KAPITOLA 6

Plnohodnotné kreslení

V této kapitole:
- Odbočka (terminologická): Yak Shaving
- HTML <canvas>
- Klíčové slovo this a pseudo-privátní vlastnosti
- Příprava canvasu a mřížka
- Buňky a atomy v nich
- Ostatní soubory

Předchozí kapitola nepřinesla žádné velké novoty, je tedy nejvyšší čas pustit se do vylepšování některých aspektů aplikace. Do plné hratelnosti zatím schází podpora pro více hráčů (a tím pádem i možnost umísťovat atomy různé barvy), počítání skóre (jde ruku v ruce s testováním, zdali už někdo vyhrál) a také klíčová herní mechanika – exploze těch polí, ve kterých počet atomů překročil kritické množství. Právě tato část bude patrně algoritmicky nejzajímavější, neboť exploze v jedné buňce může snadno odstartovat řetězovou reakci v buňkách dalších. Zároveň se dá ale očekávat, že půjde i o jednu z komplikovanějších komponent aplikace.

Odbočka (terminologická)

Yak Shaving

Ne vždy se chceme do hlavního problému vrhnout po hlavě a prát se s ním tak dlouho, než bude celý vyřešen. Někdy se nám prostě nechce, jindy máme dojem, že před hlavním *útokem* bude taktičtější chvíli chodit okolo a situaci obhlížet. Třeba si při řešení dílčích a doplňkových úkolů všimneme něčeho zajímavého, možná tak najdeme způsob, jak ten největší rozdělit na menší nebo úplně rozmělnit.

Takovému odkládání něčeho velkého se anglicky říká *Yak Shaving* (stříhání jaků). Nesmíme si tento osobitý termín plést s obyčejnou prokrastinací – při ní se věnujeme zcela nesouvisejícím činnostem. U vývojářského Yak Shavingu se stále soustředíme na společný cíl, ale snažíme se věnovat se doplňkovým úkolům, které by klidně mohly počkat.

Pro zvídavé: Tento termín vznikl počátkem devadesátých let celkem náhodně v návaznosti na díl animovaného seriálu *The Ren & Stimpy Show*, ve kterém figuroval *Yak Shaving Day* coby alterna-

tiva Štědrého dne. Je tedy dost pravděpodobné, že se autor termínu Carlin Vieri[3] nikdy žádného opravdového stříhání jaků neúčastnil.

Obrázek 6.1 Jak, nestříhaný. Foto © Dennis Jarvis

HTML <canvas>

Pojďme tedy otázku řetězové reakce ještě chvíli odkládat. Namísto toho můžeme ozkoušet, jak dobře jsme v předchozí kapitole provedli rozdělení kódu na komponenty – třeba tím, že zkusíme naimplementovat nějakou estetičtější formu vizualizace. Současné vypisování písmene **o** je přece nouzové řešení.

Pro potřeby kreslení je v jazyce HTML5 k dispozici značka `<canvas>`. Její vznik se datuje do roku 2004 a jedná se (s ohledem na ostatní HTML prvky) o poměrně revoluční techniku. Pomocí jednoduchého JavaScriptového rozhraní lze do canvasu (přímý překlad *plátno* se nepoužívá) kreslit velmi obdobně, jako se například maluje v tradičním programu Malování (MS Paint) ve Windows. HTML canvas je tedy rastrová oblast, nad kterou máme přesnou kontrolu až na úroveň jednotlivých pixelů.

Tatáž značka se používá také k realizaci technologie WebGL, to jest přístup k hardwarově akcelerovanému vykreslování pomocí jazyka OpenGL ES. Této technice se budeme krátce věnovat až v sedmnácté kapitole; pro naše účely zatím bohatě postačí využívání běžných dvourozměrných rozhraní.

3 *http://en.wiktionary.org/wiki/Talk:yak_shaving#Etymology*

Pojďme se nejprve podívat, jaké hlavní funkce jsou pro nás k dispozici :

- `getContext` vrací tzv. kontext – objekt, na kterém jsou volány všechny následující metody,
- `fillText` a `strokeText` pro kreslení písmen,
- `clearRect`, `fillRect` a `strokeRect` pro kreslení obdélníků,
- `getImageData` a `putImageData` pro změnu jednotlivých pixelů,
- `drawImage` pro vložení jiného obrázku,
- sada transformačních funkcí (`rotate`, `scale` a dalších),
- `beginPath` a související funkce pro nakreslení (a volitelně vyplnění) zcela obecné křivky.

Nakreslením celé hrací plochy pomocí jediné HTML značky (`<canvas>`) zcela skryjeme její strukturu pro ostatní komponenty aplikace. To pěkně koresponduje s požadavkem, že o zobrazování by se měla starat jen jedna vyhrazená část aplikace.

Přechodem na jinou zobrazovací technologii zároveň otestujeme, jak dobře jsme navrhli API pro komunikaci s komponentou `Draw`. V ideálním případě bychom vůbec nemuseli měnit signatury veřejných metod, jen jejich implementaci. Pro zbytek aplikace tak bude skutečnost, že jsme změnili vykreslování, úplně skryta.

Klíčové slovo this a pseudoprivátní vlastnosti

Při úpravách vykreslovací komponenty `Draw` zároveň odstraníme jeden z jejích dalších neduhů. Tento objekt obsahuje celou řadu metod a zdá se, že jejich počet budeme dále navyšovat. Přitom způsob jejich volání je nešťastný; když chceme nyní vykreslit jednu buňku, píšeme `Draw.atoms(...)`. Takovéto pevné umístění názvu objektu je nevhodné, protože nám například brání snadnému přejmenování. Také z kódu není na první pohled patrné, že se jedná o „naši vlastní“ metodu (tj. že `Draw` není nějaká oddělená komponenta). S vyčištěním kódu nám v tomto případě může pomoci klíčové slovo `this`.

Nejprve si ale musíme dobře připomenout, jak klíčové slovo `this` v JavaScriptu přesně funguje. Jedná se totiž o komplikovanou vlastnost jazyka, která je často pro programátory (zejména ty, kteří přichází z jiných objektových jazyků) notně zavádějící. V případě JavaScriptu není hodnota `this` známa ve chvíli parsování kódu, ale až ve chvíli jeho vykonání. Představme si tedy velmi triviální funkci *f*:

```
var f = function() {
  alert(this);
}
```

Dopředu není možné rozhodout, jakou hodnotu bude mít `this`. Existují k tomu tato pravidla:

1. Při **volání vpravo od tečky** je `this` rovno objektu vlevo od tečky. To odpovídá této situaci:

```
var obj = {};
obj.f = f;
obj.f(); // this == obj
```

2. Pokud chceme hodnotu `this` **explicitně nastavit**, voláme funkci pomocí jejích metod `call` a `apply`:

```
var obj = {};
f.call(obj);  // this == obj
f.apply(obj); // this == obj
```

   Rozdíl mezi `call` a `apply` je v tom, jak takto volané funkci předáváme parametry: u `call` jako další parametry tohoto volání, u `apply` ve druhém parametru coby pole.

3. Speciální metoda `bind` dovoluje z jedné funkce **vyrobit funkci druhou**, která tu první zavolá s hodnotou `this` pevně zadanou ve chvíli volání `bind`. Využití vypadá takto:

```
var obj = {};
var g = f.bind(obj);
g(); // ve funkci f je this == obj
```

   Zajímavé je, že nezáleží na tom, jakým způsobem nově vytvořenou funkci zavoláme. Metoda `bind` je sice standardní součástí jazyka (od verze ES5), ale dovedli bychom ji zjednodušeně nasimulovat sami:

```
f.bind = function(newThis) {
  return function() {
    return f.call(newThis);
  }
}
```

4. Pokud nepoužijeme žádný z předchozích způsobů a funkci zavoláme jen prostým **zapsáním jejího jména a kulatých závorek**, hodnota `this` je buď `undefined` (ve striktním režimu), nebo globální jmenný prostor (tj. v prohlížeči `window`). Touto formou volání se nebudeme hlouběji zabývat, protože nedává smysl – chceme-li ve funkci smysluplně využít `this`, musíme ji volat některým z předchozích zápisů.

Zde vidíme i podstatu terminologické nejednoznačnosti: JavaScriptové funkce nikomu *nepatří*, a tak o nich nelze jednoznačně říci, jestli se jedná o *metody* v objektovém slova smyslu. Metodami tedy nazýváme ty funkce, které máme v plánu volat pomocí operátoru tečky (a je v nich proto smysluplné použití `this`).

Se znalostí fungování `this` můžeme veškerá volání v rámci objektu přepsat tak, aby toto klíčové slovo využívala. Ruku v ruce s používáním `this` jde ale také problematika tzv.

privátních vlastností (a metod). Tyto jsou určeny výhradně k tomu, aby byly používány jen *zevnitř* objektu, tedy právě pomocí `this`. Do privátních vlastností ukládáme taková data (a kód), která jsou určena jen pro objekt samotný a ostatní k nim nesmí přímo přistupovat.

Situaci komplikuje skutečnost, že JavaScript žádné privátní vlastnosti nemá. Teoreticky je můžeme do jisté míry suplovat pomocí uzávěr a v nejnovější verzi jazyka ES6 je můžeme plnohodnotně nasimulovat pomocí objektu `WeakMap`. Vždy to bude ale speciální servisní logika navíc; v jazyce jako takovém jednoduše privátní vlastnosti nejsou. V praxi se nicméně často hodí odlišit veřejná a neveřejná rozhraní. My se proto budeme držet jednoduchého pravidla: Ty vlastnosti, jejichž název začíná podtržítkem, budeme považovat za soukromé a pokusíme se k nim přistupovat výhradně pomocí klíčového slova `this`. Pamatujme ale, že se jedná jen o naše vlastní pravidlo, které interpret jazyka nedokáže automaticky vynutit.

Mimochodem, čím více má objekt privátních rozhraní (na úkor těch veřejných), tím lépe se v něm provádějí změny. Při úpravách jeho privátních metod nemusíme brát ohled na případné další komponenty, které s ním spolupracují – proto je dobrý nápad definovat všechny metody zpočátku jako privátní a zveřejňovat je až dle potřeby.

Příprava canvasu a mřížka

Soubor `draw.js` změnou vykreslovací technologie notně nakyne. Pojďme se na něj tedy podívat po jednotlivých částech.

```
var Draw = {
  CELL: 60,
  LINE: 2,
  ATOM: 7,
  _context: null
};
```

V objektu `Draw` definujeme tři konstanty (skutečné konstanty jsou k dispozici až od verze ES6; nám postačí proměnné pojmenované velkými písmeny), kterými řídíme vzhled: velikost jedné buňky, tloušťka čáry, poloměr atomu (vše v pixelech). Do vlastnosti `Draw._context` uložíme 2D kontext canvasu, používaný pro všechny následné kreslicí operace.

```
/* Výroba canvasu a jeho příprava */
Draw.init = function() {
  var canvas = document.createElement("canvas");

  this.CELL += this.LINE;

  var size = Game.SIZE * this.CELL + this.LINE;
  canvas.width = size;
```

```
  canvas.height = size;

  this._context = canvas.getContext("2d");
  this._context.lineWidth = this.LINE;

  document.body.appendChild(canvas);

  this.all();
}
```

Metoda `Draw.init` slouží k výrobě canvasu a jeho nastavení. Jeho rozměry jsou dány počtem buňek, rozměrem buňky a tloušťkou oddělovací čáry. Hrací plochu chceme také orámovat, takže oddělovacích čar je v každé ose vždy o jednu více než buněk. S ohledem na tloušťku oddělovacích čar (a skutečnost, že jsme namísto opravdových konstant použili jen obyčejné proměnné) můžeme využít malý trik, kterým zvětšíme `Game.CELL` o `Game.LINE`. Pro potřeby všech dalších výpočtů se totiž hodí do rozměru buňky zahrnout i její jednostranné orámování.

Počet buněk jsme v rámci vylepšování přesunuli z definice hrací plochy (dříve `board.js`) do konstanty `Game.SIZE`. Lépe to odpovídá skutečnosti, že se jedná a veřejně užitečnou informaci.

```
/* Vykreslit celou hrací plochu */
Draw.all = function() {
  this._context.fillStyle = "#fff";
  var width = this._context.canvas.width;
  var height = this._context.canvas.height;

  this._context.fillRect(0, 0, width, height);

  this._lines();
  this._cells();
}
```

Vykreslení celé herní plochy je přímočará operace: nejprve celou oblast vyplníme bílou barvou, poté nakreslíme jednotlivé čáry, vposled pak všechny buňky (respektive atomy v nich). Za zmínku stojí, že `this._context.canvas` odpovídá použité HTML značce `<canvas>`, nemusíme si ji proto nikde nadbytečně ukládat.

```
/* Vykreslit mřížku */
Draw._lines = function() {
  this._context.beginPath();

  for (var i=0; i<Game.SIZE+1; i++) { // svislé
    var x = this.LINE/2 + i*this.CELL;
    this._context.moveTo(x, 0);
    this._context.lineTo(x, this._context.canvas.height);
  }
```

```
  for (var i=0; i<Game.SIZE+1; i++) { // vodorovné
    var y = this.LINE/2 + i*this.CELL;
    this._context.moveTo(0, y);
    this._context.lineTo(this._context.canvas.width, y);
  }

  this._context.stroke();
}
```

Pro oddělovací čáry a orámování využijeme *path*, tj. obecnou křivku. Ta naše bude sestávat z mnoha úseček: nejprve svislých, poté vodorovných. Jednotlivá volání `moveTo` a `lineTo` sama o sobě nic nevykreslují, pouze definují pohyb virtuálního *pera*; proto na konci nezapomene zavolat `this._context.stroke()` pro zobrazení nadefinovné křivky.

Buňky a atomy v nich

S klíčovým slovem `this` se nemusíme ostýchat dělit kód na větší množství krátkých funkcí. Pro vykreslení všech buněk tak máme velmi triviální metodu `Draw._cells`:

```
/* Vykreslit buňky s atomy */
Draw._cells = function() {
  for (var i=0; i<Game.SIZE; i++) {
    for (var j=0; j<Game.SIZE; j++) {
      this._cell(i, j, Board[i][j]);
    }
  }
}
```

Následuje metoda pro vykreslení jedné buňky:

```
/* Vykreslit jednu buňku */
Draw._cell = function(x, y, count) {
  x *= this.CELL;
  y *= this.CELL;
  switch (count) {
    case 1:
      this._atom(x + this.CELL/2, y + this.CELL/2);
    break;

    case 2:
      this._atom(x + this.CELL/4, y + this.CELL/4);
      this._atom(x + this.CELL*3/4, y + this.CELL*3/4);
    break;

    case 3:
      this._atom(x + this.CELL/2, y + this.CELL/2);
      this._atom(x + this.CELL/4, y + this.CELL/4);
      this._atom(x + this.CELL*3/4, y + this.CELL*3/4);
    break;
```

```
    case 4:
      this._atom(x + this.CELL/4,   y + this.CELL/4);
      this._atom(x + this.CELL*3/4, y + this.CELL*3/4);
      this._atom(x + this.CELL/4,   y + this.CELL*3/4);
      this._atom(x + this.CELL*3/4, y + this.CELL/4);
    break;
  }
}
```

V ní nejprve převedeme zadané souřadnice buňky na pixely (v rámci canvasu) tak, že je vynásobíme pixelovou velikostí buňky. Operátor `*=` je zkratka za `= původní hodnota *`; klidně si tak přitom přepíšeme hodnoty parametrů `x` a `y`. (Na tom není nic špatného; pro každý parametr máme k dispozici zcela obyčejnou lokální proměnnou a smíme ji dle libosti měnit.)

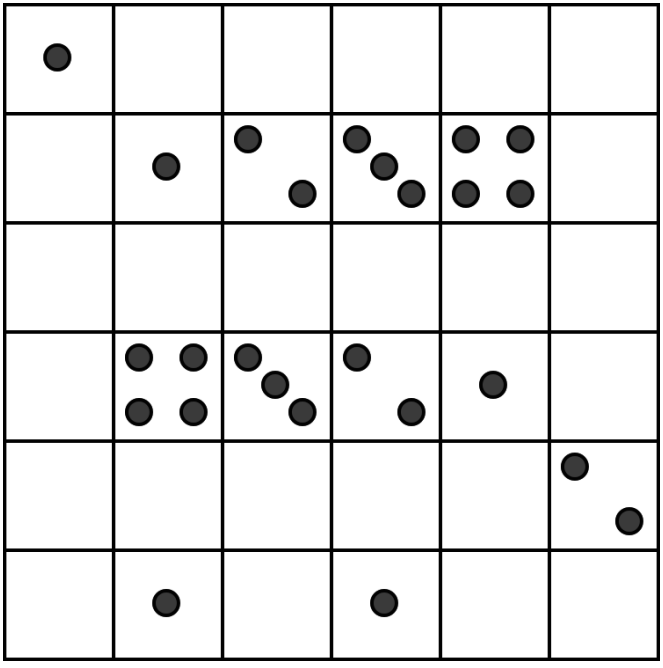
Pravidla hry říkají, že v buňce smí být nejvýše čtyři atomy (to je podkritické množství pro buňku mimo okraje plochy); pokud bychom přidali pátý, došlo by k reakci a snížení počtu. Stačí nám tedy rozhodnout o vizualizaci pro jeden až čtyři atomy v buňce. Čeká nás tak poslední kreslicí metoda:

```
/* Vykreslit jeden atom */
Draw._atom = function(x, y) {
  this._context.beginPath();

  this._context.moveTo(x+this.ATOM, y);
  this._context.arc(x, y, this.ATOM, 0, 2*Math.PI, false);

  this._context.fillStyle = "blue";
  this._context.fill();
  this._context.stroke();
}
```

Ta přijímá parametry v pixelech a má jediný úkol – vykreslit jeden atom. Opět použijeme křivku, tentokráte tvořenou jediným obloukem (anglicky *arc*) – tak velkým, že s ním obtáhneme celou kružnici. Parametry metody `arc` jsou: střed oblouku (dvě hodnoty), poloměr, počáteční a koncový úhel oblouku, směr (`true` = proti směru hodinových ručiček; v tomto případě je to jedno). Úhly se zadávají v radiánech, takže naše křivka odpovídá kružnici – ano, takto obskurně se v canvasu kreslí kolečko. Křivku pak nejen vyplníme (modrou barvou, `this._context.fill()`), ale i orámujeme. K tomu využijeme skutečnost, že výchozí barva orámování je nastavena na černou.



Obrázek 6.2 Kreslíme atomy do canvasu

Výsledek je vizuálně hodnotnější než prostá tabulka z předchozích kapitol. Zbývá veřejná metoda `Draw.getPosition`, která slouží pro získání informace o souřadnicích ve chvíli klepnutí myší. Zde budeme muset bohužel upravit signaturu, protože předávat aktuální HTML prvek nedává smysl. Ať uživatel klepne kamkoliv na hrací plochu, událost nastane vždy na značce `<canvas>`. Vhodnější tedy bude předávat spíše souřadnice kurzoru myši:

```
/* Převod pozice kurzoru na souřadnice buňky */
Draw.getPosition = function(cursorX, cursorY) {
  var rectangle = this._context.canvas.getBoundingClientRect();

  cursorX -= rectangle.left;
  cursorY -= rectangle.top;

  if (cursorX < 0 || cursorX > rectangle.width) { return null; }
  if (cursorY < 0 || cursorY > rectangle.height) { return null; }

  var cellX = Math.floor(cursorX / this.CELL);
  var cellY = Math.floor(cursorY / this.CELL);
  return [cellX, cellY];
}
```