3

Impreza

Gry takie jak *Dance Dance Revolution*, *Mario Party* i *Rock Band* są proste i dzięki temu łatwo znajdują się dla nich nowi gracze oraz są popularną rozrywką grupową. Gatunek gier w stylu „szybko naciśnij klawisz", „naciśnij klawisz w odpowiednim momencie" ostatnio zyskuje na popularności dzięki temu, że programiści i wydawcy zaczęli dostrzegać nowe grupy, potencjalnie nimi zainteresowane.

W tym rozdziale do budowy gry wykorzystamy silnik *atom.js*, który posłuży nam jako lekkie opakowanie dla logiki gry, oraz poznamy interfejs elementu `canvas` do rysowania grafiki.

Receptura. Tworzenie przykładowej gry przy użyciu silnika atom.js

W tym rozdziale użyjemy lekkiej i niewielkiej biblioteki *atom.js* jako bazy dostarczającej podstawową strukturę gry. Biblioteka ta będzie wykonywała cztery zadania polegające na normalizacji `requestAnimationFrame` we wszystkich przeglądarkach, udostępnianiu warstwy abstrakcji do obsługi zdarzeń klawiatury i myszy, obsłudze funkcji zmiany rozmiaru ekranu wraz ze zmianą rozmiaru okna oraz, co najważniejsze, definiowaniu obiektu o nazwie `Game` zawierającego kilka metod związanych z pętlami, których będziemy używać przy budowie gry.

W folderze *3_impreza/start* znajdują się dwa pliki: *atom.js* i *atom.coffee* zawierające podobne zmienne, ale różniące się pod względem struktury. Jeśli nie znasz tego sposobu programowania, drugi z wymienionych plików zawiera kod w języku CoffeeScript. Dla niektórych (wygląda na to, że zalicza się też do nich autor skryptu *atom.js*) programowanie w tym języku jest łatwiejsze niż w JavaScripcie. Jest tak nie bez powodu, ponieważ język CoffeeScript ma prostszą składnię, a jego kod jest łatwiejszy do czytania dla człowieka. W przypadku biblioteki *atom.js* dodatkową korzyścią jest to, że kod źródłowy CoffeeScript jest o 30 wierszy krótszy niż JavaScript.

Jaka są wady CoffeeScriptu? Niestety przeglądarki nie mają interpretera tego języka i rozpoznają tylko JavaScript. To oznacza, że pliki CoffeeScript muszą być tłumaczone na rozpoznawany przez przeglądarki JavaScript. Zwykle do konwersji używa się specjalnego, instalowanego w komputerze programu, ale jeśli jesteś początkującym użytkownikiem i szukasz jak najprostszych rozwiązań, to możesz skorzystać z konwertera dostępnego na stronie *js2coffee.org*. Kolejną wadą używania CoffeeScriptu jest utrudnione znajdowanie błędów. Przeglądarka wykonuje kod JavaScript i zgłasza błędy dotyczące tego języka, mimo że program napisaliśmy w innym. Pojawiają się narzędzia pozwalające rozwiązać ten problem (mapery kodu i przeglądarkowe interpretery), ale na razie żadne z nich nie przyjęło się jako ogólnie akceptowany standard.

Język CoffeeScript w tej książce jest używany tylko w tym rozdziale. Ale warto znać przynajmniej jego podstawy, ponieważ jest wykorzystywany w niektórych kręgach programistycznych, np. w społecznościach *backbone.js* i Ruby on Rails. Nie wiadomo, czy technologia ta zagości w komputerach na dłużej, czy wkrótce zniknie, ale nie należy zapominać, że jest to tylko warstwa abstrakcji na JavaScripcie. Jeśli więc dobrze znasz język JavaScript, to nie masz powodu do obaw. Niemniej jednak zawsze dobrze jest nauczyć się czegoś nowego. Gdybyś szukał więcej informacji o CoffeeScripcie, to dobrym miejscem na początek jest witryna *js2coffee.org*.

Wiesz już mniej więcej, do czego służy i jak jest napisana biblioteka *atom.js*. Możemy zatem utworzyć plik *index.html* zawierający kod przedstawiony na listingu 3.1.

Listing 3.1. Podstawowy plik atom.js

```
<!DOCTYPE html>
<html>
  <head>
    <title>Grzmotnij kreta</title>
  </head>
  <canvas></canvas>
  <script type="text/javascript" src="atom.js"></script>
  <script type="text/javascript" src="game.js"></script>
</html>
```

W pierwszym wierszu deklarujemy typ dokumentu jako HTML5, stosując deklarację `html`. Później definiujemy tytuł w elemencie `title` wewnątrz elementu head. Tytuł widać na karcie w przeglądarce i na górnej belce przeglądarki. Następnie tworzymy pusty element canvas oraz ładujemy pliki *atom.js* i *game.js*.

UWAGA

W silnikach gier element canvas jest używany na różne sposoby. Czasami nie wpisuje się go bezpośrednio w kodzie HTML, ponieważ jest tworzony przez silnik, a czasami silnik wykorzystuje istniejący element, posługując się nim samym lub jego identyfikatorem. Skrypt Atom wymaga obecności elementu canvas na stronie, a jeśli jest kilka takich elementów, to pobiera pierwszy z nich.

Jak zapewne się domyślasz, kolejną czynnością będzie utworzenie pliku *game.js*. Niezależnie od tego, jakiego silnika gier użyjesz, zawsze warto jest poszukać dwóch rzeczy: przykładowego kodu i dokumentacji. W przypadku biblioteki *atom.js* zasoby są dość skromne zarówno w pierwszym, jak i drugim przypadku, ale to, co można znaleźć, powinno wystarczyć. Na listingu 3.2 znajduje się przykładowy kod z pliku *README.md* biblioteki *atom.js*.

Listing 3.2. Przykład kodu CoffeeScript z pliku README

```
class Game extends atom.Game
  constructor: ->
    super
    atom.input.bind atom.key.LEFT_ARROW, 'left'
  update: (dt) ->
    if atom.input.pressed 'left'
      console.log "player started moving left"
    else if atom.input.down 'left'
      console.log "player still moving left"
  draw: ->
    atom.context.fillStyle = 'black'
    atom.context.fillRect 0, 0, atom.width, atom.height
game = new Game
window.onblur = -> game.stop()
window.onfocus = -> game.run()
game.run()}
```

Jest to kod CoffeeScript, który przed użyciem trzeba przekonwertować. Jednak zanim to zrobimy, zobaczymy, co ten program w ogóle robi. Utworzona jest zmienna o nazwie `Game`, która rozszerza własność `Game` obiektu `atom` zdefiniowanego w pliku *atom.js*. Słowo kluczowe `super` oznacza, że jeśli metoda nie zostanie znaleziona w obiekcie `Game`, to należy jej szukać w obiekcie nadrzędnym. Ponadto w konstruktorze definiuje się potrzebne wiązania klawiszy (w tym przypadku strzałki w lewo).

Metody `update` i `draw` działają w pętli. Pierwsza czeka, aż zostanie naciśnięty klawisz strzałki w lewo, a druga rysuje czarne tło w rozmiarze elementu canvas, co oznacza wypełnienie kolorem całego okna. Następnie zostaje utworzony nowy obiekt o nazwie `game` dziedziczący po obiekcie `Game` (który z kolei rozszerza `atom.Game`). Funkcje zatrzymują i uruchamiają grę w zależności od tego, czy okno jest aktywne. Uruchomienie gry następuje poprzez wywołanie funkcji `run`.

Na listingu 3.3 jest przedstawiony ten sam kod co wcześniej po przekonwertowaniu na JavaScript. Nie zaprzątaj sobie głowy jego zapisywaniem, bo i tak na listingu 3.4 będziemy go przerabiać.

Listing 3.3. Przykładowa gra w JavaScripcie

```
var Game;
var game;
var __hasProp = {}.hasOwnProperty;
var __extends = function(child, parent) {
  for (var key in parent) {
    if (__hasProp.call(parent, key)) child[key] = parent[key];
  }
  function ctor() {
    this.constructor = child;
  }
  ctor.prototype = parent.prototype;
  child.prototype = new ctor();
  child.__super__ = parent.prototype;
  return child;
};
Game = (function(_super) {
  __extends(Game, _super);
  function Game() {
    Game.__super__.constructor.apply(this, arguments);
    atom.input.bind(atom.key.LEFT_ARROW, 'left');
  }
  Game.prototype.update = function(dt) {
    if (atom.input.pressed('left')) {
      return console.log("player started moving left");
    } else if (atom.input.down('left')) {
      return console.log("player still moving left");
    }
  };
  Game.prototype.draw = function() {
    atom.context.fillStyle = 'black';
    return atom.context.fillRect(0, 0, atom.width, atom.height);
  };
  return Game;
})(atom.Game);
game = new Game;
window.onblur = function() {
  return game.stop();
};
window.onfocus = function() {
  return game.run();
};
game.run();
```

Konwersja na JavaScript przy użyciu narzędzia w rodzaju *js2coffee.org* powoduje znaczne zwiększenie objętości kodu. Pod względem składniowym obie wersje w dużym stopniu różnią się między sobą, chociaż większość strasznego nowego kodu znajduje się nad funkcją `uodate`. Klasy i rozszerzenia zaimplementowane w CoffeeScripcie robią się skomplikowane po konwersji kodu na JavaScript.

Przede wszystkim zmienne `game` i `Game` są zdefiniowane na początku zakresu, w którym były zdefiniowane w kodzie CoffeeScript. Jako że nie ma tu funkcji określającej zakres, zmienne te znaj-

się w zakresie globalnym. Może się to wydawać dość dziwnym podejściem, bo przecież prawie dobrze zmienne te można by było zdefiniować bezpośrednio przed nadaniem im wartości lub ich użyciem. Rozwiązanie takie zastosowano, aby nie mylić programistów, którzy są przyzwyczajeni do tradycyjnego sposobu określania dostępności zmiennych w JavaScripcie polegającego na tym, że zakres lokalny wyznaczają wszystkie bloki (kod w klamrach { }), a nie tylko funkcje. Dzięki zastosowanemu podejściu od razu wiadomo, jaki jest zakres dostępności opisywanych zmiennych, oraz nie ma problemu z ustalaniem ich zakresu w instrukcjach `for` i `if`. W tej książce nie trzymam się cały czas ściśle tej konwencji, ale warto o niej wiedzieć.

Z 42 wierszy kodu znajdującego się na listingu 3.3 co najmniej połowa (głównie na początku) dotyczy ogólnej i trochę niejasnej implementacji klas i podklas. W kompilacji CoffeeScriptu cel utworzenia zrozumiałego kodu JavaScript nie jest nawet w przybliżeniu tak ważny jak sama składnia CoffeeScriptu. Na szczęście w JavaScripcie jest dostępna metoda ułatwiająca tworzenie obiektów. Na listingu 3.4 znajduje się kod przerobiony z użyciem metody `Object.create`. Wklej go na początku pliku *game.js*.

Listing 3.4. Użycie metody Object.create do tworzenia obiektów potomnych w pliku game.js

```
atom.input.bind(atom.key.LEFT_ARROW, 'left');

game = Object.create(Game.prototype);

game.update = function(dt) {
  if (atom.input.pressed('left')) {
    return console.log("player started moving left");
  } else if (atom.input.down('left')) {
    return console.log("player still moving left");
  }
};

game.draw = function() {
  atom.context.fillStyle = 'black';
  return atom.context.fillRect(0, 0, atom.width, atom.height);
};

window.onblur = function() {
  return game.stop();
};

window.onfocus = function() {
  return game.run();
};

game.run();
```

Jak widać, ilość kodu została znacznie zmniejszona. Składa się on z mniejszej liczby wierszy oraz jest o wiele bardziej klarowny niż wersja skompilowana z CoffeeScriptu. Najlepsze jest to, że cały skomplikowany kod dotyczący tworzenia obiektu z szablonu został zredukowany do jednego wiersza zawierającego wywołanie metody `Object.create`. Trzeba jednak zwrócić uwagę na pewien ważny fakt dotyczący tej funkcji. Zmienna `Game` udostępniona przez skrypt *atom.js* jest konstruktorem obiektu gry, po którym powinniśmy dziedziczyć. Mimo że można spotkać sytuacje, w których jest inaczej, ogólnie rzecz biorąc, własności `prototype` i `constructor` są prawie swoimi przeciwieństwami. Prototyp obiektu to szablon, z którego ten obiekt został utworzony, natomiast konstruktor to funkcja służąca do tworzenia obiektu na bazie prototypu.

Jeśli teraz uruchomisz grę, zobaczysz, że naciśnięcie klawisza strzałki w lewo powoduje wysłanie informacji do konsoli (Firebug albo *Narzędzia dla programistów* Chrome). W nowoczesnych przeglądarkach metoda `create` działa prawidłowo. Konstruktorem obiektu `game` jest funkcja `Game` służąca do tworzenia gier według prototypu `Game`.

UWAGA

Jeśli chcesz dokładnie zrozumieć, czym jest prototyp i konstruktor, wykonaj w konsoli instrukcję `game.constructor`. Można nawet tworzyć połączenia `.constructor.prototype.con` ↪`structor.prototype` z `game` w nieskończoność i zawsze uzyska się dwa takie same wyniki: funkcję opisującą tworzenie obiektu (konstruktor) i szablon tworzony przy użyciu tego konstruktora (prototyp).

Nie jest to jakiś specjalny przypadek dotyczący tylko obiektu `game`. Podobnie można postępować ze zwykłymi obiektami JavaScript. Wpisz w konsoli `"mójŁańcuch".construc` ↪`tor.prototype` i `var obj = {}; obj.constructor.prototype` (razem z np. liczbami i własnymi obiektami) i zobacz, co się stanie.

Nie przejmuj się, jeśli nie całkiem rozumiesz, o co chodzi z całym tym dziedziczeniem. Najważniejsze jest, aby wiedzieć, że w przypadku obiektu naszej gry metoda `Object.create` jest dobrym zamiennikiem dla kodu dotyczącego dziedziczenia z kompilacji CoffeeScriptu. Trzeba tylko pamiętać, że jest to stosunkowo nowa metoda i nie obsługują jej starsze przeglądarki, dla których może być konieczne dodanie „wypełniacza". Jeśli nie wiesz, czym są wypełniacze, poczytaj o bibliotece Modernizr w dodatku C „Zasoby".

OSTRZEŻENIE

RÓŻNE PRZEGLĄDARKI RÓŻNIE OPISUJĄ TE SAME OBIEKTY. Debugując program w kilku przeglądarkach, możesz odkryć, że ten sam obiekt, np. atrybut `__proto__`, i jego zawartość w konsoli każdej z nich jest opisany inaczej. Jeśli sprawia Ci to problemy, możesz przetestować interfejsy obiektów, wywołując na nich metody. Często okazuje się, że mimo różnic w wewnętrznej reprezentacji oraz jej opisu interfejs jest akceptowalny w różnych przeglądarkach.

Receptura. Rysowanie na kanwie

Obecnie nasza gra zawiera tylko metody do aktualizowania i rysowania oraz odbiera dane z klawiatury, aby wydrukować informacje w konsoli. Do tej pory jeszcze nie opisywałem technik rysowania na elemencie `canvas`, który jest jednym z najważniejszych elementów języka i gier HTML5. W tej recepturze elementu `canvas` użyjemy do narysowania tła gry.

Biblioteka *atom.js* pozwala wygodnie pracować z elementem `canvas` głównie dlatego, że niewiele z nim robi. Inne szkielety używane do tworzenia gier przesłaniają metody rysujące i rdzeń, który stanowi bardzo proste API. Na listingu 3.5 znajduje się kod źródłowy przedstawiający, jak biblioteka *atom.js* obsługuje API `canvas`. Nie trzeba w tym nic zmieniać.

Listing 3.5. Dostęp do elementu canvas przez atom.js

```
atom.canvas = document.getElementsByTagName('canvas')[0];
atom.context = atom.canvas.getContext('2d');
```

Te dwa wiersze kodu stanowią podstawę każdego programu wykorzystującego element `canvas`. Pierwszy z nich znajduje pierwszy element `canvas` w pliku HTML i przypisuje go do zmiennej

canvas. Natomiast drugi definiuje interfejs opisujący go jako dwuwymiarowy kontekst rysunkowy. Istnieje też eksperymentalna opcja kontekstu trójwymiarowego. Znajdujące się dalej wiersze kodu dotyczą wiązania zdarzeń myszy.

UWAGA

Kontekst trójwymiarowy jest znacznie bardziej skomplikowany i opis technik analogicznych do użytych w pracy z kontekstem dwuwymiarowym zajałby całą książkę (albo i dwie). Poza tym gry dwuwymiarowe są bardzo popularne i do ich utworzenia nie trzeba mieć aż tak dużych umiejętności programistycznych i graficznych. Technika dwuwymiarowa pozwala utworzyć wiele ciekawych tytułów we wszystkich opisanych w tej książce gatunkach, a ponadto gry takie jak *Paper Mario*, *Super Smash Brothers* i *Street Fighter* są dowodem, że nawet nowe produkcje mogą osiągnąć duży sukces. Jeśli jednak interesuje Cię technika 3D, to w dodatku C znajdziesz wskazówki, gdzie zacząć szukać informacji na ten temat.

uchwyty do elementu canvas i jego trójwymiarowego kontekstu, możemy rozpocząć rysowanie. Zamień metodę draw w pliku *game.js* na metodę przedstawioną na listingu 3.6.

Listing 3.6. Rysowanie tła

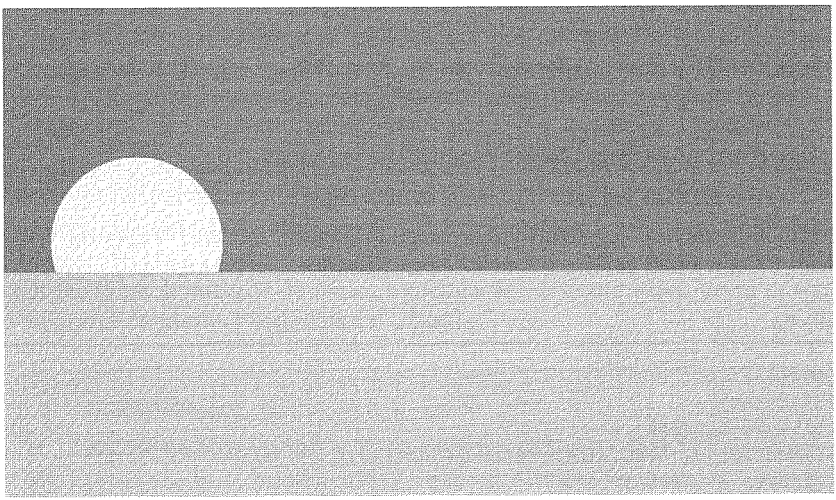
```
game.draw = function() {
  atom.context.beginPath();
  atom.context.fillStyle = '#34e';
  atom.context.fillRect(0, 0, atom.width, atom.height/2);
  atom.context.fillStyle = '#ee3';
  atom.context.arc(140, atom.height/2 -30, 90, Math.PI*2, 0);
  atom.context.fill();
  atom.context.fillStyle = '#2e2';
  atom.context.fillRect(0, atom.height/2, atom.width, atom.height/2);
};
```

początku została użyta funkcja beginPath oznaczająca rozpoczęcie nowej ścieżki. jest wszystko najpierw rysować na ścieżce, aby uniknąć pojawiania się na ekranie pozostałości po poprzednich obrazach. Następnie ustawiamy atrybut fillStyle kontekstu na kolor niebieski. W kolejnym wierszu wypełniamy górną połowę kanwy za pomocą metody fillRect. Parametrami tej metody są współrzędne x i y punktu początkowego, szerokość oraz wysokość. Następnie za pomocą metody fillStyle zmieniamy kolor na żółty, aby przygotować się do narysowania słońca. Przy użyciu metody arc rysujemy koło. W tym celu metodzie tej jako parametry przekazujemy współrzędne x (140 pikseli w prawo) i y (nieco nad środkiem) punktu początkowego, promień koła (90 pikseli), kąt początkowy (360 stopni w radianach to 2·pi) oraz kąt końcowy (0). Aby zdefiniowane koło zostało narysowane, konieczne jest dodatkowe użycie metody fill. Pozostałe dwa wiersze kodu rysują ziemię w sposób podobny do tego, w jaki wcześniej było rysowane niebo. Pamiętaj, że rysowane figury zasłaniają wcześniej narysowane figury. Warto tę zasadę wykorzystać, aby zakryć częściowo słońce ziemią, co pozwala uniknąć konieczności rysowania znacznie bardziej skomplikowanej figury, jaką jest ścięte koło.

Po zmianie metody draw plik *index.html* w przeglądarce internetowej powinien wyglądać tak, jak pokazano na rysunku 3.1.

UWAGA

W wielu bibliotekach i silnikach gier (np. w opisanym w rozdziale 4. „Puzzle" silniku *easel.js*) są zaimplementowane metody pomocnicze ułatwiające rysowanie takich figur jak koło, w przypadku którego wystarczy tylko podać promień, wysokość i szerokość. Ponadto w wielu przypadkach nie trzeba oddzielać kodu opisującego kształt od kodu go rysującego. W dalszych rozdziałach dowiesz się jeszcze więcej na temat wysokopoziomowych technik rysowania.



Rysunek 3.1. Tło gry

Receptura. Rysowanie dziur

Mamy już prostą makietę do budowy naszej gry w polowanie na kreta. Teraz musimy dodać jakieś obiekty, z których ten kret będzie mógł wychodzić. Nazwiemy je dziurami i zaraz je narysujemy. Wcześniej jednak zrobimy trochę porządku w metodzie `draw`, jak pokazano na listingu 3.7.

Listing 3.7. Uproszczenie metody draw

```
game.draw = function() {
  this.drawBackground();
};
game.drawBackground = function(){
  atom.context.beginPath();
  atom.context.fillStyle = '#34e';
  atom.context.fillRect(0, 0, atom.width, atom.height/2);
  atom.context.fillStyle = '#ee3';
  atom.context.arc(140, atom.height/2 -30, 90, Math.PI*2, 0);
  atom.context.fill();
  atom.context.fillStyle = '#2e2';
  atom.context.fillRect(0, atom.height/2, atom.width, atom.height/2);
};
```

Cały dotychczas napisany kod został przeniesiony do metody `game.drawBackground`. Następnie wywołujemy tę metodę w funkcji `draw`. Dla osób słabo znających język JavaScript użycie słowa

kluczowego this w metodzie draw może być niejasne. W tym kontekście this odnosi się do game, a więc instrukcja this.drawBackground() jest równoznaczna z instrukcją game.drawBackground(). Zawsze gdy coś jest niejasne, można spróbować sobie pomóc, wpisując w konsoli instrukcję console.log(toCzegoNieRozumiem). Więcej informacji na temat tego, co robić w trudnej sytuacji, znajduje się w dodatku B „Kontrola jakości".

WSKAZÓWKA

Technika przenoszenia kodu bez zmiany jego funkcjonalności nazywa się **refaktoryzacją** (ang. *refactoring*). Nie da się idealnie zaprojektować programu i dlatego podczas pracy czasami trzeba trochę pozmieniać jego strukturę, aby utrzymać porządek. Istnieje wiele metod refaktoryzacji; jest to w ogóle szeroki temat. Ogólnie rzecz biorąc, należy szukać funkcji o zbyt długich lub bezsensownych nazwach, zbyt dużych plików, zbyt wielu definicji zmiennych oraz nienaturalnych skupisk instrukcji warunkowych. Wszystkie wymienione problemy mogą powodować trudności z utrzymaniem kodu i przy współpracy programistów nad projektem, dlatego warto je w miarę możliwości eliminować.

Na listingu 3.8 znajduje się kod odpowiedzialny za rysowanie dziur. Została dodana metoda o nazwie drawHoles rysująca cztery koła reprezentujące dziury oraz litery wskazujące, którego klawisza należy użyć, aby grzmotnąć kreta. Metodę tę można także wywołać w metodzie draw.

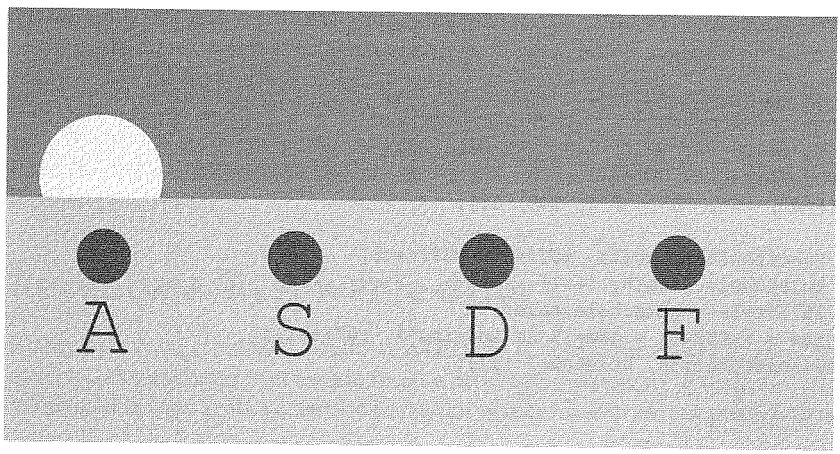
Listing 3.8. Rysowanie dziur

```
game.draw = function() {
  this.drawBackground();
  this.drawHoles(['A', 'S', 'D', 'F'], 145, 85);
};
game.drawHoles = function(holeLabels, xOffset, yOffset){
  for(i = 0; i < holeLabels.length; i++){
    atom.context.fillStyle = game.hole.color;
    var holeLocation = [xOffset + game.hole.spacing*i, yOffset];
    game.hole.draw(holeLocation, holeLabels[i]);
  }
};
game.hole = {
  size: 40,
  spacing: 280,
  color: '#311',
  labelOffset: 140,
  labelColor: '#000',
  labelFont: "130px monospace",
  draw: function(holeLocation, holeLabel){
    atom.context.beginPath();
    atom.context.arc(holeLocation[0], atom.height/2+holeLocation[1], this.size, 0,
    ↪Math.PI*2, false);
    atom.context.fill();
    atom.context.fillStyle = this.labelColor;
    atom.context.font = this.labelFont;
    atom.context.fillText(holeLabel, holeLocation[0] - this.size,
    ↪atom.height/2+holeLocation[1] + this.labelOffset);
  }
};
game.drawBackground = function(){
...
```

Przyjmując, że kod znajdujący się w funkcji będzie wykonywany w każdej klatce pętli gry (jest to jedno z najważniejszych narzędzi biblioteki *atom.js*), pozostaje nam tylko umieścić kod rysujący wewnątrz funkcji `draw`. W tym celu dodaliśmy wywołanie funkcji `drawHoles`, która przyjmuje trzy parametry. Pierwszy parametr to tablica liter do oznaczenia dziur. Natomiast parametry drugi i trzeci określają poziome i pionowe przesunięcie tych dziur. Należy pamiętać, że na kanwie, w odróżnieniu od typowych układów współrzędnych, punkt (0,0) nie znajduje się w lewym dolnym rogu, tylko w lewym górnym.

Funkcja `drawHoles` przegląda za pomocą pętli tablicę etykiet dziur, określa położenie tych dziur oraz rysuje je przy użyciu metody `draw` obiektu `hole`, którego definicja znajduje się dalej. W obiekcie `game.hole` znajdują się definicje własności, łańcuchów, liczb oraz funkcji `draw`. Łańcuchy i liczby są wykorzystywane przez funkcję `draw`, poprzez instrukcje `this.nazwaWłasności`. Większość kodu w tej funkcji powinna być zrozumiała, ponieważ jest podobna do wcześniej omawianego kodu rysującego słońce. Nowe są tylko dwa ostatnie wiersze. Pierwszy ustawia rozmiar i krój pisma za pomocą metody `context.font`, a drugi rysuje tekst przy użyciu metody `context.fillText` wywołanej z trzema parametrami: tekstem do wyświetlenia, współrzędną x oraz współrzędną y.

Teraz plik *index.html* w przeglądarce powinien wyglądać tak jak na rysunku 3.2.



Rysunek 3.2. Tło z dziurami

Receptura. Rysowanie kreta

Dziury i całe środowisko mamy już gotowe, potrzebujemy więc do naszej gry czarnego charakteru (kreta). Jako że na razie tylko testujemy renderowanie rysunku, miejsce jego wyświetlenia nie gra teraz roli. Dodamy tylko do funkcji `draw` wywołanie powodujące pojawienie się kreta w lewym górnym rogu gry (pogrubiony wiersz na listingu 3.9).

Listing 3.9. Rysowanie kreta w głównej funkcji rysującej

```
game.draw = function() {
  this.drawBackground();
  this.drawHoles(['A', 'S', 'D', 'F'], 145, 85);
  this.mole.draw(100, 100);
};
```

Na razie nie ma jeszcze obiektu `game.mole`, więc musimy go utworzyć i zdefiniować w nim metodę `draw`. Kod przedstawiony na listingu 3.10 można umieścić w dowolnym miejscu w pliku *game.js*, pod warunkiem że nie jest to wnętrze funkcji lub obiektu.

Listing 3.10. Obiekt reprezentujący kreta z metodą rysującą

```
game.mole = {
  size: 40,
  color: '#557',
  noseSize: 8,
  noseColor: "#c55",
  eyeSize: 5,
  eyeOffset: 10,
  eyeColor: "#000",
  draw: function(xPosition, yPosition){
    this.drawHead(xPosition, yPosition);
    this.drawEyes(xPosition, yPosition);
    this.drawNose(xPosition, yPosition);
    this.drawWhiskers(xPosition, yPosition);
  },
  drawHead: function(xPosition, yPosition){
    atom.context.beginPath();
    atom.context.fillStyle = this.color;
    atom.context.arc(xPosition, yPosition, this.size, 0, Math.PI*2);
    atom.context.fill();
  },
  drawNose: function(xPosition, yPosition){
    atom.context.beginPath();
    atom.context.fillStyle = this.noseColor;
    atom.context.arc(xPosition, yPosition, this.noseSize, 0, Math.PI*2);
    atom.context.fill();
  },
  drawEyes: function(xPosition, yPosition){
    atom.context.beginPath();
    atom.context.fillStyle = this.eyeColor;
    atom.context.arc(xPosition + this.eyeOffset, yPosition - this.eyeOffset,
    ➥this.eyeSize, 0, Math.PI*2);
    atom.context.fill();
    atom.context.beginPath();
    atom.context.fillStyle = this.eyeColor;
    atom.context.arc(xPosition - this.eyeOffset, yPosition - this.eyeOffset,
    ➥this.eyeSize, 0, Math.PI*2);
    atom.context.fill();
  },
  drawWhiskers: function(xPosition, yPosition){
    atom.context.beginPath();
    atom.context.moveTo(xPosition - 10, yPosition);
    atom.context.lineTo(xPosition - 30, yPosition);
    atom.context.moveTo(xPosition + 10, yPosition);
    atom.context.lineTo(xPosition + 30, yPosition);
    atom.context.moveTo(xPosition - 10, yPosition + 5);
    atom.context.lineTo(xPosition - 30, yPosition + 10);
    atom.context.moveTo(xPosition + 10, yPosition + 5);
    atom.context.lineTo(xPosition + 30, yPosition + 10);
    atom.context.stroke();
  }
};
```

Na początku znajdują się definicje liczbowych i łańcuchowych własności, które będą potrzebne w dalszej części funkcji. Dalej definiujemy funkcję `draw` obiektu `mole`, której wywołanie znajduje się w głównej funkcji rysującej. Funkcja ta przyjmuje jako parametry współrzędne położenia kreta i oddelegowuje zadanie rysowania do różnych funkcji rysujących poszczególne części kreta (głowę, nos, oczy i wąsy). Funkcje rysujące głowę, nos i oczy są już dla Ciebie zrozumiałe, ponieważ doskonale wiesz, jak się rysuje koła. Natomiast w funkcji rysującej wąsy zostały użyte trzy nowe funkcje, które należy znać.

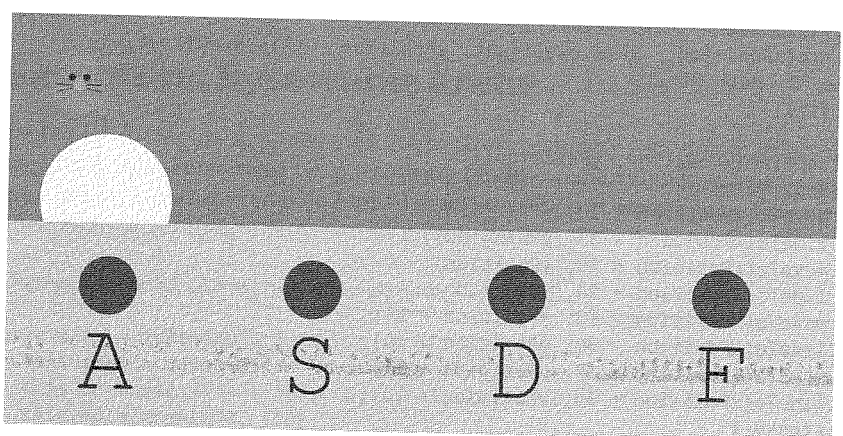
Funkcję `moveTo` można traktować jako rozkaz „przystaw ołówek w tym miejscu", a `lineTo` jako „przeciągnij ołówek z miejsca, w którym się znajduje, do tego miejsca". Tylko że te pociągnięcia ołówkiem są niewidoczne. Dlatego została użyta metoda `stroke` powodująca pojawienie się niewidzialnych kresek.

Lista własności w obiekcie `mole` jest dość długa i można by było podzielić ją na mniejsze części, ale w pozostałej części kodu gry obiekt `mole` jest dla nas pojedynczą jednostką. Podobną sytuację mamy jeszcze w kilku innych miejscach (np. w funkcji `drawWhiskers`), w których można by było trochę skondensować kod. Jeśli masz pewność, że to dobry pomysł, to zrób to, ale uważaj, aby nadmiernie nie skompresować kodu i nie zaciemnić jego znaczenia. Kod, który jest zwięzły i zarazem zrozumiały, to ideał, do jakiego należy dążyć, ale w wielu przypadkach jedynym sposobem na zmniejszenie objętości kodu jest jego skomplikowanie.

UWAGA

Przy użyciu standardowych metod z API `canvas` nawet tak prosta czynność jak narysowanie kreta wymaga dużo wysiłku. Biblioteki zawierają łatwiejsze w użyciu funkcje rysujące, dzięki którym nasz kod może być krótszy. Inną możliwością uniknięcia tak szczegółowego definiowania obiektów jest wykorzystanie obrazów graficznych. Oczywiście to rozwiązanie również jest niepozbawione wad, ale większość używanych do tworzenia gier bibliotek (nie dotyczy to *atom.js*) umożliwia używanie **sprite'ów** ułatwiających wielokrotne wykorzystanie grafik i optymalizację wydajności aplikacji. Sprite'y są często wykorzystywane w dalszej części książki, ale na razie skupiamy się na możliwościach elementu `canvas`.

Po zaimplementowaniu obiektu `mole` i wywołaniu jego metody `draw` plik *index.html* w przeglądarce powinien wyglądać jak na rysunku 3.3.



Rysunek 3.3. Kret narysowany w lewym górnym rogu gry

Receptura. Umieszczanie kretów w dziurach

W tej recepturze znacznie przebudujemy kod dziur. Aktualnie mamy tylko funkcję powodującą rysowanie kilku dziur na ekranie. Jeśli chodzi o pojawianie się kreta, to musimy m.in. zastosować jakiś sposób oznaczania, że kret jest właśnie widoczny. Mamy kilka możliwości do wyboru. Możemy dodać do obiektu `mole` specjalny atrybut i sprawdzać, w której dziurze znajduje się kret. Ale jeśli weźmiemy fizyczną wersję tej gry, to zauważymy, że kret jest po prostu kawałkiem plastiku stanowiącym dekorację dziury. Nie powinno się niepotrzebnie komplikować obiektów w grze i dlatego kreta zaimplementujemy jako dekorację dziury pojawiającą się, gdy ta dziura jest aktywna.

Pierwsza zmiana w kodzie znajduje się na samym dole pliku *game.js*, przed wywołaniem metody `run`. Widać ją na listingu 3.11, na którym została pogrubiona.

Listing 3.11. Wywołanie funkcji makeHoles

```
window.onfocus = function() {
  return game.run();
};

game.makeHoles(['A', 'S', 'D', 'F'], 145, atom.height/2 + 85);
game.run();
```

Nadal rysujemy dziury w każdej iteracji głównej funkcji rysującej, ale obiekty powinny być utworzone tylko raz. Dlatego właśnie ta funkcja jest wywoływana poza funkcjami `draw`, `update` i `run`. Teraz spójrz na listing 3.12, na którym znajduje się kod tej funkcji. Można go umieścić zaraz za funkcją `game.draw`.

Listing 3.12. Definicja funkcji makeHoles

```
game.makeHoles = function(labels, xOffset, yOffset){
  game.holes = [];
  for(var i = 0; i < labels.length; i++){
    var newHole = Object.create(game.hole);
    newHole.holeLocation = [xOffset + game.hole.spacing*i, yOffset];
    newHole.label = labels[i];
    newHole.active = true;
    game.holes.push(newHole);
  };
};
```

W funkcji jest tworzona tablica `game.holes` do przechowywania dziur jako obiektów. Przed dodaniem do tablicy każda dziura ma ustawianych parę własności. Jako że dziury będą rysowane pojedynczo, można usunąć już niepotrzebną funkcję `drawHoles`.

Sporo nowego kodu pojawi się w obiekcie `game.hole`, co widać na listingu 3.13. Obiekt ten nie jest już tylko opakowaniem dla metody `draw`, lecz zawiera ważny kod dotyczący rysowania na ekranie etykiety, dziury oraz kreta.

Listing 3.13. Obiekt hole z dodatkowym kodem dotyczącym rysowania

```
game.hole = {
  size: 40,
  spacing: 280,
```

```
  color: '#311',
  labelOffset: 140,
  labelColor: '#000',
  labelFont: "130px monospace",
  moleOffset: 20,
  draw: function(){
    this.drawHole();
    this.drawLabel();
    if (this.active === true){
      this.drawMole(this.holeLocation[0], this.holeLocation[1] - this.moleOffset);
    };
  },
  drawHole: function(){
    atom.context.fillStyle = this.color;
    atom.context.beginPath();
    atom.context.arc(this.holeLocation[0], this.holeLocation[1], this.size, 0,
    ↪Math.PI*2, false);
    atom.context.fill();
  },
  drawLabel: function(){
    atom.context.fillStyle = this.labelColor;
    atom.context.font = this.labelFont;
    atom.context.fillText(this.label, this.holeLocation[0] - this.size,
    ↪this.holeLocation[1] + this.labelOffset);
  },
  drawMole: function(xPosition, yPosition){
    game.mole.draw(xPosition, yPosition);
  }
};
```

Pierwsza zmiana to dodanie atrybutu `moleOffset`, przy użyciu którego kret jest później umieszczany nieco ponad dziurą. Następnie podobnie jak wcześniej podzieliliśmy zadanie rysowania na kilka etapów, teraz w analogiczny sposób postąpiliśmy z rysowaniem dziury, etykiety i kreta. Kret jest rysowany i „wygląda z dziury" tylko wtedy, gdy dziura jest aktywna (`active`). Kod rysujący kreta zawiera wiele szczegółów, więc pozostawiliśmy go w obiekcie `game.mole` i jego metodzie `draw`, w której nic się nie zmieniło.

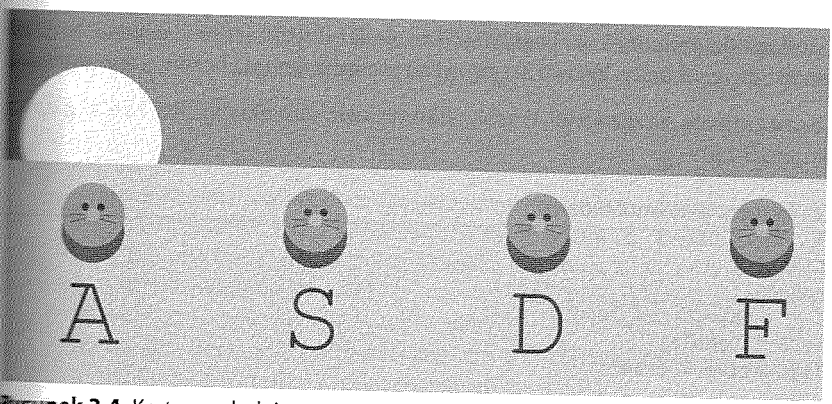
Ostatnia zmiana, jakiej dokonamy w tej recepturze, znajduje się w funkcji `game.draw`. Zaznaczono ją pogrubieniem na listingu 3.14.

Listing 3.14. Nowa metoda draw

```
game.draw = function() {
  this.drawBackground();
  for (var i = 0; i < game.holes.length; i++){
    game.holes[i].draw();
  }
};
```

Tło rysujemy tak jak do tej pory, ale zamiast wywoływać funkcje rysujące dziury i kreta, przeglądamy za pomocą pętli wszystkie dziury i wywołujemy ich metodę `draw`.

Jeśli poprawnie wykonałeś wszystkie instrukcje, to na stronie *index.html* powinny znajdować się cztery aktywne dziury i z każdej z nich powinien wyglądać kret (rysunek 3.4).



Rysunek 3.4. Krety wyglądające ze wszystkich dziur

WSKAZÓWKA

W tej recepturze wprowadziliśmy wiele zmian. Jeśli wystąpią jakieś problemy, dokładnie przejrzyj zmienione fragmenty i zwróć szczególną uwagę na to, jakie parametry są przekazywane do wywoływanych funkcji. Jeśli wszystko inne zawiedzie, możesz porównać zawartość swojego pliku *game.js* z zawartością pliku *impreza/po_recepturze5/game.js*.

Rysunki są już gotowe, dzięki czemu na stronie widać miły obrazek kretów i zachodzącego słońca. Ale to jeszcze nie jest gra. W następnej recepturze trochę zbliżymy się do naszego celu, sprawiając, że krety będą pojawiać się i znikać.

Receptura. Dynamiczne pokazywanie kreta

Normalnie wszystkie dziury są cały czas aktywne. W tej recepturze sprawimy, że dziury będą aktywowane na zmianę co kilka sekund, tak że w każdej chwili aktywna będzie tylko jedna z nich. Zaczniemy od zmodyfikowania funkcji `update` w pliku *game.js* i ustawienia kilku używanych przez nią zmiennych (pogrubiony kod na listingu 3.15).

Listing 3.15. Modyfikacja funkcji update

```
atom.currentMoleTime = 0;
atom.tillNewMole = 2;
game.update = function(dt) {
  atom.currentMoleTime = atom.currentMoleTime + dt;
  if (atom.currentMoleTime > atom.tillNewMole){
    game.activeMole = Math.floor(Math.random()*4);
    atom.currentMoleTime = 0;
  }
}
```

W zmiennej `currentMoleTime` zapisujemy, jak długo określona dziura jest aktywna. W kolejnym wierszu jest zapisana liczba sekund do aktywacji nowej dziury. Jeśli chcesz, aby kret wyglądał z dziury dłużej, zwiększ wartość w tym wierszu. W funkcji `update` został usunięty stary kod wypisujący tekst do konsoli i w końcu wykorzystano parametr `dt`, który przechowuje liczbę sekund w postaci liczby zmiennoprzecinkowej, np. `0.017`) od ostatniego wykonania funkcji `update`.

Wartość ta została dodana do zmiennej `currentMoleTime`. Jeśli od wyświetlenia ostatniego kreta upłynęło więcej czasu niż dwie sekundy, losowo aktywowana jest inna dziura i licznik jest zerowany.

W głównej funkcji `draw` zmienia się niewiele. Trzeba tylko ustawić stan aktywności każdej z dziur, jak pokazano na listingu 3.16.

Listing 3.16. Ustawienie stanu aktywności dziur

```
game.draw = function() {
  this.drawBackground();
  for (var i = 0; i < game.holes.length; i++){
    if (i === game.activeMole){
      game.holes[i].active = true;
    }
    else{
      game.holes[i].active = false;
    };
    game.holes[i].draw();
  }
};
```

Zmienione zostały tylko wiersze dotyczące ustawiania stanu dziur. Jako że własność tę teraz ustawiamy tutaj, nie musimy już ustawiać stanu wszystkich dziur na aktywny, jak to robiliśmy wcześniej. Skoro ustawiamy własność `active` w funkcji `draw`, powinniśmy usunąć poniższy wiersz kodu z funkcji `game.makeHoles`:

```
newHole.active = true;
```

Jeśli teraz otworzysz plik *index.html* w przeglądarce, zobaczysz, że kret na krótko pojawia się w różnych dziurach.

Receptura. Bicie kretów

Teraz krety wymknęły się nam spod kontroli, ale w końcu będziemy mogli wykorzystać etykiety (ASDF) do czegoś więcej niż tylko jako wątpliwą ozdobę. Pierwsza zmiana dotyczy tego, że literału tablicowego `['A', 'S', 'D', 'F']` będziemy używać więcej niż raz. Na listingu 3.17 widać, jakie zmiany należy wprowadzić na początku i końcu pliku.

Listing 3.17. Użycie literału game.keys

```
// Usuń wiersz atom.input.bind(atom.key.LEFT_ARROW, 'left');
game = Object.create(Game.prototype);
game.keys = ['A', 'S', 'D', 'F'];
for (var i = 0; i < game.keys.length; i++){
  atom.input.bind(atom.key[game.keys[i]], game.keys[i]);
};
...
game.makeHoles(game.keys, 145, atom.height/2 + 85);
game.run();
```

W istocie tego usuniętego pierwszego wiersza kodu nie użyliśmy ani razu od momentu zaimportowania pliku gry po raz pierwszy. Wiąże on klawisz strzałki w lewo z nazwą `left`. W pętli są wiązane wszystkie cztery klawisze, dla których później będziemy wykrywać zdarzenia naciśnięcia.

Na końcu pliku zamieniliśmy pierwszy parametr wywołania funkcji `game.makeHoles`, którym był literał tablicowy, na `game.keys`.

UWAGA

Pewnie się zastanawiasz, dlaczego napisałem `game.keys[i]` zamiast `game.keys.i`. Zarówno wersja z nawiasem, jak i z kropką odnosi się do własności obiektu, ale powody użycia każdej z nich mogą być różne. Składnia z kropką jest zwięzła i przejrzysta; uważa się, że jeśli nie ma przeciwwskazań, to należy jej używać. Natomiast składnia z nawiasem kwadratowym jest wykorzystywana w sytuacjach, w których użycie składni z kropką mogłoby spowodować powstanie błędu, np. `["mojaWlasnosc" + "1"]`, lub, jak w tym przypadku, gdy zamiast nazwy własności trzeba użyć nazwy zmiennej.

Teraz utworzymy obiekt `bop`, w którym zapiszemy logikę uderzania kretów. Kod przedstawiony na listingu 3.18 można umieścić za funkcją `update`.

Listing 3.18. Obiekt game.bop

```
game.bop = {
  bopped: true,
  total:0,
  draw: function(){
    atom.context.fillStyle = '#000';
    atom.context.font = '130px monospace';
    atom.context.fillText('Wynik: ' + this.total, 300, 200);
  },
  with_key: function(key){
    if (!!(game.activeMole + 1) === true && key ===
    ➥game.holes[game.activeMole].label){
      this.total = this.total+1;
      game.activeMole = -1;
      this.bopped = true;
    }
    else{
      this.total = this.total-1;
    }
  }
}
```

Kod ten można umieścić bezpośrednio przed funkcją `game.draw`. Jako że kod, który napiszemy później, będzie zmniejszał wynik podczas rysowania kreta, jeśli własność `bopped` będzie miała wartość `false`, ustawiliśmy ją na `true`, aby wartość zmiennej `total` nie została zmniejszona już przy pierwszym rysowaniu kreta. Następnie ustawiamy zmienną `total` na zero. Dalej znajduje się definicja funkcji `draw` rysującej wynik na ekranie. Funkcja `with_key` jest wywoływana pośrednio, gdy zostanie naciśnięty jeden z zarejestrowanych klawiszy. Jeśli naciśnięty klawisz zgadza się z etykietą aktywnego kreta (`activeMole` — druga część konstrukcji warunkowej), następuje zwiększenie wartości zmiennej `total`, ustawienie `activeMole` na `-1` oraz `bopped` na `true`. W przypadku gdy gracz naciśnie niewłaściwy klawisz, następuje zmniejszenie wartości zmiennej `total`. Jak działa algorytm sprawdzający, czy kret został trafiony, możesz zobaczyć w metodzie `game.update` pokazanej na listingu 3.19.

UWAGA

Pierwsza część warunku w funkcji `with_key` jest trochę niejasna. Ogólnie jej zadaniem jest uniemożliwić wykonanie drugiej części i spowodowanie błędu, gdy `game.activeMole` jest `undefined`. Technikę tę nazywa się **strażnikiem** (ang. *guard*). Zamiast tego można by było ustawić gdzieś wartość domyślną, ale jest proste rozwiązanie tego problemu. Wymaga ono jednak dodatkowych objaśnień.

W języku JavaScript 0 w warunkach, np. `if (0)`, oznacza `false`. Zwróć uwagę, że wyrażenie `!!(wartość)` sprawdza, czy obiekt ma wartość prawdziwą, czy fałszywą. W związku z tym, jako że `!!(undefined)` i `!!(0)` oznaczają `false`, zakres możliwych wartości zmiennej `game.activeMole` (0, 1, 2, 3) może sprawiać problemy podczas sprawdzania, czy coś do niej przypisano. Gdy `activeMole` odnosi się do pierwszej dziury o indeksie 0, to wynikiem warunku powinno być `true`. Ponadto trzeba przewidzieć sytuację, w której nie ma aktywnego kreta, ponieważ został trafiony i zniknął. Można by było ręcznie ustawić wartość na niezdefiniowaną, ale to powodowałoby później zamieszanie. Dlatego dodaliśmy do sprawdzanej wartości 1. Dzięki temu zakres możliwych wartości zmienia się na 1 – 4, z których każda w warunku oznacza `true`. Wartość -1 (reprezentująca stan trafionego kreta) zamieni się na 0 i będzie oznaczała `false`. Gdy `activeMole` jest `undefined`, ma wartość `NaN` (ang. *not a number*). Gdy wartość tę zwiększy się o jeden, będzie oznaczać `false`, czyli to, o co nam w tym przypadku chodzi.

To było zwięzłe wprowadzenie do technik sprawdzania, czy coś istnieje, czy ma wartość `null` albo `true` lub `false` w języku JavaScript. Istnieje wiele nietypowych przypadków i ogólnie ten język programowania czasami bywa mało przyjazny w tych kwestiach.

Listing 3.19. Sprawdzanie, czy kret został trafiony

```
game.update = function(dt) {
  atom.currentMoleTime = atom.currentMoleTime + dt;
  if (atom.currentMoleTime > atom.tillNewMole){
    game.activeMole = Math.floor(Math.random()*4);
    atom.currentMoleTime = 0;
    if(game.bop.bopped === false){
      game.bop.total = game.bop.total-1;
    }
    else{
      game.bop.bopped = false;
    }
  };
  for (var i = 0; i < game.keys.length; i++){
    if (atom.input.pressed(game.keys[i])){
      game.bop.with_key(game.keys[i]);
    }
  };
};
```

W pierwszej części pogrubionego kodu widać, co się dzieje, gdy gracz nie uderzy kreta, zanim nadejdzie czas na wyświetlenie kolejnego — następuje zmniejszenie liczby punktów. Instrukcja `else` dotyczy sytuacji, gdy kret zostanie trafiony. Wówczas zmienna `bopped` zostaje ustawiona

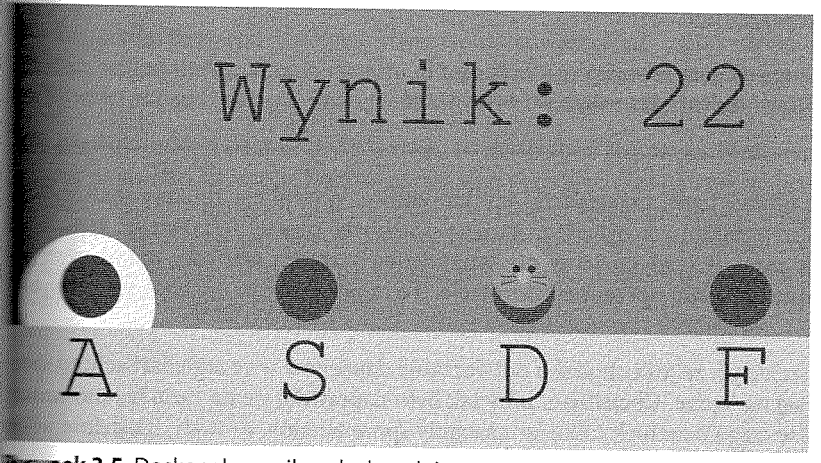
z powrotem na `false` i przygotowana na pojawienie się następnego kreta. Pętla `for` przegląda zarejestrowane klawisze i wywołuje metodę `game.bop.with_key`, gdy któryś z nich zostanie naciśnięty.

Ostatnia zmiana, jakiej należy dokonać, dotyczy rysowania sumy uderzeń w głównej funkcji rysującej (listing 3.20).

Listing 3.20. Zmieniona funkcja draw

```
game.draw = function() {
  this.drawBackground();
  for (var i = 0; i < game.holes.length; i++){
    if (i === game.activeMole){
      game.holes[i].active = true;
    }
    else{
      game.holes[i].active = false;
    };
    game.holes[i].draw();
  }
  this.bop.draw();
```

Wspaniale! Gra jest już ukończona. Jeśli prawidłowo wykonałeś wszystkie opisane czynności i zagrałeś w grę, na ekranie powinieneś mieć widok podobny do pokazanego na rysunku 3.5.



Rysunek 3.5. Doskonały wynik w doskonałej grze

Prawie skończyliśmy. Zanim jednak przejdziesz do następnego rozdziału, pomyśl, jak można tę imprezową grę zamienić w rytmiczną.

Pogrążanie się w rozpaczy z powodu elementu <audio> HTML5

Chciałbym powiedzieć, że w HTML5 i JavaScripcie odtwarzanie dźwięków jest bardzo łatwe. Dodam, że dzięki czemuś takiemu jak `audiocontext.play(noteOrFrequency)` odtworzenie dźwięku jest równie łatwe jak wyświetlenie obrazu. Pragnę również podkreślić, że zarówno starsze,

jak i nowe przeglądarki, a nawet aplikacje mobilne całkiem dobrze radzą sobie z odtwarzaniem dźwięku. Informuję, że w internecie bardzo łatwo można sprawić, by z naszych głośników wydobywał się taki dźwięk, jaki chcemy, i niezależnie od tego, czy komponujesz utwory, miksujesz, czy grasz na jakimś instrumencie, będziesz się świetnie bawić, badając całkiem nowe możliwości udostępniania muzyki. Myślę, że z wiedzą zdobytą w tym rozdziale mógłbyś utworzyć grę podobną do *Rock Band*. Wystarczy dodać do gry informacje o licencji, trochę więcej grafiki, więcej poziomów oraz bardziej precyzyjny system punktacji.

Niestety wszystko, co napisałem w powyższym akapicie, jest nieprawdą. Każdy producent przeglądarek ma własne zdanie na temat tego, jakie kodeki powinny być obsługiwane, przy czym w większości przypadków kwestia dotyczy tego, czy używać kodeków o otwartym, czy zamkniętym kodzie źródłowym. W przypadku plików audio oznacza to, że potrzebne są co najmniej dwie wersje każdego pliku: w formacie OGG i MP3. Ponadto nie ma zgody w kwestii niskopoziomowego tworzenia (nie odtwarzania) dźwięków. Mozilla ostatnio koncentruje swoją uwagę na tzw. Audio API Extension, które jest obecnie wycofywane. Chrome obsługuje standard o nazwie Web Audio rozwijany przez W3C (*http://www.w3.org/TR/webaudio/*). Twórcy Firefoksa ogłosili, że również planują zaimplementować tę technologię, ale jak na razie nie ma ani jednego API obsługiwanego przez obie te przeglądarki.

Są jeszcze inne problemy. Próba wczytania kilku plików dźwiękowych w urządzeniu mobilnym może spowodować awarię gry. Co gorsza, jeśli oczekujesz, że będziesz mógł łatwo sprawdzić, co określona przeglądarka obsługuje, to również spotka Cię zawód. Spójrz na listing 3.21.

Listing 3.21. Sprawdzanie możliwości odtworzenia dźwięków w przeglądarce

```
<audio id="myAudio"></audio>
<script>
  var myAudio = document.getElementById("myAudio");
  myAudio.canPlayType('audio/ogg; codecs="vorbis"');
</script>
```

Wykrywanie obsługi funkcji przez przeglądarki jest kłopotliwe. Wywołując funkcję `canPlayType` na elemencie medialnym takim jak `myAudio`, można oczekiwać prostej odpowiedzi typu `true` lub `false`. A jednak zamiast tego możemy otrzymać jedną z trzech wartości brzmiących jak burknięcia na odczepnego, a nie uprzejme informacje. Te odpowiedzi to `"probably"`, `"maybe"` oraz `""`.

Na koniec pragnę zaznaczyć, że jeśli chodzi o odtwarzanie dźwięków w przeglądarkach, to sytuacja powoli zmienia się na lepsze. Chciałbym, żeby tak było naprawdę, ale na razie trudno szukać pozytywów. Chociaż w ciągu ostatnich kilku lat nastąpiły spore zmiany, to w tej dziedzinie wciąż panuje Dziki Zachód.

Gdybyś z opisanej w tym rozdziale gry chciał utworzyć produkcję w stylu *Rock Band*, to w kwestii udźwiękowienia masz dwie możliwości do wyboru. Możesz generować dźwięki w locie przy użyciu niskopoziomowego API Chrome albo Firefoksa lub użyć gotowych dźwięków. Jeśli zdecydujesz się na drugą z wymienionych opcji, musisz wytworzyć dźwięki (albo użyć cudzych utworów, wykupując licencję), oznaczyć wszystkie fragmenty muzyki oraz dodać uchwyty do tych fragmentów, mapując je na przyciski i synchronizując w czasie. Opis realizacji któregokolwiek z tych pomysłów zająłby bardzo dużo miejsca i dlatego nie przedstawiam go w tej książce.

UWAGA

Nie martw się! Powyżej opisałem tylko niskopoziomowe problemy dotyczące odtwarzania dźwięków w przeglądarkach, ale jest nadzieja. Poszukaj informacji w źródłach wymienionych w dodatku C. Są ludzie, którzy potrafią robić niesamowite rzeczy. Mimo to problemy dotyczące zgodności międzyprzeglądarkowej są bardzo trudne do rozwiązania i obsługa audio HTML5 w przeglądarkach jest na poziomie o wiele niższym niż obsługa grafiki.

Przy użyciu odpowiednich narzędzi powinno Ci się udać załadować muzykę w tle i efekty dźwiękowe. Należy jednak podkreślić, że często jedynym wyjściem jest zaimplementowanie rozwiązania awaryjnego we Flashu, na wypadek gdyby wszystkie inne metody zawiodły. To jednak nie rozwiązuje problemu dynamicznego generowania dźwięku.

Podsumowanie

W tym rozdziale przedstawiłem dużo materiału. Dowiedziałeś się, jak się pracuje z minimalistycznym silnikiem gier *atom.js*, nauczyłeś się podstaw obsługi API `canvas` bez korzystania z udogodnień dostępnych w bardziej rozbudowanych bibliotekach, liznąłeś języka CoffeeScript oraz poznałeś niektóre z trudniejszych do zrozumienia zagadnień dotyczących modelu obiektowego JavaScriptu. Wiesz już, jak posługiwać się słowami kluczowymi `prototype` i `constructor` oraz jak uniknąć związanego z nimi zamętu, używając funkcji `Object.create`. Na zakończenie dowiedziałeś się, jak aktualnie wygląda kwestia odtwarzania dźwięków w przeglądarkach internetowych.

Nie licząc zdarzeń myszy, poznałeś wszystkie najważniejsze części silnika *atom.js*. Aby dowiedzieć się więcej na temat tego skryptu, trzeba by było lepiej poznać CoffeeScript albo JavaScript, tak by móc dodawać własne funkcje lub utworzyć jeszcze mniejszy silnik gier.

Jeśli chodzi o udoskonalanie gry z kretem, to możliwości jest wiele. Można utrudnić grę, sprawiając, aby pojawiało się więcej kretów naraz, albo skracając czas na reakcję po pojawieniu się kreta. Można dodać listę najlepszych wyników i zdefiniować zakończenie gry, w którym zostaje wyświetlony przycisk *Zagraj jeszcze raz*. Można zmienić sterowanie klawiaturą na mysz. Można nawet za pomocą CSS zmienić kursor na obrazek młotka albo dżdżownicy (pożywienie kretów), jeśli nie chcesz, aby kret był źle traktowany.