HTML5-Tutorium: Canvas: MiniPong 02

Wechseln zu:Navigation, Suche

Dieser Artikel erfüllt die GlossarWiki-Qualitätsanforderungen **nur teilweise**:

Korrektheit: 3 (zu größeren Teilen überprüft)	**Umfang**: 3 (einige wichtige Fakten fehlen)	**Quellenangaben**: 5 (vollständig vorhanden)	**Quellenarten**: 5 (ausgezeichnet)	**Konformität**: 5 (ausgezeichnet)

HTML-Tutorium: MiniPong

MiniPong: | Teil 1 | Teil 2 | Teil 3 | Teil 4 | Teil 5

Musterlösung: `index1.html`, `index2.html`, `index3.html` (WK_MiniPong02 (SVN))

Ergänzungsaufgabe: `index1a.html`, `index2a.html`, `index2a_many.html`, `index3a.html`, `index3a_many.html` `index4a.html`, `index4a_many.html` (WK_MiniPong02a (SVN))

Inhaltsverzeichnis

1 Ziel: Animation des Balls

Die Ball-Animation des ersten Teils des Tutoriums zeichnet sich noch durch ein prinzipielles Problem aus: Bei der Darstellung der Animation wird die Leistungsfähigkeit des Rechners des Benutzers der Anwendung nicht berücksichtigt. Das ist bei einer einfachen Ball-Animation sicher auch nicht notwendig. Bei komplexeren Anwendungen kann dies allerdings schnell zu großen Performanz-Problemen führen.

Mit Hilfe zweier Optimierungstechniken kann man diese Probleme deutlich abmildern. (Das heißt allerdings nicht, dass jede Animation auf jedem Endgerät läuft.)

2 Neues Projekt anlegen

Legen Sie ein neues Projekt mit dem Namen „`MiniPong02`“ an.

Kopieren Sie alle Dateien des Projektes MiniPong01 in das neue Projekt.

Nehmen Sie folgende Anpassungen vor:

Umbenennung des Ordners „`app`“ in „`app1`“.
Umbenennung der Datei „`main.js`“ in „`main1.js`“.
Umbenennung der Datei „`index.html`“ in „`index1.html`“.
In der Datei „`main1.js`“:
- Ersetzen von „`app: 'app'`“ durch „`app: 'app1'`“.

In der Datei „`index1.html`“:
- Ändern des Titels in „`MiniPong02 (App1)`“:
- Ersetzen von „`js/main'`“ durch „`js/main1`“ im Script-Befehl.

Nun sollte die Anwendung wieder laufen.

3 Trennung von Physik und Grafik

Prinzipiell müssen in jedem Animationsschritt der Anwendung „MiniPong“ aus zwei wesentliche Aktionen durchgeführt werden:

Aktualisierung der Simulation eine physikalischen Welt.
Aktualisierung der Darstellung der Objekte, deren Physik simuliert wird, auf der Bühne.

Der erste Schritt ist meist nicht sonderlich rechenintensiv: Es müssen einige mathematische Operationen durchgeführt werden, um die Position, die Masse, den Zustand und evtl. andere Eigenschaften der simulierten Objekte zu ermitteln. Allerdings ist es notwendig, diese Berechnungen möglichst häufig pro Sekunde durchzuführen. In der realen Welt erfolgen die meisten Änderungen physikalischer Größen kontinuierlich. Beispielsweise ändern sich die Position, die Geschwindigkeit und die Beschleunigung eines Balles stetig. Die Simulation derartiger Größenänderungen erfolgt an Rechner dagegen fast immer in diskreten Schritten (Ausnahme: Analogrechner): Mehrmals pro Sekunde werden die Größen neu berechnet. Durch die diskrete Berechnung kontinuierlicher Änderungen kommt es unweigerlich zu Rechenfehlern.

Aus mathematischer Sicht wird bei der Simulation einer physikalischen Welt die Integration durch die Bildung diskreter Summen ersetzt. Bekanntlich kann ein Integral mit Hilfe von Grenzwerten von Flächensummen definiert werden. Die Grenzwertbildung bedeutet, dass die Flächen von immer mehr, immer kleineren Bereichen aufsummiert werden. Im Grenzfall sind die betrachteten Flächen unendlich klein und die Anzahl der Flächen ist unendlich groß. Diese Grenzwertbildung können wir am Rechner natürlich nicht nachbilden. Wir können allerdings versuchen, diskrete Summen von möglichst vielen, möglichst kleinen Elementen zu bilden. Das erreicht man, indem die Frequenz der physikalischen Berechnungen möglichst groß gewählt wird. 500 Neuberechnungen pro Sekunde sind deutlich besser als 20 Neuberechnungen. Und diese sollten möglichst gleichmäßig erfolgen, ohne große Lücken zwischen den einzelnen Schritten, da der zeitliche Abstand zwischen zwei Berechnungsschritten bei der Simulation berücksichtigt werden muss: Ein Ball bewegt sich in 20 ms weiter als in 10. Es ist zwar auch möglich bei jeder Neuberechnung die aktuelle Berechnungsfrequenz zu berücksichtigen – indem man für jeden Berechnungsschritt einen Zeitstempel vergibt, mit dem man ermitteln kann, wann die letzte Berechnung erfolgt ist –, besser ist es jedoch, wenn zwischen zwei Schritten stets gleich viel Zeit vergeht.

Ganz anders stellt sich die Situation im Falle der Grafikausgabe dar. Während für die physikalische Simulation ein paar tausend oder zehntausend Rechenschritte notwendig sind, benötigt man für die Darstellung der Objekte auf dem Bildschirm meist mehrere Millionen Operationen. Beispielsweise besteht ein Full-HD-Bild aus $1920 \cdot 1080 = 2.073.600$ Pixeln. Heute werden i. Allg. Hochleistungsgrafikarten eingesetzt, um mindestens 60 Bilder pro Sekunde zeichnen zu können. Anderenfalls würde das Bild zu sehr flackern. Eine Framerate von 500 Bildern pro Sekunde ist meist vollkommen undenkbar. Das ist aber auch gar nicht notwendig. Es reicht, wenn das Bild auf dem Minitor so oft wie möglich aktualisiert wird. Es müssen nur jeweils alle Objekte, an ihren aktuellen Positionen gezeichnet werden. Und ob zwischen zwei Frames mal ein paar Millisekunden mehr oder oder weniger vergehen, ist auch nicht wesentlich. Wichtiger ist, dass der Rechner für die Verarbeitung des nächsten Bildes bereit ist. Wenn er das letzte Bild noch nicht gezeichnet hat, sollte er nicht mit dem nächsten anfangen. Sonst kann es passieren, das viele aufeinander folgende Bilder nur jeweils unvollständig gezeichnet werden oder dass das System zu ruckeln anfängt, weil die Zeichnen eines jeden Bildes zu lange dauert.

Um diese Probleme zu umgehen, gibt es in JavaScript den Befehl „`window.requestAnimationFrame`“. Diese Methode erwartet als Input eine Callback-Funktion, die aufgerufen wird, sobald der Bildschirm wieder für eine Grafikausgabe bereitsteht. Üblicherweise aktualisiert diese Callback-Funktion zunächst die Bildschirmdarstellung (Neuzeichnen eins Canvas-Elements, Ändern von SVG-Elementen im DOM-Baum etc.) und ruft dann `requestAnimationFrame` rekursiv mit derselben Call-Backfunktion auf:

```
function f_update_view()
{
   // update view

   window.requestAnimationFrame( f_update_view);
 }
```

Der Callback-Funktion wird bei jedem Aufruf ein Zeitstempel mitgegeben. Damit kann beispielsweise die aktuelle Framerate ermittelt werden:

```
var l_last_step = Date.now();
function  f_update_view(p_timestamp)
{
  // update view

  console.log('FPS: ' + (Math.round(1000/(p_timestamp - l_last_step))));
  l_last_step = p_timestamp:
  window.requestAnimationFrame( f_update_view);
}
```

Jetzt muss die Animation nur noch gestartet werden. Dazu reicht es, `requestAnimationFrame` einfach einmal direkt aufzurufen:

```
window.requestAnimationFrame( f_update_view);
```

Dieser Befehl liefert übrigens einen Identifikator zurück, den man in einer Variablen speichern kann:

```
var v_animation_id = window.requestAnimationFrame(f_update_view);
```

Damit ist es möglich, die Animation zu einem späteren Zeitpunkt mittels `cancelAnimationFrame` wieder zu stoppen:

```
window.cancelAnimationFrame(v_animation_id);
```

4 Model und View

Aufgrund der obigen Erkenntnisse, sollte man in jede Web-Anwendung, die eine Welt simuliert und animiert, zwei Update-Methoden zur Verfügung haben:

`f_update_model`: Aktualisiert das (physikalische) Modell.
`f_update_view`: Aktualisiert die Bildschirmausgabe.

Die erste Methode wir regelmäßig mit Hilfe eines Timers aufgerufen. Die zweite Methode wird dagegen mit Hilfe von `requestAnimationFrame` aktiviert.

Erhöhen Sie die Simulations-Frequenz in der Datei „`init.json`“ auf 120. Laut developer.mozilla.org ist der minimale Abstand zwischen zwei Timer-Events gemäß HTML5-Spezifikation 4 ms. Dieser Wert wird auch von Browsern, die nach 2010 veröffentlicht wurden erreicht. Bei älteren Browsern betrug der minimale zeitliche Abstand noch 10 ms. Das heißt, mit einem Wert von 100 (= 1000/10) wäre man immer auf der sicheren Seite, HTML5 erlaubt Werte bis 250 (=1000/4).

Überprüfen Sie, ob Ihre Anwendung noch läuft. Wenn das der Fall ist, sollten Sie in der Datei „`minipong.js`“ die Funktion „`draw`“ durch folgende Funktion ersetzen:

```
function f_update_model()
{
  l_ball.move(l_seconds);

  // a posteriori collision detection and handling
  collision(l_canvas, l_ball);
}
```

Als nächste muss noch die Timerfunktion angepasst werden. Anstelle von `draw` ruft sie jetzt periodisch `f_update_model` auf:

```
p_window.setInterval(f_update_model, l_milliseconds);
```

Damit wir jetzt zwar das physikalische Modell des Balles animiert, aber die Animation wird noch nicht angezeigt. Dazu wird jetzt wie oben beschrieben der Befehl „`requestAnimationFrame`“ verwendet. Fügen Sie zusätzlich folgenden Code **in** die Funktion „`minipong`“ ein :

```
function f_update_view()
{
  l_context.clearRect(0, 0, l_canvas.width, l_canvas.height);
  l_ball.draw(l_context);
  p_window.requestAnimationFrame(f_update_view);
}

p_window.requestAnimationFrame(f_update_view);
```

5 Trennung von Model-Klasse und View-Klasse

Klassendiagramm

Jetzt, nachdem die Berechnung der Animation erfolgreich in Model und View getrennt wurden, sollte man dasselbe auch mit der Ball-Klasse machen. Künftig gibt es zwei Ball-Klassen:

`ModelBall`: beschreibt das physikalische Ball-Objekt
`ViewBall`: beschreibt das Aussehen des Ball-Objekts (Theme, Skin)

Damit ist es möglich, künftig unterschiedliche Themes für das Ballspiel zu definieren.

Erstellen Sie zunächst folgende Kopien:

„`app1`“ → „`app2`“ (samt Inhalt)
„`main1.js`“ → „`main2.js`“
„`index1.js`“ → „`index2.js`“
In der Datei „`main2.js`“:
- Ersetzen von „`app: 'app1'`“ durch „`app: 'app2'`“.

In der Datei „`index2.html`“:
- Ändern des Titels in „`MiniPong02 (App2)`“:
- Ersetzen von „`js/main1'`“ durch „`js/main2`“ im Script-Befehl.

Überprüfen Sie, ob Ihre Anwendung noch läuft. Erzeugen Sie danach zwei neue Ordner:

```
js/app2/model
js/app2/view
```

Verschieben Sie anschließend die Datei „`ball.js`“ in den Ordner „`js/app2/model`“ und fügen Sie anschließend eine Kopie derselben Datei in den Ordner „`js/app2/view`“ ein. Sie sollten auch noch

die Datei „`collision.js`“ in den Ordner „`js/app2/model`“ verschieben, da die Kollisionsberechnung nur etwas mit dem physikalischen Verhalten des Balles zu tun hat, aber nichts mit dessen Darstellung.

Nennen Die in der Datei „`js/app2/model/ball.js`“ den Konstruktor in „`ModelBall`“ um (Refactoring; der Bezeichner „`Ball`“ kommt mehrfach in dieser Datei vor) und entfernen Sie danach alle Parameter, Befehle und Methoden mit Bezug zur View:

```
p_init_view
this.color = p_init_view.color;
this.borderWidth = p_init_view.borderWidth;
this.borderColor = p_init_view.borderColor;
Ball.prototype.draw
```

Den Parameter „`p_init_model`“ können Sie „`p_init`“ umbenennen. Unbedingt notwendig ist das nicht. Allerdings wird es künftig in so gut wie jedem Modell-, Controller-, View- und Logic-Konstruktor einen Parameter „`p_init`“ zum Initialisieren der zugehörigen Objekte geben.

In der Datei „`js/app2/view/ball.js`“ benennen Sie den Konstruktor in „`ViewBall`“ um (abermals Refactoring) und entfernen dann alle Parameter, Befehle und Methoden mit Bezug zum Modell. Das heißt, es sollten genau diejenigen Parameter, Befehle und Methoden übrigbleiben, die Sie in der anderen Datei gelöscht haben. Hier können und sollten Sie den Parameter „`p_init_view`“ in „`p_init`“ umbenennen.

Nach diesen Änderungen haben Sie ein funktionierendes Modul „`model/ball.js`“, aber ein defektes Modul „`view/ball.js`“. Der Grund für den Defekt des View-Moduls ist die Methode „`draw`“. Diese Methode greift auf die Position und den Radius des Balles zu, d. h. auf Daten, die im Model enthalten sind, um den Ball an der richtigen Position in der richtigen Größe zu zeichnen.

Grundsätzlich gilt Folgendes:

> Die View muss das Model das Objektes kennen, das sie darstellen möchte, um das Objekt korrekt darstellen zu können. Dies wird im obigen Klassendiagramm durch einen Pfeil von `ViewBall` nach `ModelBall` ausgedrückt. Die an der Pfeilspitze angebrachte Vielfachheit `1` sagt aus, dass zu jedem View-Objekt **genau ein** Model-Objekt gehört.
> Ein Model braucht dagegen nicht zu wissen, ob, wie viele oder gar welche Views für das modellierte Objekt existieren.

Es muss also dem View-Objekt des Balles das zugehörige Model-Objekt des Balles bekannt gemacht werden. Dazu wird dem Ball-Konstruktor in der Datei „`js/app2/view/ball.js`“ das zugehörige Modell in einem zweitem Parameter übergeben. Dieses Modell wird in einem Attribut „`model`“ gespeichert:

```
function ViewBall(p_model, p_init)
{
  this.model       = p_model;

  this.color       = p_init.color;
  this.borderWidth = p_init.borderWidth;
  this.borderColor = p_init.borderColor;
}
```

Nun kann die Draw-Methode so umgeschrieben werden, dass Sie auf die Position und den Radius des

Balles wieder zugreifen kann:

```
p_context.arc(this.x, this.y, this.r, 0, 2*Math.PI);
```

wird ersetzt durch

```
p_context.arc(this.model.x, this.model.y, this.model.r, 0, 2*Math.PI);
```

Zu guter Letzt muss nur noch die MiniPong-Klasse, die alle Objekte des Spiels erzeugt, angepasst werden. Anstelle eines Ball-Objektes müssen Sie nun ein Objekt der Klasse „`ModelBall`“ und ein Objekt der Klasse „`ViewBall`“ erzeugen. (Hier könnte, sofern es mehrere geeignete View-Klassen gäbe, in der JSON-Datei angegeben werden, welche dieser Klassen verwendet werden soll. Diese Wahl bezeichnet man als Skinning.)

Zunächst müssen die beiden benötigten Module geladen werden. Der RequireJS-Rahmen sieht daher folgendermaßen aus:

```
['app/model/ball', 'app/model/collision', 'app/view/ball'],
  function(ModelBall, collision, ViewBall)
...
```

Anstelle von `l_ball` müssen zwei Objekte erzeugt werden: `l_model_ball` und `l_view_ball`:

```
var l_model_ball = new ModelBall(p_init.model.ball),
    l_view_ball  = new ViewBall(l_model_ball, p_init.view.ball),
...
```

Dem View-Objekt wird dabei – zusätzlich zum JSON-Initialisierungsobjekt – das zuvor erzeugte Model-Objekt als Argument übergeben.

Jetzt müssen nur noch die beiden Animationsmethoden angepasst werden. Die Methode „`f_update_model`“ greift auf das Model-Objekt des Balls zu

```
...
l_model_ball.move(l_seconds);
collision(l_canvas, l_model_ball);
...
```

und die Methode „`f_update_view`“ auf das View-Objekt:

```
...
l_view_ball.draw(p_context);
...
```

6 Caching der Ball-Textur

Die Draw-Methode der Klasse „`ViewBall`“ wird sehr häufig durch `requestAnimationFrame` aufgerufen (optimalerweise 60 mal pro Sekunde und Ball). Die 2D-Kontext-Zeichenbefehle (`beginPath`, `arc`, `lineWidth` etc.) sind relativ „teuer“, d. h., es dauert relativ lang diese Befehle auszuführen. Bei vielen Bällen und macht sich das durchaus bemerkbar.

Viel besser ist es, die Ball-Textur einmal beim Erzeugen des View-Objektes ein eigenständiges kleines Canvas-Objekt zu erstellen und in diesem Objekt die Balltextur zu speichern. Das heißt, die zuvor genannten 2D-Kontext-Zeichenbefehle werden nur ein einziges mal beim Erstellen des Ball-View-Objektes ausgeführt. Die Draw-Methode kopiert künftig nur noch den Inhalt dieses Mini-Canvas-Objektes an die richtige Stelle der eigentlichen Spielbühne. Dies geht wesentliche schneller, als das ständige Neuzeichnen des Balles.

Der Inhalt des View-Moduls sieht nun folgendermaßen aus (der Modulrahmen bleibt natürlich erhalten):

```
function ViewBall(p_model, p_init, p_document)
{
  this.model       = p_model;

  this.color       = p_init.color;
  this.borderWidth = p_init.borderWidth;
  this.borderColor = p_init.borderWidth;

  // Define a local canvas containing the view of the ball.
  var l_center  = this.v_center = p_model.r + this.borderWidth + 1,
      l_canvas  = this.v_canvas = p_document.createElement("canvas"),
      l_context = l_canvas.getContext("2d");

  l_canvas.width  = 2 * l_center;
  l_canvas.height = 2 * l_center;

  l_context.beginPath();
  l_context.arc(l_center, l_center, p_model.r, 0, 2*Math.PI);
  l_context.lineWidth = this.borderWidth;
  l_context.fillStyle = this.color;
  if (this.borderWidth > 0)
  {
    l_context.lineWidth   = this.borderWidth;
    l_context.strokeStyle = this.borderColor;
    l_context.stroke(); // Draw the border.
  }
  l_context.fill();     // Fill the inner area of the ball with its
color.
}

ViewBall.prototype =
{
  draw:
    function(p_context)
    {
      // Instead of drawing the ball each time, just copy the
      // mini canvas containing the image of the ball to the
      // 2d context (at the right position of course).
      p_context.drawImage(this.v_canvas,
                          this.model.x - this.v_center,
                          this.model.y - this.v_center
                         );
    }
};
```

Anmerkung: Es ist jetzt gar nicht mehr notwendig, die Attribute „`color`“, „`borderWidth`“ und „`borderColor`“ im Objekt zu speichern. Die drei im Initialisierungsobjekt „`p_init`“ enthaltenen Werte werden vom Konstruktor direkt verwendet, um ein Bild des Balls auf einem Mini-Canvas zu

zeichnen. Die Draw-Methode benötigt nur noch einen Zugriff auf das Model „`this.model`“, das Bild des Balls „`this.v_canvas`“ und den Bildmittelpunkt „`this.v_center`“. Letzteres wird benötigt, da der Aufhängepunkt eine Canvas-Elements dessen linke obere Ecke ist. Der Aufhängepunkt des Balls ist jedoch dessen Mittelpunkt. Dieser befindet sich in der Mitte des Bildes.

Man beachte, dass der Konstruktor jetzt noch einen dritten Parameter erwartet: `p_document`. Das Dokument-Objekt wird benötigt, um das lokale Canvas-Objekt zu erstellen (aber nicht, um dieses Canvas-Objekt in den DOM-Baum des Dokuments einzufügen!).

Dieser Parameter muss beim Aufruf des Konstruktor natürlich mit einem geeigneten Argument bedacht werden. Dafür ist folgende Änderung in der Datei „`minipong.js`“ notwendig:

```
...
l_view_ball  = new ViewBall(l_model_ball, p_init.view.ball,
p_window.document),
...
```

Schöner ist es natürlich, wenn zunächst eine lokale Variable „`l_document = p_window.document`“ definiert wird, da in der Funktion „`minipong`“ mehrfach auf das Dokumentobjekt zugegriffen wird.

Anmerkung: Die Technik, Bereiche aus einem Canvas auf die Hauptbühne zu kopieren, kann auch sehr gut für die Realisierung von Animationen verwendet werden. Man erstelle dazu ein Sprite Sheet, d. h. ein Bild, in dem die einzelnen Frames der Animation nebeneinander liegen und kopiert dieses Bild in ein lokales Canvas-Element. Die Draw-Methode kopiert dann nicht immer denselben Bereich, sondern nacheinander jeweils einen anderen Bereich des Sprite Sheets.

7 Erweiterung

Welchen Unterschied diese Optimierung der View-Klasse zur Folge hat, ermitteln Sie am besten, wenn Sie die Ergänzungsaufgabe von HTML5-Tutorium: Canvas: MiniPong 01 auch für diesen Teil des Tutoriums durchführen. Animieren Sie nicht nur einen Ball, sondern mehrere Bälle auf einmal. Erhöhen Sie die die Anzahl der Bälle solange, bis die Animation anfängt zu ruckeln. Welcher Unterschied zwischen App2 und App3 stellen Sie fest?

Erweitern Sie die Anwendung so, dass sich zwanzig bzw. zumindest im Falle der App2 und der App3 2000 Bälle gleichzeitig über die Bühne bewegen. Kollisionen zwischen den Bällen brauchen Sie nicht zu behandeln.

Musterlösung: `index1.html`, `index2.html`, `index2_many.html`, `index3.html`, `index3_many.html`, `index4.html`, `index4_many.html` (WK_MiniPong02a (SVN))

8 Quellen

Braun (2011): Herbert Braun; Webanimationen mit Canvas; in: c't Webdesign; Band: 2011; Seite(n): 44–48; Verlag: Heise Zeitschriften Verlag; Adresse: Hannover; 2011; Quellengüte: 5 (Artikel)
Kowarschick (MMProg): Wolfgang Kowarschick; Vorlesung „Multimedia-Programmierung“; Hochschule: Hochschule Augsburg; Adresse: Augsburg; Web-Link; 2018; Quellengüte: 3 (Vorlesung)
Musterlösung MiniPong 02 (SVN)
Musterlösung MiniPong 02a (SVN)

9 Fortsetzung des Tutoriums

Sie sollten nun Teil 3 des Tutoriums bearbeiten.

Kategorien:
Multimedia-Programmierung/Tutorium
HTML5-Tutorium: Canvas: MiniPong
HTML5-Beispiel
Kapitel:Multimedia-Programmierung:Beispiele