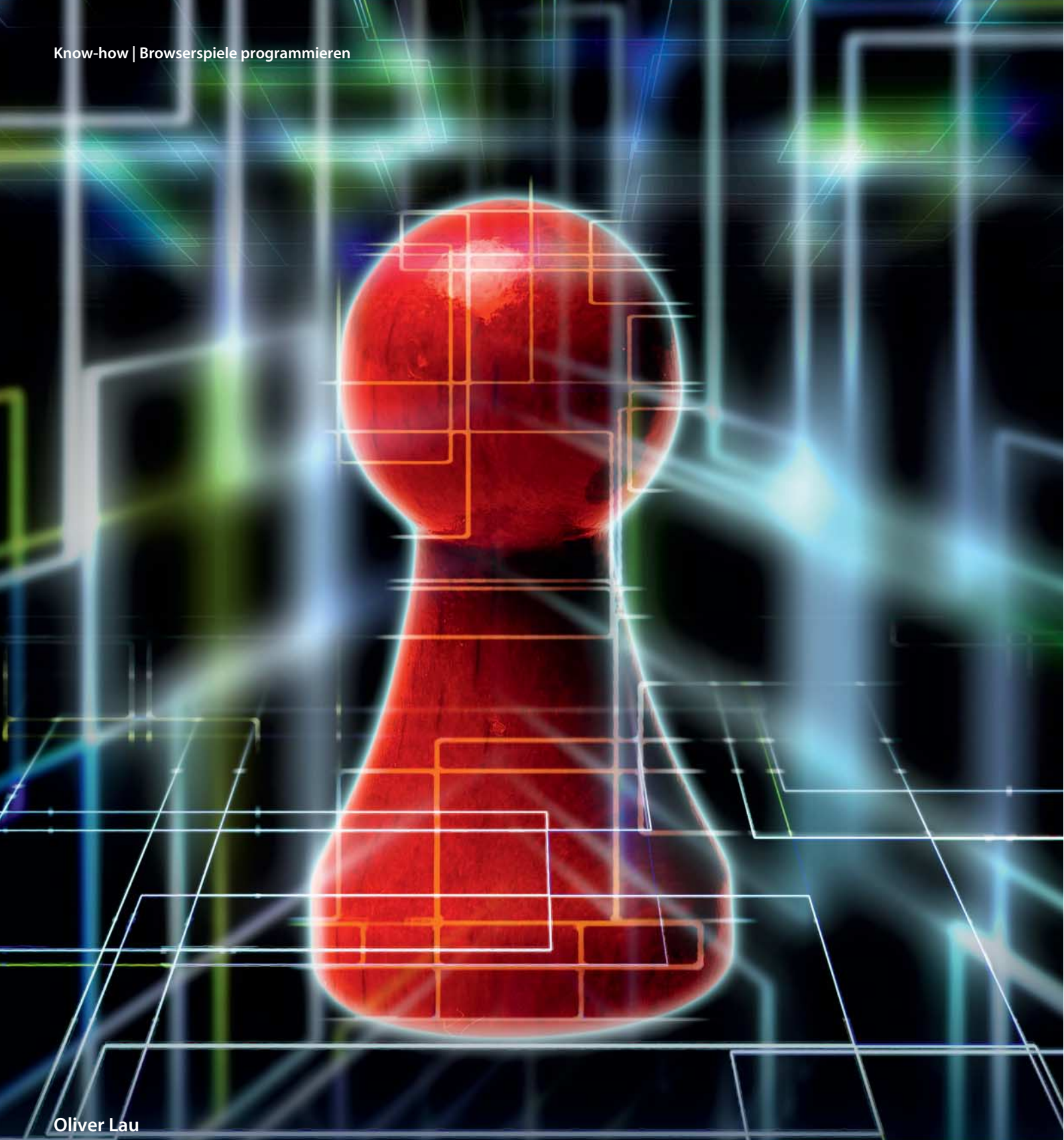


Oliver Lau

Retro aktuell

Spiele für den Browser programmieren

Ein Web-Browser ist eine erfreulich leicht zu handhabende Entwicklungs- und Spieleplattform. Am schnellsten gelingt der Einstieg in die Programmierung eines Spiels, wenn man nachbildet, was einem gefällt und was nicht übermäßig kompliziert ist, zum Beispiel ein Retro-Game wie Pac-man.

Wenn Sie in die Spieleentwicklung einsteigen möchten, müssen Sie kein Multitalent in Programmierung, Grafikgestaltung, Musikkomposition und Klangeffektkreation sein – oder reich, um all das an ein professionelles Team zu delegieren –, sondern bauen zum Anfang vielleicht ein simples, aber nichtsdestotrotz reizvolles Spiel nach. Damit müssen Sie sich fast nur noch um die Programmierung kümmern, denn Grafiken und Sounds sind en masse im Internet zu finden. Die können Sie zumindest privat verwenden, ohne sich gleich in einem Paragrafengeflecht zu verheddern.

Weil Pac-man in die Kategorien einfach und reizvoll fällt und weil die im Web spielbaren Klone entweder nicht originalgetreu sind oder ein Flash-Plug-in benötigen (oder beides), wollte ich einen eigenen Klon für den Browser entwickeln, der das besser macht. Na ja, nicht ganz: Denn meine Version des Spieleklassikers ist etwas langsamer als das Original, man könnte auch sagen: weniger hektisch. Darin liegt gerade der Vorteil, ein lieb gewonnenes Spiel neu zu interpretieren: Man kann es dem eigenen Geschmack anpassen.

Apropos Pac-man ist ein „einfaches“ Spiel: Das ist relativ, wie sich nach einiger Recherche herausstellte. Nicht nur die Spielelogik ist überraschend komplex [1], auch der Umfang der benötigten Browser-Techniken ist nicht ohne: Meine Implementierung nutzt zum Beispiel Sprites, CSS-Transitionen und -Animationen sowie Webfonts, lokalen Speicher und das Web Audio API.

Bunt und laut

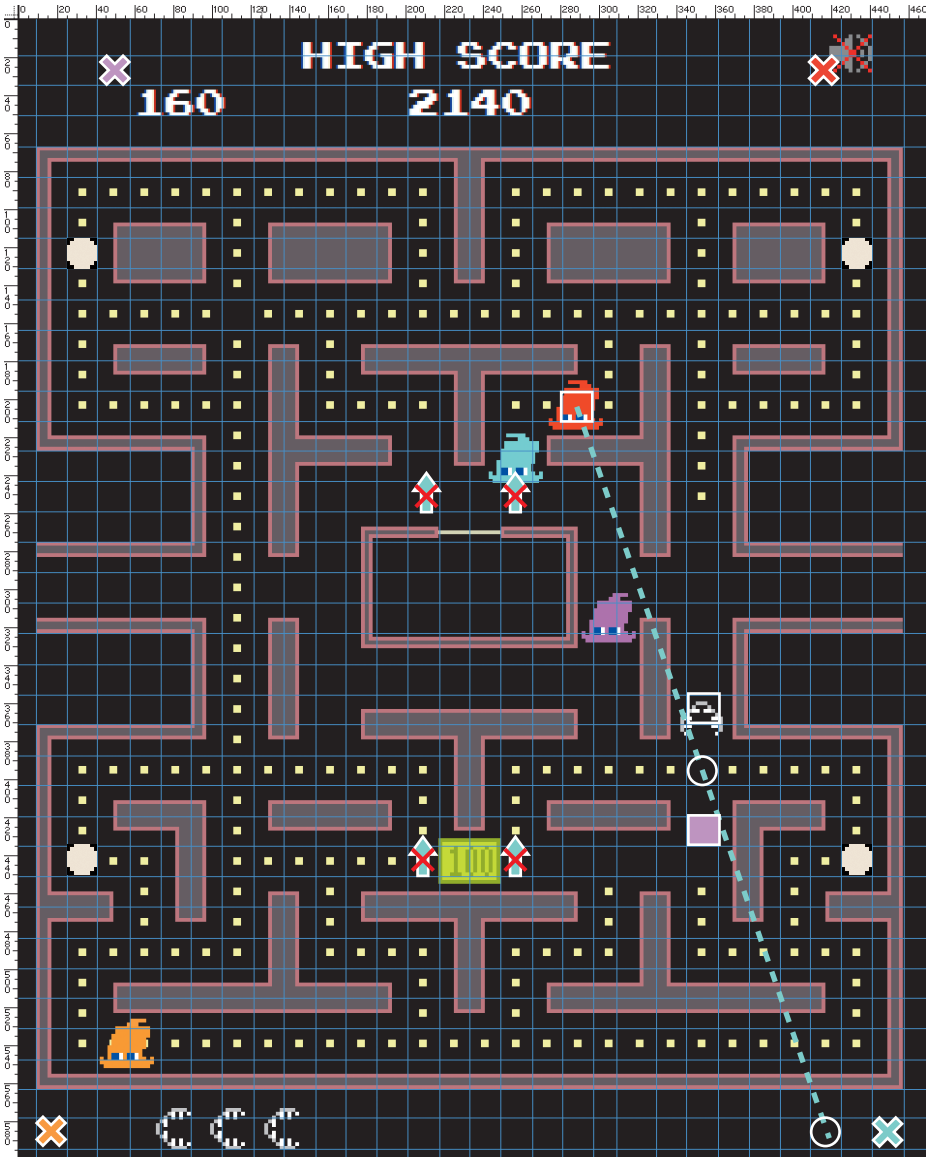
Sämtliche Grafiken (Figuren, Futter, Labyrinth) kann man zu rein privaten Zwecken nach der Vorlage von Bildern aus dem Web nachzeichnen. Aus markenschutzrechtlichen Gründen ist aus Pac-man im meiner Version ein Gebiss geworden, die Geister sind Schlapphüte und die Extras Geldscheine.

Nach passenden Sounds musste ich zum Glück nicht lange suchen: Offenbar per Mikrofon von der Original-Arcade-Konsole aufgezeichnete waren leicht zu finden. Hie und da galt es, unerwünschte Pausen rauszuschneiden, was mit der freien Soundbearbeitungssoftware Audacity schnell gelang, dann konnte ich sie für die Verwendung im Browser mit Hilfe des LAME-Plug-in als MP3 speichern (andere Formate wie OGG wären auch gegangen), natürlich ganz retro in Mono. Bei der Bitrate bin ich nicht über 128 kBit/s gegangen, weil die Sounds ohnehin in keiner besseren Qualität vorlagen.

Auch beim pixeligen Font wollte ich möglichst dicht am Original bleiben. Die Entscheidung fiel auf die Schriftart Emulogic, die zur freien Verfügung steht und sich auch für viele andere Retro-Spiele eignet.

Grundmauern

Wie jede Webseite steht auch ein Browser-Spiel auf einem Fundament aus HTML. Wie



Im „Scatter“-Modus steuert jeder Gegner ein bestimmtes Feld an, hier gekennzeichnet durch die Kreuze in den Ecken. Im „Chase“-Modus streben die Gegner zu Feldern, die sich aus der Spielfigurposition und -blickrichtung berechnen lassen. Wichtig für den Spieler: An vier Stellen im Labyrinth (durchgekreuzte Pfeile) können die Gegner nicht nach oben abbiegen.

der HTML-Code im Browser dargestellt werden soll, bestimmen Stilvorlagen (CSS). Um beispielsweise die Schriftart Emulogic aus einer TTF-Datei zu laden und als Standard festzulegen, schreiben Sie:

```
@font-face {
  font-family: Emulogic;
  src: url(emulogic.ttf);
}
body { font-family: Emulogic; }
```

CSS-Definitionen wie diese speichern Sie am besten in einer separaten Datei und verknüpfen sie mit der Zeile

```
<link rel="stylesheet" type="text/css"
  href="game.css" />
```

im <head>-Bereich des HTML-Dokuments mit Selbigem.

So ähnlich ist auch mit dem in JavaScript geschriebenen Programmcode zu verfahren:

```
<script src="jquery.js" type="text/javascript"></script>
<script src="util.js" type="text/javascript"></script>
<script src="game.js" type="text/javascript"></script>
```

Diese Zeilen laden zuerst die ebenso nützliche wie verbreite JavaScript-Bibliothek jQuery, dann ein paar Hilfsfunktionen und anschließend den Spiel-Code.

Wichtig: Da JavaScript-Code typischerweise Änderungen am HTML-Elemente-Baum (Document Object Model, DOM) vornimmt – beim Klon zum Beispiel werden die Figuren und das Spielfeld erst zur Laufzeit generiert –, darf das erst nach dem Laden und Fertigstellen des HTML-Dokuments passieren. Das kann man garantieren, indem

```
var scene =
  'XXXXXXXXXXXXXXXXXXXXXXXXXXXX' +
  'X............XX............X' +
  'X.XXXX.XXXXX.XX.XXXXX.XXXX.X' +
  'XoXXXX.XXXXX.XX.XXXXX.XXXXoX' +
  'X.XXXX.XXXXX.XX.XXXXX.XXXX.X' +
  'X...........................X' +
  'X.XXXX.XX.XXXXXXXX.XX.XXXX.X' +
  'X.XXXX.XX.XXXXXXXX.XX.XXXX.X' +
  'X......XX....XX....XX......X' +
  'XXXXXX.XXXXX XX XXXXX.XXXXXX' +
  'XXXXXX.XXXXX XX XXXXX.XXXXXX' +
  'XXXXXX.XX          XX.XXXXXX' +
  'XXXXXX.XX XXX^^XXX XX.XXXXXX' +
  'XXXXXX.XX X      X XX.XXXXXX' +
  'tttttt.   X      X   .tttttt' +
  'XXXXXX.XX X      X XX.XXXXXX' +
  'XXXXXX.XX XXXXXXXX XX.XXXXXX' +
  'XXXXXX.XX          XX.XXXXXX' +
  'XXXXXX.XX XXXXXXXX XX.XXXXXX' +
  'XXXXXX.XX XXXXXXXX XX.XXXXXX' +
  'X............XX............X' +
  'X.XXXX.XXXXX.XX.XXXXX.XXXX.X' +
  'X.XXXX.XXXXX.XX.XXXXX.XXXX.X' +
  'Xo..XX.......  .......XX..oX' +
  'XXX.XX.XX.XXXXXXXX.XX.XX.XXX' +
  'XXX.XX.XX.XXXXXXXX.XX.XX.XXX' +
  'X......XX....XX....XX......X' +
  'X.XXXXXXXXXX.XX.XXXXXXXXXX.X' +
  'X.XXXXXXXXXX.XX.XXXXXXXXXX.X' +
  'X...........................X' +
  'XXXXXXXXXXXXXXXXXXXXXXXXXXXX';
var WIDTH = 28;
var HEIGHT = 31;
var playground = null;
var board = (function () {
  var a = [], x;
  for (x = 0; x < WIDTH; ++x)
    a.push(new Array(HEIGHT));
  return a;
})();
Playground.build = function() {
  var x, y, field, row;
  playground = $('#playground');
  playground.find('.food').remove();
  for (y = 0; y < HEIGHT; ++y) {
    row = y * WIDTH;
    for (x = 0; x < WIDTH; ++x) {
      field = scene[row + x];
      switch (field) {
        case 'X': // Wand
        case '^': // Gatter
        case 't': // Tunnel
          board[x][y] = field;
          break;
        case '.': // Futter
          board[x][y]=new FoodField(x, y);
          break;
        case 'o': // Kraftpille
          board[x][y]=new EnergizerField(x, y);
          break;
      }
      if (board[x][y] instanceof Field)
        playground.append(board[x][y].el);
    }
  }
}
```

Das Spiel verwendet die Variable board, um festzustellen, wo sich Wände, Tunnel und Futter befinden.

man die Initialisierungsfunktion als Callback des onload-Ereignisses angibt:

```
<body onload="Game.init()">
```

Moderner, sicherer und übersichtlicher, weil es JavaScript nicht mit HTML vermischt, ist es allerdings, die Initialisierungsfunktion per jQuery an das Ereignis zu binden:

```
$(document).ready(function () {
  var match = navigator.userAgent
    .match(/Chrome\/(\d+)/);
  if (match !== null && match.length > 0
    && parseInt(match[1]) >= 14)
    Game.init();
  else
    $(document.body).html('<p class="fatal">' +
      '<a href="http://www.google.de/chrome/" ' +
      'target="_blank">Chrome</a> &gt; 14 ' +
      'erforderlich</p>');
});
```

Das liefert eine Fehlermeldung, wenn der Code nicht im Chrome-Browser mit einer Versionsnummer größer oder gleich 14 ausgeführt wird. Warum das so sein muss, erfahren Sie in ein paar Absätzen.

Die Funktion Game.init() initialisiert das Spiel. Zuerst lädt die als Closure [2] in Game eingeschlossene Funktion loadImages() alle Grafiken, damit sie verzögerungsfrei zur Verfügung stehen, sobald sie benötigt werden:

```
function loadImages(callback) {
  var images = [ /* Dateinamen zu ladender Bilder */ ];
  var imagesLoaded = 0;
  $.each(images, function(i, filename) {
    var img = new Image;
    img.onload = function() {
      if (++imagesLoaded == images.length)
        callback.call();
    };
    img.src = 'img/' + filename;
  })
}
```

Der Mechanismus mit dem onload-Handler stellt sicher, dass das Callback erst aufgerufen wird, wenn alle Bilder im Speicher sind.

Musik

Das Callback ist die Funktion loadSounds(), die nach einem ähnlichen Schema die Sounds initialisiert:

```
function loadSounds(callback) {
  var NUM_SOUNDS = Object.keys(sounds).length;
  audioCtx = new AudioContext;
  gainNode = audioCtx.createGain();
  gainNode.gain.value = 0.5;
  gainNode.connect(audioCtx.destination);
  $.each(sounds, function (name, sound) {
    audioCtx.decodeAudioData(Base64Binary
      .decodeArrayBuffer(sound.base64),
        function (buffer) {
          sound.buffer = buffer;
          if (++soundsLoaded === NUM_SOUNDS)
            callback.call();
        });
  });
}
```

Die Sounds werden nicht aus Dateien geladen, sondern aus Base64-kodierten Daten, die in den JavaScript-Quelltext eingebettet sind. Das hat schlicht den Grund, dass mit diesem Verfahren zwar ein Drittel mehr Daten geladen werden müssen, aber das in einem Rutsch und nicht aus neun verschiedenen Dateien. Gerade die Latenzen beim Anfordern einer Datei vom Webserver drücken auf die Ladegeschwindigkeit.

Die Daten enthält das Array sounds:

```
var sounds = {
  'beginning': {
    buffer: null,
    base64: '/+OAxAAAAAAAAAAAAEluZm8...'
  },
  // weitere Sounds ...
};
```

Wenn Sie in eigenen Projekten Base64-kodierte Daten aus binären erzeugen wollen, können Sie dafür einen der zahlreichen Online-Konverter oder das Kommandozeilenwerkzeug base64 benutzen (siehe c't-Link am Artikelende).

Das Dekodieren erledigt die Funktion Base64Binary.decodeArrayBuffer() aus util.js, die im Unterschied zum JavaScript-eigenen btoa() nicht einen String erzeugt, sondern einen ArrayBuffer. Einen solchen erwartet die Funktion decodeAudioData(), die die darin enthaltenen WAV-, MP3-, OGG- oder AAC-kodierten Sounds in ein internes Format (PCM) wandelt.

Diese Funktion gehört zu einem Objekt vom Typ AudioContext aus dem recht jungen Web Audio API [3]. Das ist der Grund dafür, dass mein Spiel nur in Chrome 14 oder neuer läuft: Ältere Versionen oder andere Browser haben diese Programmierschnittstelle nicht implementiert.

Schade, denn sie glänzt mit einer für Spiele wichtigen Eigenschaft: Sie kann Sounds besser als das HTML5-Element <audio> nahezu latenzfrei abspielen. Das ging bislang nur mit der JavaScript-Bibliothek Soundmanager 2, die allerdings das Flash-Plug-in benötigt.

Mit dem Web Audio API lassen sich darüber hinaus Sounds aus verschiedenen Quellen mixen und durch kaskadierbare Filter schicken. Das Spiel verwendet nur den Gain-Filter, der im obigen Code-Ausschnitt die Lautstärke exemplarisch auf 50 Prozent setzt und das Ausgangssignal in die Soundausgabe (audioCtx.destination) weiterleitet.

Damit ist die Senke vorgegeben, fehlt noch die Quelle (AudioBufferSourceNode). Der Aufruf der Methode createBufferSource() erzeugt sie, setzt ihren Klangdatenpuffer auf den zuvor mit decodeAudioData() gefüllten und füttert damit den Gain-Knoten:

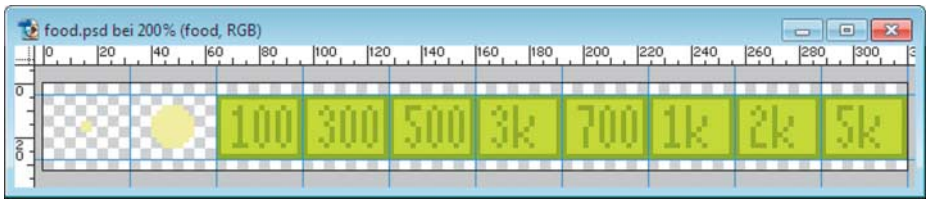
```
function playSound(name, loop) {
  var source = audioCtx.createBufferSource();
  var sound = sounds[name];
  source.buffer = sound.buffer;
  source.connect(gainNode);
  source.loop = loop || false;
  source.start(0);
}
```

Die Methode start() spielt den Sound ab. Ihr einziges Argument legt fest, um wie viele Sekunden das Abspielen verzögert werden soll. Die Eigenschaft loop muss true sein, wenn der Sound in einer Endlosschleife laufen soll.

Man kann einen AudioBufferSourceNode übrigens kein weiteres Mal zum Abspielen desselben Sounds verwenden, sondern muss wie im Beispiel stets einen neuen erzeugen.

Architektur

Nach dem Sound wird das Spielfeld initialisiert (siehe die Funktion Playground.build() im



CSS-Sprites: Um nicht jeden Bonus separat als Bild laden müssen, was die Ladezeit durch Summierung der Latenzen erhöht, fasst sie food.png in einer einzigen Datei zusammen. Durch Verschieben des sichtbaren Teils des Hintergrunds in einem -Element kann man die jeweils gewünschten Ausschnitte einblenden.

Listing links). In der Spiellogik handelt es sich dabei um eine Fläche mit 28 Feldern in der Breite und 31 in der Höhe.

Die Extras sind -Elemente, die nach der Vorlage des als zweidimensionale Ebene interpretierten Strings scene auf den Feldern platziert werden. Der Vorteil dieses Verfahrens im Unterschied etwa zum Zeichnen auf einem <canvas>-Element: Wenn ein Futterstückchen weggeknabbert wird, entfernt man es einfach aus dem DOM und weg ist es vom Bildschirm; ein Neuzeichnen des <canvas> ist nicht erforderlich. Auch mit den gelegentlich erscheinenden Geldscheinen verfährt das Spiel so. Das leere Labyrinth ist indes nur eine daruntergelegte Grafik.

Ein weiterer Vorteil: Die vier Kraftpillen (Energizer) lassen sich bequem per CSS animieren. Im HTML-Code sieht die Pille rechts oben auf dem Spielfeld wie folgt aus:

```
<div class="field food energizer"
  style="left: 408px; top: 40px;"></div>
```

Das Aussehen ergibt sich aus den drei Stilvorlagen. field bestimmt, dass das Feld 32 Pixel breit und hoch ist sowie innerhalb des umgebenden -Elements absolut positioniert wird:

```
.field {
  width: 32px;
  height: 32px;
  position: absolute;
}
```

food legt fest, dass der Hintergrund mit Pixeln aus einer PNG-Datei bemalt wird:

```
.food {
  background-image: url(food.png);
  background-position: 0px 0px;
  z-index: 2;
}
```

In der PNG-Datei befinden sich sämtliche Futtersymbole nebeneinander, die Kraftpille an der Position 32 Pixel von links, repräsentiert durch den Stil energizer:

```
.food.energizer {
  background-position-x: -32px;
  animation: energizer-anim 0.34s steps(2) infinite;
}
```

Der Stil referenziert eine sich unendlich lang wiederholende, in zwei Schritten ablaufende Animation, die die Pille im Rhythmus von 340 Millisekunden ein- und ausblendet:

```
@keyframes energizer-anim {
  from { visibility: hidden; }
  to { visibility: visible; }
}
```

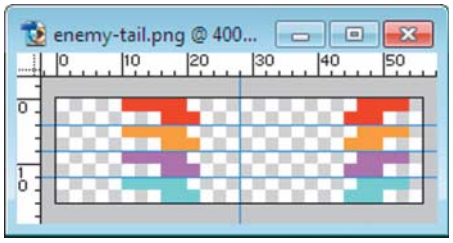
Parallel zu den HTML-Elementen entsteht das zweidimensionale Array board mit Informationen darüber, ob sich auf einem Feld eine Mauer oder Futter befindet, ob das Feld begehbar ist und ob es im Tunnel liegt. Letzteres ist wichtig, weil die Gegner sich langsamer bewegen sollen, solange sie sich im Tunnel befinden.

Die Spiellogik verwendet board, um den Überblick zu behalten, wo noch ein Futterstück liegt. Wird es gefressen, verwendet das Spiel die Referenz darauf, um es mit der jQuery-Funktion remove() aus dem DOM zu löschen. Das geht schneller, als nach dem betreffenden Element im DOM zu suchen.

Gehalten wird diese Referenz von Objekten des Typs FoodField und EnergizerField, die von Field erben, das wiederum von Vec2 abgeleitet ist. Dabei handelt es sich um ein Objekt zur Verwaltung von 2D-Koordinaten:

```
var Movable = function (x, y) {
  Vec2.call(this, x, y);
  this.el = null;
  this.name = undefined;
  this.startingFieldPos = new FieldPosition;
  this.fieldPos = new FieldPosition;
  this.width = NaN;
  this.height = NaN;
  this.defaultDirection = Direction.NONE;
  this.direction = Direction.NONE;
  this.speed = gameSpeed; // pixel/sec
  this.moving = false;
  this.place(x, y);
}
Movable.prototype = new Vec2;
Movable.prototype.place = function (_x, _y) {
  var x = (_x > 448)? 1 : ((_x < 1)? 447 : _x);
  var y = (_y > 496)? 1 : ((_y < 1)? 495 : _y);
  Vec2.prototype.set.call(this, x, y);
  this.fieldPos.set(Math.floor(this.x / 16),
                    Math.floor(this.y / 16));
  if (this.el)
    this.el
      .css('left', Math.floor(this.x -
           this.width / 2) + 'px')
      .css('top', Math.floor(this.y -
           this.height / 2) + 'px');
}
Movable.prototype.placeOnField = function(x, y) {
  this.place(16 * x + 8, 16 * y + 8);
}
Movable.prototype.moveBy = function (xd, yd) {
  this.place(this.x + xd, this.y + yd);
}
```

Der Prototyp Movable legt die Eigenschaften und Funktionen beweglicher Objekte fest, hier ein Auszug der wichtigsten.




```
.enemy {
  width: 28px;
  height: 26px;
  display: inline-block;
  position: absolute;
}
.torso {
  width: 28px;
  height: 22px;
  display: inline-block;
  position: absolute;
  top: 4px;
  left: 0;
  background-image: url(enemy-torso.png);
}
.torso.pinky {
  background-position-x: -56px;
}
.pinky.bored {
  animation: bored-torso-anim 3.0s steps(4)
    infinite;
}
@keyframes bored-torso-anim {
  from {
    background-position-y: 0;
  }
  to {
    background-position-y: -88px;
  }
}
.tail {
  width: 28px;
  height: 4px;
  display: inline-block;
  position: absolute;
  top: 0;
  left: 0;
  background-image: url(enemy-tail.png);
}
.tail.pinky {
  background-position-y: 8px;
  animation: tail-anim 0.25s steps(2) infinite;
}
@keyframes tail-anim {
  from {
    background-position-x: 0px;
  }
  to {
    background-position-x: -56px;
  }
}
```

```
<div class="enemy" id="pinky">
  <div class="torso pinky bored"></div>
  <div class="tail pinky"></div>
</div>
```

```
var Vec2 = function (x, y) {
  this.set(x, y);
}
Vec2.prototype.set = function (x, y) {
  this.x = x || 0;
  this.y = y || 0;
  return this;
}
```

Im Falle eines Objekts vom Typ Field oder eines, das von Field erbt, bezeichnen x und y die Feld-Koordinaten. Daraus berechnet der Konstruktor die absolute Position des Elements innerhalb des Spielfelds und generiert das gewünschte -Element:

```
var Field = function (x, y, cls) {
  Vec2.call(this, x, y);
  this.el = $('<div class="field"></div>')
    .css('left', (this.x * 16 - 8) + 'px')
    .css('top', (this.y * 16 - 8) + 'px');
  if (typeof cls === 'string')
    this.el.addClass(cls);
}
```

Im Konstruktor von FoodField und EnergizerField sieht man, wofür der dritte Field-Konstruktor-Parameter gut ist. Er fügt den übergebenen String als CSS-Klasse dem -Element hinzu:

```
var FoodField = function (x, y, points, extra) {
  Field.call(this, x, y, 'food' +
   ((typeof extra === 'string') ? ' ' + extra : ''));
  this.digestible = true;
  this.points = points || 10;
}
FoodField.prototype = new Field;
var EnergizerField = function (x, y) {
  FoodField.call(this, x, y, 50, 'energizer');
}
EnergizerField.prototype = new FoodField;
```

Zur Komplettierung fehlen jetzt nur noch die Figuren.

Figuren

Im Unterschied zu den statischen Field-Objekten können sie sich bewegen und basieren daher auf einem anderen Prototypen, nämlich Movable (siehe Listing auf S. 121).

An den Wertebereichen von x und y in der Methode place() lässt sich erahnen, dass Movables nicht auf Feldern verankert sind, sondern sich frei darüber bewegen können, begrenzt lediglich von den Wänden, die das Spielfeld umschließen. Nur im Tunnel gelangen sie an den äußersten rechten und linken Rand; überschreiten sie diesen, erscheinen sie auf der gegenüberliegenden Seite.

Für spätere Berechnungen merkt sich ein Movable die Feldposition in der Variablen fieldPos. Sie ist vom Typ FieldPosition, der auf Vec2 fußt. Die Geschwindigkeit in Pixeln pro Sekunde, mit der sich das Movable bewegen soll, steht in speed, die aktuelle Ausrichtung (UP, RIGHT, DOWN, LEFT und NONE) in direction.

Das in Movable deklarierte Feld startingFieldPos für die Koordinaten des Startfelds sowie width und height für die Breite und Höhe der Figur werden in den Konstruktoren der konkreten Movable-Implementierungen gesetzt:

Die Gegner setzen sich aus zwei -Elementen zusammen. Deren Hintergründe sind Ausschnitte aus einer Grafikdatei. Mit einer Animation wie „bored-torso-anim“ verdreht der Gegner gelangweilt die Augen, „tail-anim“ sorgt für das Rotieren des Bömmels.

```
var Player = function () {
  this.el = $('<div></div>').attr('id', 'player');
  this.name = 'player';
  this.startingFieldPos = new FieldPosition(13.5, 23);
  // ...
}
var Enemy = function (param) {
  this.name = param.name;
  this.startingFieldPos = param.startingFieldPos;
  this.width = 28;
  this.height = 26;
}
```

Die Player-Instanz entsteht in makePlayer():

```
player = new Player;
Playground.playground().append(player.el);
```

Die Gegner generiert makeEnemies():

```
$.each([
  { name: 'blinky',
    startingFieldPos: new FieldPosition(13.5, 11) },
  { name: 'inky',
    startingFieldPos: new FieldPosition(11.5, 14) },
  { name: 'pinky',
    startingFieldPos: new FieldPosition(13.5, 14) },
  { name: 'clyde',
    startingFieldPos: new FieldPosition(15.5, 14) }
], function (i, e) {
  var enemy = new Enemy(e);
  enemies.push(enemy);
  Playground.playground()
    .append(enemy.el);
});
blinky = enemy[0];
inky = enemy[1];
pinky = enemy[2];
clyde = enemy[3];
```

Spieleschleife

Nun kann das Spiel beginnen. Im Browser laufen Spiele wie Animationen in einer Schleife ab, in der sich folgende drei Schritte endlos wiederholen:
– Zustände aktualisieren,
– zeichnen,
– warten (Tastendrücke, Mausbewegungen et cetera).
Diese Funktionen kann man allerdings nicht einfach in einer Schleife wie

```
while (game.running()) { /* ... */ }
```

ausführen. Das liegt daran, dass der Browser Veränderungen am DOM nur darstellt, wenn er gerade keinen JavaScript-Code abarbeitet. Der Spiel-Code muss also zurückkehren, nachdem er die ersten beiden Schritte vollzogen hat. Die while-Schleife tut das nicht.

Nun könnte man auf die Idee kommen, per setInterval() zyklisch eine Funktion aufzuru-

fen, die die beiden Schritte vereint, zum Beispiel 60 Mal pro Sekunde, um flüssige Bewegungen zu ermöglichen. Das aber hätte den Nachteil, dass der Browser im 1/60-Sekunden-Takt zum Neuberechnen und -zeichnen gezwungen würde, auch dann, wenn es nicht notwendig ist.

Besser geht das mit der Funktion requestAnimFrame(). Ihr Aufruf mit einer Callback-Funktion als einziges Argument signalisiert dem Browser, dass er das Callback bei nächster Gelegenheit (typischerweise etwa jede sechzigstel Sekunde) ausführen soll. Bis dahin kann der Browser das eventuell geänderte DOM neu darstellen, Tastendrücke auswerten und auf Mausbewegungen reagieren.

Auch wenn das Spiel davon nicht betroffen ist, weil es nur im Chrome läuft, ist es für andere Spiele wichtig, für requestAnimFrame() einen Wrapper zu basteln, der den Namen browserübergreifend vereinheitlicht und nötigenfalls sogar die Funktion mit setTimeout() emuliert, weil nicht alle Browser requestAnimFrame() unter diesem Namen implementieren:

```
window.requestAnimationFrame =
  window.requestAnimationFrame ||
  window.webkitRequestAnimationFrame ||
  window.mozRequestAnimationFrame ||
  window.oRequestAnimationFrame ||
  window.msRequestAnimationFrame ||
  function (callback) {
    window.setTimeout(callback, 1000 / 60);
  };
```

Wenn man sie wie in der Funktion update() einsetzt (siehe Listing auf S. 124), entsteht daraus die Spielschleife.

Dem Callback übergibt sie beim Aufruf einen Zeitstempel (DOMHighResTimeStamp), mit dem seit dem Laden der Webseite verstrichenen Millisekunden, und zwar laut Spezifikation auf mindestens 10 µs genau. Daraus lässt sich ermitteln, wie viel Zeit zwischen zwei update()-Aufrufen vergangen ist. Diese Information benötigen die Figuren, um aus ihrer Geschwindigkeit (Membervariable speed) die zurückzulegende Strecke zu berechnen.

Das ist simpel, wie die Implementierung des Verhaltens der Spielfigur bei jeder Zustandsaktualisierung zeigt: In Abhängigkeit von der Laufrichtung wird die aktuelle Position innerhalb des Spielfelds um die berechnete Strecke versetzt – allerdings nur, falls sich in dieser Richtung keine Wand befindet. Bei Richtungsänderungen wird das -Element der Spielfigur entsprechend gedreht, sodass sie immer in die Richtung ihr Maul öffnet, in die sie läuft.

```
var lastTimeStamp = NaN;
function update(timeStamp) {
  if (paused)
    return;
  if (isNaN(lastTimeStamp)
    lastTimeStamp = timeStamp;
  var secs = (timeStamp - lastTimeStamp)
    / 1000;
  player.update(secs);
  for (var i = 0; i < enemies.length; ++i)
    enemy[i].update(secs);
  displayScore();
  if (player.score > HighScore.get()) {
    HighScore.set(player.score);
    displayHighScore();
  }
  lastTimeStamp = timeStamp;
  animationFrameID = requestAnimationFrame
    (update);
}
```

Erreicht die Spielfigur ein Feld mit Futter, verleibt sie es sich ein, was den Aufruf der Methode munch() nach sich zieht:

```
Player.prototype.munch = function (food) {
  this.score += food.points;
  $.event.trigger({
    type: 'munched',
    message: food });
  if (food.points == 10)
    this.delayCycles = 1;
  else if (food.points == 50)
    this.delayCycles = 3;
}
```

Damit wird auch klar, was es mit der Abfrage auf delayCycles am Anfang von Player.update() auf sich hat: Das Fressen von Futter bremst die Spielfigur für ein oder drei Zyklen aus, was sie langsamer und damit leichter zum Opfer der verfolgenden Gegner macht.

An dieser Stelle könnte der Code einfach den passenden Sound abspielen und das Futterstück vom Spielfeld entfernen sowie ein Extraleben spendieren, falls der neue Punktestand über 10 000 oder 100 000 liegt, aber das würde die Veränderung globaler Variablen bedingen. Für mehr Übersichtlichkeit und weniger Abhängigkeiten zwischen Objekten sollen daher nur globale Funktionen globale Variablen ändern dürfen. Für die bestmögliche Entkopplung des Player-Objekts vom globalen Kontext wird die verarbeitende Funktion daher nicht direkt aufgerufen, sondern durch das Auslösen eines Ereignisses mit $.event.trigger().

Der Empfänger des Ereignisses „munched“ ist das Browser-Fenster. Die Verknüpfung hat die Funktion Game.init() bereits hergestellt:

```
$(window).on({
  munched: munched,
  levelcomplete: levelComplete,
  playerkilled: playerKilled,
  frightenedenemykilled: frightenedEnemyKilled,
  keydown: function (e) {
    switch (e.keyCode) {
      case 38 /* KeyUp */:
        player.turnUp();
        break;
      // weitere Tasten
    }
    return true;
  }
});
```

Vergleichbare Signalisierungen finden auch dann statt, wenn ein Level zu Ende ist, die Spielfigur von einem Gegner gefressen wurde oder sie einen Gegner gefressen hat.

Die update()-Funktion ist der Kern der Spielschleife. Sie aktualisiert die Zustände der Figuren und den Punktestand.

Letzteres Ereignis wird von der Funktion Player.update() ausgelöst (siehe Listing unten): Sie schaut in einer Schleife nach, ob die Spielfigur auf demselben Feld wie ein Gegner steht und dieser Gegner gerade „erschrocken" ist. Dann gilt der betreffende Gegner nämlich als gefressen und nur seine Augen kehren zurück zum Haus – und der Spieler heimst 200, 400, 800 oder 1600 Punkte ein.

Der obige Code-Schnipsel zeigt außerdem, wie man Tastendrücke auswertet: Man bindet einfach das Ereignis „keydown" an einen Handler und unterscheidet zum Beispiel per switch/case nach den Tastencodes. Nach gleichem Muster kann man auch auf Mausereignisse reagieren („click", „mouseenter", „mousemove" …).

Spiellogik

Die originalen Pac-man-Geister wandern nicht ziellos umher, sondern bewegen sich nach starren Regeln. Die sind allerdings so ausgefuchst, dass das Verhalten der Geister (hier: Schlapphüte) gleichermaßen natürlich wie schwer kalkulierbar wirkt. Die folgenden Ausführungen beziehen sich auf Gegner im Normalzustand, also solange sie nicht erschrocken sind oder auf dem Weg nach Hause:

– Ein Gegner kehrt niemals um (keine 180°-Wende).
– Ein Gegner entscheidet zwei Felder im Voraus, wohin er an der nächsten Kreuzung abbiegt.
– Ein Gegner biegt zu dem Feld ab, das seinem aktuellen Zielfeld am nächsten liegt.
– Das Zielfeld ändert sich mit dem Zielmodus („scatter" oder „chase"), der von der seit Level-Start verstrichenen Zeit abhängt.

```
Player.prototype.update = function (secs) {
  if (this.delayCycles > 0) {
    --this.delayCycles;
    return;
  }
  var i, enemy, d, field, dx, dy;
  d = secs * this.speed;
  dx = this.x % 16;
  dy = this.y % 16;
  switch (this.direction) {
    case Direction.UP:
      if (this.fieldPos.canGoUp() || dy > 8)
        this.moveBy(8 - dx, -d);
      if (this.direction!=this.lastDirection) {
        this.el.css('transform',
                    'rotate(90deg)');
        this.lastDirection = this.direction;
      }
      break;
    case Direction.RIGHT:
      // ...
    case Direction.DOWN:
      // ...
    case Direction.LEFT:
      // ...
    case Direction.NONE:
      // ...
  }
  field = Playground.field(this.fieldPos.x,
                           this.fieldPos.y);
  if (field instanceof FoodField)
    this.munch(field);
  for (i = 0; i < enemies.length; ++i) {
    enemy = enemies[i];
    if (enemy.isFrightened() &&
        this.fieldPos.equals(enemy.fieldPos))
      $.event.trigger({
        type: 'frightenedenemykilled',
        message: enemy });
  }
}
```

Im Modus SCATTER (zerstreuen) streben die Gegner zu ihrem jeweiligen Scatter-Feld. Das ist ein Feld außerhalb des Spielfelds, und zwar für jeden Gegner in einer anderen Ecke, sodass sie sich in diesem Modus dorthin bewegen und in dessen Nähe umherkreisen.

Im Modus CHASE (verfolgen) jagen sie den Spieler. Der rote Gegner Blinky wählt als Ziel die momentane Position der Spielfigur. Der orangefarbene Gegner Clyde tut das auch, allerdings versucht er dabei, dem Spieler nicht näher als acht Felder zu kommen (Manhattan-Distanz). Pinky (seine Farbe kann man sich denken) steuert das Feld an, das vier Felder vor dem Spieler liegt. Durch einen Fehler im Original-Pac-man-Code greift diese Regel nicht, wenn sich Pac-man nach oben bewegt. Dann nimmt Pinky das Feld vier Positionen links und vier Positionen oberhalb von Pac-man. Meine Implementierung ahmt dieses Verhalten nach.

Komplizierter wirds bei Inky, der sich wie folgt entscheidet (siehe die Funktion Enemy.update()):

```
target = player.fieldPos.clone();
switch (player.direction) {
  case Direction.UP:
    target.x -= 2; target.y -= 2; break;
  case Direction.RIGHT:
    target.x += 2; break;
  case Direction.DOWN:
    target.y += 2; break;
  case Direction.LEFT:
    target.x -= 2; break;
}
target = target.sub(blinky.fieldPos)
  .mul(2).add(blinky.fieldPos);
```

Demnach orientiert sich Inky ähnlich wie Pinky zunächst an einem Feld, das zwei Felder vor der Spielfigur liegt, mit der gleichen Ausnahme für den Fall, dass sie nach oben geht. Dann denkt sich Inky eine Linie von Blinky zu diesem Feld und verdoppelt ihre Länge. Das Ende dieser Linie bestimmt das Zielfeld. Ob es außerhalb des Spielfelds liegt, spielt keine Rolle. Die Methoden sub(), mul() und add() gehören zu FieldPos und führen Rechenoperationen auf Vektoren aus (Subtraktion, Skalarmultiplikation und Addition).

Die Bewegungsmuster der Gegner zu kennen hilft nicht nur dabei, ein möglichst originalgetreues Spiel zu programmieren, sondern auch beim Spielen selbst. Ein Beispiel aus der Praxis: Die Spielfigur kann sorglos auf Pinky zulaufen, ohne die Gefahr mit ihm zu kollidieren, wenn ihr Abstand nicht mehr als vier Felder beträgt und es zwischen ihnen eine Abzweigung gibt. Denn Pinky wird diese nehmen; auf die Spielfigur zugehen würde ihn weiter von seinem Zielfeld entfernen.

Gesteuert von requestAnimFrame() wird Player.update() bis zu 60 Mal in der Sekunde aufgerufen. Abhängig von der seit dem letzten Aufruf verstrichenen Zeit und der Laufrichtung berechnet sie seine Position und stellt daraufhin fest, ob die Spielfigure auf dem aktuellen Feld Futter gefunden hat.

Die einzige Ausnahme von der Regel, dass Gegner keine Kehrtwende machen, ist der Augenblick, in dem die Spielfigur eine der vier Kraftpillen futtert und damit die Gegner in Angst und Schrecken versetzt. Durch Umschalten auf den CSS-Stil „frightened" erhalten die Gegner ein anderes Aussehen und irren für eine Weile umher, indem sie an allen Abzweigungen per Zufall entscheiden, wo es weitergehen soll.

Zeitmaschine

Wie lange sie erschrocken sind und wie oft sie vor der Rückkehr in den Normalzustand blinken, das heißt ihre Farbe auf Weiß und zurück auf Blau wechseln, hängt vom Level ab. Die Variable Specs hält die Daten für alle 255 Levels bereit; ab dem 22. Level bleiben die letzten Einstellungen erhalten:

```
var Specs = {
 /* Level */ 1: {
  bonus: 'cherry',
  points: 100,
  playerSpeed: 80,
  enemySpeed: 75,
  enemyTunnelSpeed: 40,
  elroyDotsLeft1: 20,
  elroySpeed1: 80,
  elroyDotsLeft2: 10,
  elroySpeed2: 85,
  frightenedPlayerSpeed: 90,
  frightenedPlayerDotsSpeed: 79,
  frightenedEnemySpeed: 50,
  frightenedTime: 6000,
  numFlashes: 5 },
 // Spezifikationen für Levels 2 bis 21 ...
};
```

bonus zeigt an, welcher Bonus auf den Spieler wartet, und points, wie viele Punkte er fürs Einheimsen bekommt. Der Bonus erscheint zufallsgesteuert für 9 bis 10 Sekunden, wenn noch 174 und 74 Futterstücke übrig sind (siehe die Funktionen placeFruit() und Level.decreaseNumberOfDots()).

playerSpeed gibt an, mit wie viel Prozent der in gameSpeed eingestellten Pixel pro Sekunde der Spieler sich bewegt. Gleiches gilt für die Gegner mit enemySpeed (normal) und enemyTunnelSpeed (im Tunnel). Nach dem Fressen einer Kraftpille gelten die korrespondierenden mit frightened… benannten Werte. Wie lange dieser Zustand in Millisekunden andauert, bestimmt frightenedTime. Wie oft die Gegner blinken – so sie denn noch leben –, gibt numFlashes an.

Vorsicht ist für den Spieler geboten, wenn nur 20 beziehungsweise 10 Futterstücke übrig sind (elroyDotsLeft1 und elroyDotsLeft2). Dann nämlich mutiert Blinky zum Berserker „Elroy", der ungeachtet des aktuellen Bewegungsmodus die Spielfigur verfolgt: Erst wird er so schnell wie die Spielfigur, dann fünf Prozentpunkte schneller.

Die Werte in Specs bleiben den ganzen Level über unverändert. Die Bewegungsmodi der Gegner ändern sich indes mit der Zeit nach den Vorgaben in der Variable EnemyModes:

```
function munched(e) {
  var food = e.message;
  if (food instanceof EnergizerField) {
    playSound('eatenergizer');
    frightenAll();
    Level.decreaseNumberOfDots();
  }
  else if (food.points == 10) {
    playSound('chomp');
  }
  else {
    playSound('eatfruit');
    var points = new Field(player.fieldPos.x,
      player.fieldPos.y, 'points p' + food.points);
    Playground.playground().append(points.el);
    setTimeout(function () { points.remove(); }, 2000);
  }
  Playground.removeField(food);
  food.remove();
  if (player.score >= 10000 && extraPlayerAvailable[10000]) {
    extraPlayerAvailable[10000] = false;
    giveBirthToExtraPlayer();
  }
  else if (player.score >= 100000 &&
           extraPlayerAvailable[100000]) {
    extraPlayerAvailable[100000] = false;
    giveBirthToExtraPlayer();
  }
}
```

Beim Auslösen eines Ereignisses kann man dem Handler im zweiten Parameter von $.event.trigger() eine Nachricht mitgeben. Beim Ereignis „munched“ ist das ein Futterobjekt, das unter anderem die durch Fressen erzielbaren Punkte enthält.

```
var EnemyModes = {
 /* Level */ 1: [
  { mode: Enemy.Mode.SCATTER, duration: 7 },
  { mode: Enemy.Mode.CHASE, duration: 20 },
  { mode: Enemy.Mode.SCATTER, duration: 7 },
  { mode: Enemy.Mode.CHASE, duration: 20 },
  { mode: Enemy.Mode.SCATTER, duration: 5 },
  { mode: Enemy.Mode.CHASE, duration: 20 },
  { mode: Enemy.Mode.SCATTER, duration: 5 },
  { mode: Enemy.Mode.CHASE, duration: Infinity }
 ],
 // Bewegungsmodi für Levels 2 bis 5
};
```

Das heißt, dass die Gegner in den ersten 7 Sekunden in ihre Ecken streben, dann für 20 Sekunden die Spielfigur verfolgen und so weiter, um sie schließlich ohne Unterlass bis zum Level-Ende zu jagen.

Das Geniale an der Steuerung über diese beiden recht einfachen Parameterlisten: Nur durch Erhöhen von Geschwindigkeiten sowie Verkleinern von Werten, die für den Spieler günstig sind („Frightened“-Dauer, „Scatter“-Dauer) und Verlängern jener, die für ihn ungünstig sind (Blink-Zahl, „Chase“-Dauer), ist es den Spiel-Erfindern gelungen, die Schwierigkeit sukzessive von Level zu Level und sogar innerhalb der Levels zu steigern.

Zeitliche Abläufe wie das Verschwindenlassen der Boni, das Ändern des Gegner-Bewegungsmodus sowie das Zurücksetzen der Gegner von „erschrocken“ auf „normal“ werden per setTimeout() gesteuert. Am kompaktesten lässt sich die Verwendung der Funktion daran zeigen, wie die mit dem Fressen eines Bonus oder Gegner erzielten Punkte für zwei Sekunden erscheinen (siehe die Funktion munched() im Listing oben).

Highscores

Wie der echte Arcade-Automat soll sich auch meine Implementierung die erzielten Highscores dauerhaft merken. Übergangsweise speichert sie das Spiel intern als Objekt mit dem Pseudonym des jeweiligen Spielers als Schlüssel und dem Highscore als Wert:

```
highscores[pseudonym] = player.score;
```

Ablegen kann man solche Werte zum Beispiel in einem Cookie, damit sie beim nächsten Aufruf der Webseite wieder zur Verfügung stehen. Zeitgemäßer ist allerdings die Verwendung lokalen Speichers. Mit localStorage.setItem(key, value) setzt man den Schlüssel key auf den Wert value. Der Wert darf nur von einem primitiven Typ sein, also Ganzzahl, String oder Boolean, aber kein Objekt. Das muss man vorher in einen String wandeln, zum Beispiel, indem man es als JSON-Literal serialisiert:

```
localStorage.setItem('HighScores',
  JSON.stringify(highscores))
```

Das Deserialisieren geht per:

```
highscores = JSON.parse(
  localStorage.getItem('HighScores')
) || {}
```

Falls getItem() den Wert null zurückliefert, sorgt die Oder-Verknüpfung mit dem leeren Objektliteral für die Initialisierung einer leeren Liste in highscores. Den vollständigen Code zur Verwaltung der Highscores finden Sie im gleichnamigen Objekt im Quelltext.

Ausblick

In einer künftigen Version werden Sie Ihren Highscore möglicherweise auf einen Webserver hochladen können, wo er dann nebst Ihrem Pseudonym für jedermann sichtbar ist. Bis dahin viel Spaß beim Retro-Zocken! Vergessen Sie darüber aber bitte nicht, ein anderes flottes Spiel nachzuprogrammieren.

(ola)

Literatur

[1] Jamey Pittman, The Pac-man Dossier: http://home.comcast.net/~jpittman2/pacman/pacmandossier.html

[2] Oliver Lau, Andreas Linke, Torsten T. Will, Variablen to go, Closures in aktuellen Programmiersprachen, c’t 17/13, S. 168

[3] Web Audio API: www.w3.org/TR/webaudio

[4] Gerhard Völkl, Rettet CeTman!, 3D-Browser-Spiele mit Three.js entwickeln, c’t 19/13, S. 176

www.ct.de/1323118 c't