  ...
</body>
</html>
```

<!--[if IE]> und <![endif]--> sind bedingte Kommentare. Der Internet Explorer interpretiert sie als if-Anweisung: »Wenn der aktuelle Browser eine Version des Internet Explorer ist, führe diesen Block aus.« Andere Browser behandeln den gesamten Block als HTML-Kommentar. Die Folge ist, dass der Internet Explorer das Skript *excanvas.js* herunterlädt und ausführt, andere Browser es aber vollständig ignorieren (nicht herunterladen, nicht ausführen, nichts). Das bewirkt, dass Ihre Seite in Browsern, die die Canvas-API nativ unterstützen, schneller geladen wird.

Haben Sie das Skript *excanvas.js* in das <head>-Element Ihrer Seite aufgenommen, müssen Sie nichts mehr tun, um auch den Belangen des IE zu genügen. Sie können einfach <canvas>-Elemente in Ihr Markup aufnehmen oder dynamisch mit JavaScript erstellen. Folgen Sie lediglich den Anweisungen in diesem Kapitel, um den Zeichenkontext eines <canvas>-Elements abzurufen, und schon können Sie loslegen und Figuren, Text oder Muster zeichnen.

Schön wäre es. Aber leider stimmt das nicht ganz. Folgende Einschränkungen müssen Sie beachten:

- Verläufe (siehe dazu den Abschnitt »Verläufe« auf Seite 70) können nur linear sein. Radiale Verläufe werden nicht unterstützt.
- Muster müssen in beide Richtungen wiederholt werden.
- Beschneidungskanten werden nicht unterstützt.
- Nicht proportionale Skalierung skaliert Striche nicht korrekt.
- Es läuft langsam. Das sollte eigentlich niemanden sonderlich überraschen, da der JavaScript-Parser des Internet Explorer bekanntermaßen langsamer als der jedes anderen Browsers ist. Und wenn Sie dann auch noch beginnen, komplexe Figuren über eine JavaScript-Bibliothek zu zeichnen, die die Befehle für eine ganz andere Technologie übersetzt, wird das alles naturgemäß ausgebremst. Bei einfachen Beispielen, dem Zeichnen einiger Linien oder der Transformation eines Bilds, wird Ihnen der Leistungsabfall kaum auffallen. Aber er wird Ihnen sofort in die Augen springen, wenn Sie sich an Canvas-basierte Animationen und andere ausgefallene Dinge wagen.

Der Einsatz von *excanvas.js* hat noch einen weiteren Haken. Das ist ein Problem, auf das ich gestoßen bin, als ich die Beispiele für dieses Kapitel erstellte. ExplorerCanvas initialisiert seine eigene Pseudo-Canvas-Schnittstelle automatisch jedes Mal, wenn Sie das Skript *excanvas.js* in Ihre HTML-Seite einschließen. Das aber bedeutet nicht, dass der Internet Explorer sie dann auch unmittelbar nutzen kann. In bestimmten Situationen kann eine Race-Condition auftreten, bei der die Pseudo-Canvas-Schnittstelle fast, aber eben nur fast, einsatzbereit ist. Das wichtigste Symptom für diesen Zustand ist, dass sich der Internet Explorer darüber beschwert, dass das Objekt eine Eigenschaft oder Methode nicht unterstützt, wenn Sie versuchen, etwas mit einem <canvas>-Element zu tun, beispielsweise den Zeichenkontext abzurufen.

Die einfachste Lösung ist, alle Canvas-bezogenen Aktionen aufzuschieben, bis das onload-Event ausgelöst wurde. Das kann eine Weile dauern. Wenn Ihre Seite viele Bilder oder

Videos enthält, verzögern diese das `onload`-Event. Aber es gibt ExplorerCanvas Zeit, seine magischen Transformation durchzuführen.

Ein vollständiges Beispiel

Halma ist ein Jahrhunderte altes Spiel, das es in vielen Varianten gibt. Für dieses Beispiel habe ich eine Solitär-Version von Halma mit neun Spielsteinen auf einem 9 × 9 Felder großen Spielfeld erstellt. Zu Anfang des Spiels bilden die Spielsteine ein 3 × 3 Felder großes Rechteck in der unteren linken Ecke des Spielfelds. Ziel des Spiels ist es, alle Spielsteine in so wenig Zügen wie möglich so zu verschieben, dass sie ein 3 × 3 Felder großes Quadrat in der oberen rechten Ecke des Spielfelds belegen.

In Halma gibt es zwei Arten zulässiger Spielzüge:

- Einen Stein auf ein angrenzendes leeres Feld bewegen. Ein »leeres« Feld ist ein Feld, auf dem aktuell kein Stein steht. Ein angrenzendes Feld ist ein Feld, das nördlich, südlich, östlich, westlich, nordwestlich, nordöstlich, südwestlich oder südöstlich an das aktuelle Feld anschließt. (Das Spiel springt nicht von der einen Seite zur anderen über. Befindet sich ein Stein in der äußersten linken Spalte, kann er sich nicht nach Westen, Nordwesten oder Nordosten bewegen. Ist ein Stein in der untersten Reihe, kann er nicht nach Süden, Südosten oder Südwesten gehen.)
- Mit einem Stein über einen anderen Stein auf einem angrenzenden Feld springen und das so oft wiederholen, wie es möglich ist. Das bedeutet, dass es als ein Zug zählt, wenn Sie über einen Stein auf einem angrenzenden Feld springen und dann wieder über einen weiteren Stein, der sich auf einem an Ihre neue Position angrenzenden Feld befindet. Genau gesagt, eine beliebige Anzahl von Sprungen zählt als ein Zug. (Da es Ziel des Spiels ist, die Anzahl an Zügen so gering wie möglich zu halten, geht es bei Halma also darum, lange Ketten verteilter Spielsteine aufzubauen und diese dann so zu nutzen, dass andere Steine über sie hinweg mit Sprüngen eine möglichst große Strecke zurücklegen können.)

Abbildung 4-16 ist ein Screenshot des Spiels selbst. Sie können es auch online spielen (*http://diveintohtml5.org/examples/canvas-halma.html*), wenn Sie mit den Entwicklerwerkzeugen Ihres Browsers einen Blick darauf werfen wollen.

Wie das alles funktioniert? Ich kann Ihnen gar nicht sagen, wie ich mich über diese Frage freue. Ich werde Ihnen hier nicht den gesamten Code vorführen (Sie können ihn sich unter *http://diveintohtml5.org/examples/halma.js* ansehen). Den größten Teil des Codes für den Spielablauf werde ich überspringen. Aber ich möchte einige Teile daraus hervorheben, die sich um das Zeichnen auf das Canvas kümmern und auf die Mausklicks auf das `<canvas>`-Element reagieren.

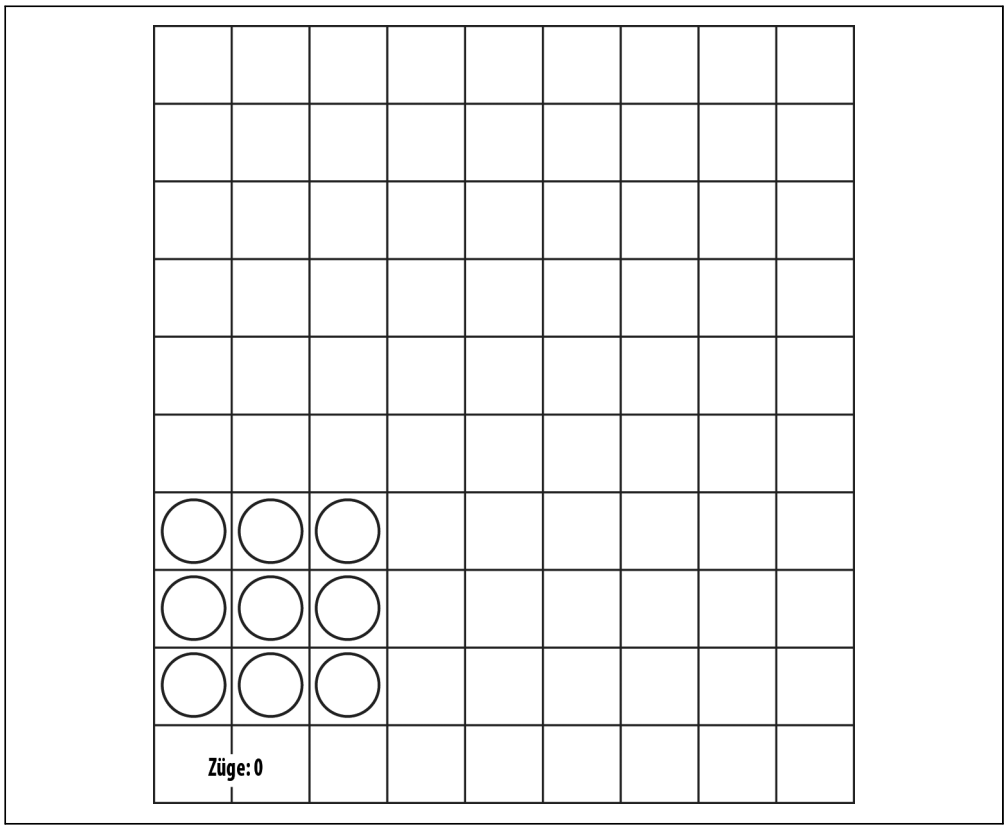

Abbildung 4-16: Startposition eines Halma-Spiels

Während die Seite geladen wird, initialisieren wir das Spiel, indem wir die Dimensionen des <canvas> selbst festlegen und eine Referenz auf seinen Zeichenkontext speichern:

```
gCanvasElement.width = kPixelWidth;
gCanvasElement.height = kPixelHeight;
gDrawingContext = gCanvasElement.getContext("2d");
```

Dann machen wir etwas, das Sie bislang noch nicht gesehen haben – wir fügen dem <canvas>-Element einen Event-Listener hinzu, der Click-Events überwacht:

```
gCanvasElement.addEventListener("click", halmaOnClick, false);
```

Die Funktion `halmaOnClick()` wird aufgerufen, wenn der Benutzer irgendwo in das Canvas klickt. Ihr Argument ist ein `MouseEvent`-Objekt, das Informationen dazu enthält, wohin der Benutzer geklickt hat:

```
function halmaOnClick(e) {
    var cell = getCursorPosition(e);

    // Was hier folgt, dient nur noch der Steuerung des Spielablaufs.
    for (var i = 0; i < gNumPieces; i++) {
```

```
            if ((gPieces[i].row == cell.row) &&
                (gPieces[i].column == cell.column)) {
                clickOnPiece(i);
                return;
            }
        }
        clickOnEmptyCell(cell);
    }
```

Der nächste Schritt ist, anhand des `MouseEvent`-Objekts zu berechnen, auf welches Feld des Halma-Bretts gerade geklickt wurde. Das Halma-Spiel nimmt das gesamte Canvas ein. Jeder Klick ist also ein Klick irgendwo auf das Spielbrett. Wir müssen herausfinden, wo. Das ist nicht ganz einfach, weil Maus-Events in beinahe allen Browsern unterschiedlich implementiert werden:

```
function getCursorPosition(e) {
    var x;
    var y;
    if (e.pageX || e.pageY) {
      x = e.pageX;
      y = e.pageY;
    }
    else {
      x = e.clientX + document.body.scrollLeft +
           document.documentElement.scrollLeft;
      y = e.clientY + document.body.scrollTop +
           document.documentElement.scrollTop;
    }
```

Die `x`- und `y`-Koordinaten, die wir jetzt haben, sind relativ zum Dokument (d.h. zur ganzen HTML-Seite). Das, was wir brauchen, sind aber Koordinaten in Bezug auf das Canvas. Diese erhalten wir so:

```
x -= gCanvasElement.offsetLeft;
y -= gCanvasElement.offsetTop;
```

Jetzt haben wir `x`- und `y`-Koordinaten in Bezug auf das Canvas (siehe dazu den Abschnitt »Canvas-Koordinaten« auf Seite 62). Das heißt, wenn `x` gleich `0` ist und `y` gleich `0` ist, wissen wir, dass der Benutzer gerade auf das Pixel oben links im Canvas geklickt hat.

Mit diesen Informationen können wir das Feld auf dem Spielbrett berechnen, auf das geklickt wurde, und dann entsprechend reagieren:

```
    var cell = new Cell(Math.floor(y/kPieceWidth),
                        Math.floor(x/kPieceHeight));
    return cell;
}
```

Wow! Maus-Events sind eine harte Nuss. Aber Sie können die gleiche Logik (eigentlich genau diesen Code) in allen Canvas-basierten Anwendungen nutzen. Denken Sie daran: Mausklick → dokumentbezogene Koordinaten → Canvas-bezogene Koordinaten → anwendungsspezifischer Code.

Okay, schauen wir uns die eigentliche Zeichenroutine an. Weil die Grafiken so einfach sind, habe ich beschlossen, dass Spielbrett vollständig zu löschen und neu zu zeichnen, wenn sich irgendetwas im Spiel ändert. Das ist nicht unbedingt erforderlich. Der Zeichenkontext hält alles fest, was zuvor auf ihn gezeichnet wurde, selbst wenn der Benutzer es außer Sicht scrollt oder den Tab wechselt und dann später zurückkehrt. Wenn Sie Canvas-basierte Anwendungen mit komplexeren Grafiken (beispielsweise ein Arkade-Spiel) erstellen, können Sie die Leistung verbessern, indem Sie festhalten, welche Bereiche des Canvas betroffen sind, und nur diese betroffenen Bereiche neu zeichnen. Aber das geht über den Horizont dieses Buchs hinaus. Hier ist der Code, der das Brett löscht:

```
gDrawingContext.clearRect(0, 0, kPixelWidth, kPixelHeight);
```

Die Routine zum Zeichnen des Spielbretts sollte vertraut aussehen. Sie hat große Ähnlichkeiten damit, wie wir unser Canvas-Koordinatendiagramm gezeichnet haben (siehe dazu den Abschnitt »Canvas-Koordinaten« auf Seite 62):

```
gDrawingContext.beginPath();

/* Vertikale Linien */
for (var x = 0; x <= kPixelWidth; x += kPieceWidth) {
    gDrawingContext.moveTo(0.5 + x, 0);
    gDrawingContext.lineTo(0.5 + x, kPixelHeight);
}

/* Horizontale Linien */
for (var y = 0; y <= kPixelHeight; y += kPieceHeight) {
    gDrawingContext.moveTo(0, 0.5 + y);
    gDrawingContext.lineTo(kPixelWidth, 0.5 +  y);
}

/* Zeichnen! */
gDrawingContext.strokeStyle = "#ccc";
gDrawingContext.stroke();
```

Richtig interessant wird es, wenn wir die einzelnen Spielsteine zeichnen müssen. Ein Spielstein ist ein Kreis, etwas also, das wir zuvor noch nicht gezeichnet haben. Und wenn der Benutzer in Vorbereitung des Zugs auf einen Spielstein klickt, soll dieser Kreis außerdem noch gefüllt werden. Hier folgt der Code, das Argument `p` repräsentiert einen Spielstein, der die Eigenschaften `row` und `column` hat, um anzuzeigen, wo er sich aktuell auf dem Spielfeld befindet. Wir nutzen einige spielspezifische Konstanten, um (`Spalte`, `Zeile`) in Canvas-bezogene (`x`, `y`)-Koordinaten zu übersetzen, und zeichnen dann einen Kreis, den wir (wenn der Spielstein ausgewählt ist) einfarbig füllen:

```
function drawPiece(p, selected) {
    var column = p.column;
    var row = p.row;
    var x = (column * kPieceWidth) + (kPieceWidth/2);
    var y = (row * kPieceHeight) + (kPieceHeight/2);
    var radius = (kPieceWidth/2) - (kPieceWidth/10);
```

Das ist das Ende des spielspezifischen Logik. Jetzt haben wir (`x`, `y`)-Koordinaten in Bezug auf das Canvas für den Mittelpunkt des zu zeichnenden Kreises. Die Canvas-API kennt

keine `circle()`-Methode. Aber es gibt eine `arc()`-Methode. Und was ist ein Kreis anderes als ein geschlossener Kreisbogen? Erinnern Sie sich noch an Ihre Geometriestunden? Die `arc()`-Methode erwartet einen Mittelpunkt (`x, y`), einen Radius, einen Start- und einen Endwinkel (beides in Radiant) sowie einen Schalter, der die Richtung anzeigt (`false` für »im Uhrzeigersinn«, `true` für »entgegen dem Uhrzeigersinn«). Wir können JavaScripts eingebautes `Math`-Modul nutzen, um die Winkel in Radiant zu berechnen:

```
gDrawingContext.beginPath();
gDrawingContext.arc(x, y, radius, 0, Math.PI * 2, false);
gDrawingContext.closePath();
```

Aber Moment! Es wurde noch nichts gezeichnet. Wie `moveTo()` und `lineTo()` ist `arc()` eine »Bleistiftmethode« (siehe dazu den Abschnitt »Pfade« auf Seite 63). Damit der Kreis tatsächlich gezeichnet und in »Tinte« verewigt wird, müssen wir `strokeStyle` setzen und `stroke()` aufrufen

```
gDrawingContext.strokeStyle = "#000";
gDrawingContext.stroke();
```

Was wir machen, wenn der Stein ausgewählt ist? Wir können den Pfad wiederverwenden, den wir für den Umriss des Steins erstellt haben, und den Kreis einfarbig füllen:

```
if (selected) {
    gDrawingContext.fillStyle = "#000";
    gDrawingContext.fill();
}
```

Und das war es so ziemlich. Der Rest des Programms ist spielspezifische Logik – gültige und ungültige Züge trennen, die Anzahl von Zügen festhalten, prüfen, ob das Spiel zu Ende ist ... Mit neuen Kreisen, ein paar geraden Linien und einem `onclick`-Handler haben wir ein vollständige Spiel in `<canvas>` erzeugt. Hurra!

Weitere Lektüre

- Canvas-Tutorial (*https://developer.mozilla.org/en/Canvas_tutorial*) im Mozilla Developer Center
- »HTML5 `canvas`–the basics« (*http://dev.opera.com/articles/view/html-5-canvas-the-basics/*) von Mihai Sucan
- Canvas Demos (*http://www.canvasdemos.com*): Demos, Tools und Einführungen zum HTML-`<canvas>`-Element
- »The `canvas` element« (*http://bit.ly/9JHzOf*) im Entwurf für den HTML5-Standard