Jacob Seidelin

HTML5

TWORZENIE GIER

Już dziś zacznij tworzyć gry w HTML5!

Helion

Tytuł oryginału: HTML5 Games: Creating Fun with HTML5, CSS3, and WebGL

Tłumaczenie: Maciej Reszotnik

Projekt okładki: Jan Paluch
Materiały graficzne na okładce zostały wykorzystane za zgodą Shutterstock Images LLC.

ISBN: 978-83-246-4480-3

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: http://helion.pl (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/htm5tg
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem: ftp://ftp.helion.pl/przyklady/htm5tg.zip

Printed in Poland.

Spis treści

ROZDZIAŁ 4

Konstruowanie gry

W tym rozdziale:

- Tworzenie modułu planszy
- Zapisywanie stanu gry
- Mechanizmy układania klejnotów
- Wdrażanie reguł
- Reagowanie na zmianę położenia klejnotów

W tym rozdziale pokażę, w jaki sposób można zaimplementować reguły i mechanikę, które będą sterować działaniem gry. Przedstawię Ci kod niezbędny do stworzenia planszy, z którą pozostałe elementy programu będą mogły prowadzić interakcję. Zaprezentuję też metodę zachowania stanu gry poprzez odrębny moduł, który pozwoli na modyfikację planszy w pewnych ściśle określonych warunkach. Dodatkowo poprowadzę Cię przez konstrukcję kodu, który w odpowiedzi na dopasowania będzie odpowiednio zmieniał położenie pozostałych klejnotów.

Przedstawię również kilka problemów wynikających z zastosowania przez użytkownika różnych urządzeń sterujących rozgrywką. Gra dla jednej osoby działa w oparciu o lokalny kod uruchamiany po stronie klienta, który opisano w tym rozdziale. Zadbamy jednak także o serwerową implementację tych samych reguł.

Tworzenie modułu planszy

Kluczowa mechanika gry jest oddzielona od elementów wyświetlania i sterowania aplikacją. Moduł planszy, który będę omawiał w tym rozdziale, jest modelem danych stanu gry — a dokładniej: obecnego układu klejnotów. Moduł ten odsłania metody, które mogą zostać użyte przez inne moduły (głównie moduł ekranu gry) do interakcji ze stanem rozgrywki. Plansza ma służyć jako zaplecze ekranu gry, dlatego kod w tym rozdziale nie będzie zawierał żadnych nowych elementów wizualnych.

Jak sama nazwa wskazuje, moduł planszy jest logiczną reprezentacją samej planszy z klejnotami. Udostępnia ona innym modułom tylko dwie funkcje — funkcję zapytań, która służy do uzyskania dostępu do klejnotów, i funkcję umożliwiającą przemieszczanie brylantów. Funkcja przemieszczania zamienia miejscami tylko parę klejnotów, gdyż według zasad klejnoty mogą być przesuwane tylko zgodnie z ustalonymi regułami gry. Przemieszczenie klejnotów ma swoje konsekwencje: w jego wyniku klejnoty mogą zostać usunięte, co spowoduje pojawienie się nowych. To właśnie moduł planszy będzie odpowiedzialny za automatyczne spełnianie wszelkich warunków i różne zachowania. Został on zapisany w pliku *board.js* w folderze *scripts*. Listing 4.1 przedstawia jego pierwszą wersję.

Listing 4.1. Moduł planszy

```
jewel.board = (function() {
    /* funkcje gry */
    return {
        /* odsłonięte metody */
    };
})();
```

Odwołaj się teraz do pliku *board.js* w skrypcie *loader.js*, tak jak pokazano na listingu 4.2.

Listing 4.2. Ładowanie modułu planszy

```
// Ładowanie — etap 2.
if (Modernizr.standalone) {
    Modernizr.load([
    {
        load : [
            "scripts/screen.main-menu.js",
            "scripts/board.js"
        ]
    }
    ]);
}
```

Na tym etapie podstawowy moduł planszy nie daje nam żadnych opcji. Dlatego też nadszedł czas, byśmy go usprawnili i zajęli się jego pierwszą metodą.

Inicjalizowanie stanu gry

Kod planszy definiuje kilka ustawień, w tym: liczbę wierszy i kolumn, typy klejnotów itd. Tego rodzaju opcje najlepiej jest odseparować od samego kodu gry, tak by można było je łatwo zmienić bez potrzeby analizowania całego skryptu linijka po linijce. Listing 4.3 przedstawia nowy obiekt — `settings` — dodany do przestrzeni nazw w skrypcie *loader.js*.

Listing 4.3. Dodawanie obiektu settings

```
var jewel = {
    screens : {},
    settings : {
        rows : 8,
        cols : 8,
        baseScore : 100,
        numJewelTypes : 7
    }
};
```

Parametry `rows` i `cols` definiują rozmiar — odpowiednio — wierszy i kolumn. Plansza w *Brylantowym wojowniku* powstała na planie siatki 8×8, która bezproblemowo mieści się na małym ekranie. Z kolei ustawienie `baseScore` określa liczbę punktów, jaką gracz otrzymuje po usunięciu jednego klejnotu z łańcucha. Wynik ten jest mnożony w przypadku łańcuchów, w których znajdują się więcej niż trzy klejnoty. Ja ustawiłem podstawową liczbę punktów na 100, ale nie ma przeciwwskazań, byś ją zmienił, jeśli zależy Ci na uzyskaniu wyższych lub niższych wyników. Ostatnie nowe ustawienie — `numJewelTypes` — wskazuje liczbę typów klejnotów. Wartość ta odpowiada również liczbie sprite'ów (dwuwymiarowych obrazków) przedstawiających kamienie.

Nowe parametry są udostępniane pozostałej części gry i, co ważniejsze (przynajmniej na razie), modułowi planszy.

Inicjalizowanie planszy

Usprawnianie modułu zaczniemy od opracowania funkcji ustawiającej i inicjującej planszę. Nim właściwa gra się rozpocznie, plansza jest wypełniana losowymi klejnotami. Listing 4.4 prezentuje zawartość funkcji `initialize()` w skrypcie *board.js*.

Listing 4.4. Funkcja inicjalizująca

```
jewel.board = (function() {
    var settings,
        jewels,
        cols,
        rows,
        baseScore,
        numJewelTypes;

    function initialize() {
        settings = jewel.settings;
        numJewelTypes = settings.numJewelTypes,
        baseScore = settings.baseScore,
        cols = settings.cols;
        rows = settings.rows;
        fillBoard();
    }

    function print() {
```

```
        var str = "";
        for (var y = 0; y < rows; y++) {
            for (var x = 0; x < cols; x++) {
                str += getJewel(x, y) + " ";
            }
            str += "\r\n";
        }
        console.log(str);
    }

    return {
        initialize : initialize,
        print : print
    };
})();
```

W listingu 4.4 z pewnością Twoją uwagę przykuły deklaracje zmiennych. Pierwsza zmienna importuje moduł ustawień, którego zawartość będzie nam za chwilę potrzebna. Druga zmienna — `jewels` — jest tablicą tablic, czyli dwuwymiarową tablicą reprezentującą stan planszy. W tablicy każdy klejnot jest reprezentowany przez liczbę całkowitą, która wskazuje jego typ. Wykorzystanie struktury tablicy upraszcza dostęp do konkretnych kamieni, w naturalny sposób tworząc system współrzędnych. Przykładowo poniższy fragment kodu pobiera klejnot z kratki o współrzędnych $x=3$, $y=2$:

```
var type = jewels[3][2];
```

Listing prezentuje również kilka zmiennych, których wartości wywodzą się z modułu `settings`. Opisuję je w następnym rozdziale. Funkcja `print()`, którą stworzyłem z myślą o debugowaniu, zwraca dane z planszy do analizy w konsoli JavaScript. Moduł planszy zainicjalizujesz, wpisując następującą komendę w okno konsoli:

```
jewel.board.initialize()
```

Jeśli zaś będziesz chciał sprawdzić generowane dane, wystarczy, że wprowadzisz następującą instrukcję, która wyświetli je w konsoli:

```
jewel.board.print()
```

Korzystanie z funkcji asynchronicznych

Nim przejdę do objaśniania kolejnej funkcji, wprowadźmy wspólnie małą modyfikację w funkcji `initialize()` — tak na przyszłość. W rozdziale 5. wyłożę Ci, jak użyć pracowników (ang. *web workers*), aby przemieścić wykonywanie kodu do oddzielnego wątku poprzez skonstruowanie modułu udostępniającego te same metody co stworzone w tym rozdziale. Pracownicy komunikują się z resztą aplikacji poprzez asynchroniczny interfejs programowania, a to oznacza, że również wszystkie metody publiczne odkryte przez moduł planszy muszą działać asynchronicznie.

Podobnie gdybyś dodał moduł planszy, który korzystałby z kodu po stronie serwera, odpowiedzialnego za weryfikację i walidację danych, musiałbyś wysyłać asynchroniczne wywołania na serwer przy użyciu technologii Ajax. Każda funkcja, która modyfikuje stan gry, wymagałaby nawiązania dwukierunkowej komunikacji z serwerem, z tym że odpowiedź nie musiałaby być równoznaczna z otrzymaniem wyniku. Innymi słowy, sam fakt, że funkcja otrzymuje wywołanie zwrotne, nie oznacza, że wynik operacji jest gotowy.

Problem opóźnionej odpowiedzi można rozwiązać na kilka sposobów, w tym używając samodzielnie opracowanego systemu przydzielania zdarzeń lub mechanizmu **obiecanych obiektów** (ang. *promise objects*) znanego z bibliotek CommonJS i Node.js. Jednak najprostszym rozwiązaniem jest zapewnienie wywołania zwrotnego w formie argumentu dla właściwej metody, która wywoła funkcję

zwrotną po zakończeniu operacji. Prawdopodobnie znasz już taki wzorzec deklarowania z często używanych funkcji JavaScriptu w rodzaju `window.setTimeout()` lub `addEventListener()` dla elementów DOM. Funkcje te jako parametr przyjmują inne funkcje, które są wywoływane w pewnym momencie w trakcie wykonywania zadania. Listing 4.5 przedstawia zmodyfikowaną funkcję `initialize()` w skrypcie *board.js*.

Listing 4.5. Inicjalizowanie funkcji zwrotnej

```
jewel.board = (function() {
    ...
    function initialize(callback) {
        numJewelTypes = settings.numJewelTypes;
        baseScore = settings.baseScore;
        cols = settings.cols;
        rows = settings.rows;
        fillBoard();
        callback();
    }
    ....
})();
```

Aby zainicjalizować planszę poprzez konsolę JavaScript, użyj komendy:

```
jewel.board.initialize(function(){})
```

Od tej pory wszystkie działania w funkcji `initialize()` będą wykonywane natychmiastowo, tak że rezultat będzie ten sam bez wywoływania funkcji zwrotnej. Różnica polega na tym, że wszelkie dokonane zmiany ułatwią późniejszą integrację modułu pracownika.

Wypełnianie planszy

Funkcja `fillBoard()` z listingu 4.5 generuje siatkę pól według wartości zmiennych `cols` i `rows` i wypełnia ją klejnotami. Listing 4.6 pokazuje jej zawartość po dodaniu do skryptu *board.js*.

Listing 4.6. Wypełnianie planszy klejnotami

```
jewel.board = (function() {
    ...
    function fillBoard() {
        var x, y;
        jewels = [];
        for (x = 0; x < cols; x++) {
            jewels[x] = [];
            for (y = 0; y < rows; y++) {
                jewels[x][y] = randomJewel();
            }
        }
    }
    ...
})();
```

Typ klejnotu jest wybierany przy użyciu funkcji `randomJewel()`, która po prostu zwraca liczbę całkowitą mieszczącą się pomiędzy 0 a `(numJewelTypes – 1)`. Listing 4.7 przedstawia funkcję `randomJewel()`.

Listing 4.7. Tworzenie losowego klejnotu

```
jewel.board = (function() {
    ...
    function randomJewel() {
        return Math.floor(Math.random() * numJewelTypes);
    }
    ...
})();
```

Podstawowy algorytm wypełniania planszy jest już gotowy. Przygotowane przez nas rozwiązanie jest jednak zawodne i nie gwarantuje przydatności otrzymanej planszy. Kamienie są wybierane losowo, więc istnieje szansa, że na planszy pojawią się jeden lub dwa gotowe łańcuchy. W fazie początkowej gra nie powinna ich jednak zawierać, ponieważ gracz mógłby zostać nagrodzony punktami za nic. Aby zagwarantować, że taka sytuacja nigdy nie będzie miała miejsca, funkcja `fillBoard()` musi układać klejnoty tak, żeby nie formowały się w rzędy więcej niż dwóch takich samych kamieni.

Algorytm wypełniania ustawia brylanty, zaczynając od górnego lewego rogu i kończąc na dolnym prawym. Oznacza to, że w trakcie wypełniania względem dowolnego kamienia inne klejnoty pojawiają się u góry i po jego lewej stronie. Łańcuch składa się z trzech identycznych kamieni, a to sprawia, że ułożony właśnie brylant nie może mieć tej samej barwy co dwa u góry lub dwa po jego lewej stronie. Dla niewielkiej planszy, takiej jak w *Brylantowym wojowniku*, takie proste rozwiązanie wystarczy. Listing 4.8 ilustruje zmiany w funkcji `fillBoard()`.

Listing 4.8. Usuwanie początkowych łańcuchów

```
jewel.board = (function() {
    ...
    function fillBoard() {
        var x, y,
            type;
        jewels = [];
        for (x = 0; x < cols; x++) {
            jewels[x] = [];
            for (y = 0; y < rows; y++) {
                type = randomJewel();
                while ((type === getJewel(x-1, y) &&
                        type === getJewel(x-2, y)) ||
                       (type === getJewel(x, y-1) &&
                        type === getJewel(x, y-2))) {
                    type = randomJewel();
                }
                jewels[x][y] = type;
            }
        }
    }
    ...
})();
```

Algorytm zawiera kilka pętli, które wybierają typ klejnotu, tak żeby nie powstał żaden łańcuch. W większości przypadków losowo wybrany klejnot nie stworzy łańcucha, ale gdyby jednak tak się stało, zostaje on zastąpiony innym.

Bez mechanizmu sprawdzania granicy pętli ta z listingu 4.8 spróbowałaby uzyskać dostęp do kamieni spoza planszy, co spowodowałoby błąd. Dlatego też zamiast generować brylanty bezpośrednio, procedura `fillBoard()` korzysta z funkcji pomocniczej `getJewel()`, która zapobiega tego rodzaju uchybieniom. Listing 4.9 prezentuje zawartość tej funkcji.

Listing 4.9. Odczytywanie typu kamienia poprzez podanie jego współrzędnych

```
jewel.board = (function() {
    ...
    function getJewel(x, y) {
        if (x < 0 || x > cols-1 || y < 0 || y > rows-1) {
            return -1;
        } else {
            return jewels[x][y];
        }
    }
    ...
})();
```

Funkcja `getJewel()` zwraca cyfrę `-1`, jeśli którakolwiek ze współrzędnych wykracza poza granice planszy, tj. gdy wartość współrzędnej na osi *x* lub *y* jest mniejsza od zera lub większa od, odpowiednio, `(rows-1)` i `(cols-1)`. Wszystkie prawidłowe typy kamieni mieszczą się w zakresie `[0; numTypes-1]`, a to gwarantuje, że zwracana wartość nigdy nie będzie wskazywać typu klejnotu, który uformowałby łańcuch.

Implementacja zasad

Teraz, gdy wstępna plansza jest gotowa, możemy przejść do mechaniki przemieszczania klejnotów. Nasz moduł zawiera metodę `swap`, która przyjmuje dwa zestawy współrzędnych jako parametry — po to, by zamienić miejscami klejnoty, na które one wskazują. Wszystkie zamiany miejsc, które nie spełniają reguł gry, są uznawane za nieważne. Zaczniemy od wdrażania mechanizmu walidacji ruchów gracza.

Walidacja ruchów gracza

Zamiana klejnotów miejscami jest ważna jedynie, jeśli jeden z nich uformuje łańcuch złożony z trzech lub więcej identycznych kamieni. W celu sprawdzenia poprawności zamian przygotowałem funkcję `checkChain()`, która analizuje, czy w nowym miejscu klejnot stanie się częścią łańcucha. Cała procedura sprawdzania polega na określeniu typu klejnotu na wskazanej pozycji, po czym przyrównaniu go do innych kamieni po lewej i po prawej, wraz z policzeniem, ile kamieni tego samego rodzaju znajdzie się w bezpośrednim sąsiedztwie. Podobne przeszukiwanie odbywa się dla kierunków w górę i w dół. Jeśli suma wszystkich klejnotów w pionie lub poziomie będzie większa niż 2 (lub 3, jeśli w grę w chodzi środkowy klejnot łańcucha), dopasowanie zostanie uznane za ważne. Listing 4.10 przedstawia dokładną treść funkcji `checkChain()` w pliku *board.js*.

Listing 4.10. Sprawdzanie łańcuchów

```
jewel.board = (function() {
    ...
    // Zwraca liczbę w najdłuższym łańcuchu,
    // który zawiera kamień o współrzędnych (x,y).
    function checkChain(x, y) {
        var type = getJewel(x, y),
            left = 0, right = 0,
            down = 0, up = 0;

        // Sprawdza kamienie po prawej.
        while (type === getJewel(x + right + 1, y)) {
            right++;
        }
```

```
        // Sprawdza kamienie po lewej.
        while (type === getJewel(x - left - 1, y)) {
            left++;
        }

        // Sprawdza kamienie u góry.
        while (type === getJewel(x, y + up + 1)) {
            up++;
        }

        // Sprawdza kamienie u dołu.
        while (type === getJewel(x, y - down - 1)) {
            down++;
        }
        return Math.max(left + 1 + right, up + 1 + down);
    }
    ...
})();
```

Zwróć uwagę na to, że funkcja `checkChain()` nie zwraca wartości typu `boolean`, lecz liczbę klejnotów znalezionych w najdłuższym łańcuchu. Te wyniki dają nam trochę więcej danych na temat klejnotów, które można będzie wykorzystać podczas punktowania ruchu. Teraz, gdy już wiemy, jak się wykrywa łańcuchy, określenie, czy ruch jest ważny, będzie względnie łatwe. Pierwszym warunkiem jest to, że oba brylanty muszą znajdować się obok siebie. Tylko sąsiadujące klejnoty mogą zostać zamienione miejscami. Jeśli tak jest, gra powinna pozwolić na dokonanie tymczasowej zamiany. Zgodnie z wcześniejszymi ustaleniami, jeżeli ruch zostanie uznany za ważny, funkcja `checkChain()` zwróci liczbę większą niż 2 dla jednego lub dwóch wymiarów. Wystarczy więc przesunąć kamienie i zwrócić wartość wywołania `checkChain()`. Listing 4.11 pokazuje funkcję `canSwap()` zawartą w skrypcie *board.js*, która wdraża ten mechanizm walidacji.

Listing 4.11. Walidacja ruchu

```
jewel.board = (function() {
    ...
    // Zwraca wartość true, jeśli klejnot (x1,y1) może zostać zamieniony miejscem
    // z (x2,y2), tworząc dopasowanie.
    function canSwap(x1, y1, x2, y2) {
        var type1 = getJewel(x1,y1),
            type2 = getJewel(x2,y2),
            chain;

        if (!isAdjacent(x1, y1, x2, y2)) {
            return false;
        }
        // Tymczasowo zamienia miejscami wybrane kamienie.
        jewels[x1][y1] = type2;
        jewels[x2][y2] = type1;

        chain = (checkChain(x2, y2) > 2
              || checkChain(x1, y1) > 2);

        // Ustawia klejnoty na poprzednie miejsce.
        jewels[x1][y1] = type1;
        jewels[x2][y2] = type2;

        return chain;
```

```
    }
    return {
        canSwap : canSwap,
        ...
    }
})();
```

W listingu 4.8 wprowadzono nową funkcję pomocniczą: `isAdjacent()`. Funkcja zwraca wartość `true`, jeśli dwa podane zestawy współrzędnych wskazują na sąsiednie komórki, i `false`, jeżeli tak nie jest. Ich ustawienie względem siebie jest wyliczane na podstawie tzw. dystansu manhattańskiego. Jeśli dwie komórki przylegają do siebie, suma odległości musi wynosić 1. W listingu 4.12 znajdziesz treść funkcji `isAdjacent()`.

Listing 4.12. Sprawdzanie sąsiedztwa

```
jewel.board = (function() {
    ...
    // Zwraca wartość true, jeśli klejnot (x1,y1) sąsiaduje z kamieniem (x2,y2).
    function isAdjacent(x1, y1, x2, y2) {
        var dx = Math.abs(x1 - x2),
            dy = Math.abs(y1 - y2);
        return (dx + dy === 1);
    }
    ...
}
```

Działanie funkcji `canSwap()` przetestujesz w konsoli JavaScript po zainicjalizowaniu modułu planszy. Wykorzystaj też funkcję `print()` do sprawdzenia danych z planszy i upewnienia się, że kod działa prawidłowo, poprzez wpisywanie w konsoli instrukcji typu `jewel.board.canSwap(4,3,4,2)`.

Wykrywanie łańcuchów

Po dokonaniu zamiany gra przeszuka planszę w poszukiwaniu łańcucha i go usunie. Po wykonaniu ruchu zniknie zaledwie kilka klejnotów. Pojedynczy łańcuch powstaje w wyniku zamiany miejscami dwóch klejnotów. Jeżeli w wyniku ruchu jakieś klejnoty zostaną usunięte, w ich miejsce wejdą kamienie ponad nimi, a u góry planszy pojawią się nowe brylanty. Oznacza to, że należy sprawdzić obecność łańcuchów na planszy raz jeszcze — teraz jednak zadanie to nie będzie wcale takie proste. Jedyną opcją jest przeszukanie całej tablicy. Jeśli wykorzystasz do tego funkcję `checkChain()`, okaże się, że nie jest to takie skomplikowane. Listing 4.13 przedstawia funkcję `getChains()`, która przechodzi w pętli przez wszystkie kamienie w poszukiwaniu łańcuchów.

Listing 4.13. Przeszukiwanie planszy

```
jewel.board = (function() {
    ...
    // Zwraca dwuwymiarową mapę długości łańcuchów.
    function getChains() {
        var x, y,
            chains = [];
        for (x = 0; x < cols; x++) {
            chains[x] = [];
            for (y = 0; y < rows; y++) {
                chains[x][y] = checkChain(x, y);
            }
        }
```

```
        return chains;
    }
    ...
})();
```

Zwrócona przez funkcję `getChains()` zmienna `chains` jest dwuwymiarową mapą planszy. Zamiast danych o typach kamieni mapa ta zawiera informacje o łańcuchach, na które składają się klejnoty. Każda komórka na planszy jest sprawdzana poprzez wywołanie metody `checkChain()`, co skutkuje dopasowaniem odpowiedniej komórki na mapie do zwracanej przez funkcję wartości.

Usuwanie łańcuchów klejnotów

Samo odnalezienie łańcucha nie wystarczy. Gra musi również wykorzystać otrzymaną informację. Dokładniej rzecz ujmując, łańcuchy muszą zostać usunięte, a klejnoty ponad nimi powinny opaść na ich miejsce. Mapa łańcuchów jest przetwarzana w funkcji `check()` widocznej w listingu 4.14.

Listing 4.14. Przetwarzanie łańcuchów

```
jewel.board = (function() {
    ...
    function check() {
        var chains = getChains(),
            hadChains = false, score = 0,
            removed = [], moved = [], gaps = [];

        for (var x = 0; x < cols; x++) {
            gaps[x] = 0;
            for (var y = rows-1; y >= 0; y--) {
                if (chains[x][y] > 2) {
                    hadChains = true;
                    gaps[x]++;
                    removed.push({
                        x : x, y : y,
                        type : getJewel(x, y)
                    });
                } else if (gaps[x] > 0) {
                    moved.push({
                        toX : x, toY : y + gaps[x],
                        fromX : x, fromY : y,
                        type : getJewel(x, y)
                    });
                    jewels[x][y + gaps[x]] = getJewel(x, y);
                }
            }
        }
    }
    ...
})();
```

Funkcja ta usuwa klejnoty z planszy i przesuwa w ich miejsce nowe. Poza zmodyfikowaniem planszy funkcja `check()` zbiera informacje o usuniętych i przesuniętych kamieniach w dwóch tablicach — `removed` i `moved`. Dane te są ważne, gdyż przydają się później np. przy animowaniu zmian na ekranie.

Wykorzystując zagnieżdżone pętle, funkcja `check()` przechodzi przez wszystkie komórki na planszy. Jeśli dana komórka została oznaczona na mapie wartością większą niż 2, informacja o umiejscowieniu i typie klejnotu zostaje zapisana w tablicy `removed` z wykorzystaniem literału obiektu. Kamień,

który opadnie w nowe miejsce, nadpisze dane o pozycji w późniejszym etapie, toteż na razie nie trzeba modyfikować tablicy klejnotów.

Zwróć uwagę na sposób, w jaki pętla sprawdza wiersze: od dołu do góry zamiast z góry na dół. Rozwiązanie to pozwoli Ci natychmiastowo przemieścić inne klejnoty w dół planszy. Algorytm ten zachowuje licznik każdej kolumny wewnątrz tablicy `gaps`. Nim zacznie on przetwarzać kolejną kolumnę, ustawia jej licznik na `0`. Za każdym razem gdy usunięty zostanie klejnot, licznik jest zwiększany o 1. Z kolej jeżeli klejnot pozostaje na swoim miejscu, licznik tablicy `gaps` określi, czy powinien on zostać przesunięty w dół. Stanie się tak, jeżeli licznik ma wartość dodatnią — wtedy klejnot opadnie w dół o równą mu liczbę wierszy. Wartość ta jest zapisana w drugiej tablicy — `moved` — za pomocą podobnego literału obiektu, z tym że tym razem zachowane zostaną w niej pozycja początkowa i końcowa. W tym momencie należy uaktualnić tablicę `jewels`, ponieważ wskazywane przez nią współrzędne uległy zmianie.

Generowanie nowych klejnotów

Funkcja `check()` nie została do końca opracowana; wciąż zawiera kilka niedociągnięć. Przenosząc istniejące klejnoty w dół, wypełniasz co prawda luki, lecz tworzysz nowe w górnej części planszy. Dlatego też po przetworzeniu wszystkich klejnotów w kolumnie należy stworzyć nowe kamienie, które spłyną z górnej granicy w dół. Listing 4.15 przedstawia modyfikację odpowiedzialną za ten mechanizm.

Listing 4.15. Dodawanie nowych klejnotów

```
jewel.board = (function() {
    ...
    function check() {
        ...
        for (var x = 0; x < cols; x++) {
            gaps[x] = 0;
            for (var y = rows-1; y >= 0; y--) {
                ...
            }

            // Dodaje nowe klejnoty u góry planszy.
            for (y = 0; y < gaps[x]; y++) {
                jewels[x][y] = randomJewel();
                moved.push({
                    toX : x, toY : y,
                    fromX : x, fromY : y - gaps[x],
                    type : jewels[x][y]
                });
            }
        }
    }
    ...
})();
```

Liczba nowych kamieni, które trzeba wygenerować w kolumnie, jest równa liczbie wolnych komórek, które się w niej znajdują. Konkretne współrzędne, które mają zająć klejnoty, są łatwe do obliczenia, ponieważ nowe kamienie zawsze spadają na wolne miejsca z góry planszy. Informacje o nowych klejnotach są dodawane do tablicy `moved` poza istniejącymi wcześniej brylantami, które przeniesiono niżej. Z uwagi na fakt, że nowe klejnoty nie mają współrzędnych początkowych, wprowadziłem nieistniejące współrzędne spoza planszy, tak jakby nowe klejnoty istniały wcześniej i czekały na swoją kolej.

Przyznawanie punktów

W funkcji `initialize()` wprowadziłem zmienną `baseScore`, na podstawie której będę obliczał liczbę przyznawanych punktów. Listing 4.16 prezentuje kod odpowiedzialny za punktowanie ruchów gracza, dodany w skrypcie `check()`.

Listing 4.16. Przyznawanie punktów za uformowane łańcuchy

```
jewel.board = (function() {
    ...
    function check() {
        ...
        for (var x = 0; x < cols; x++) {
            gaps[x] = 0;
            for (var y = rows-1; y >= 0; y--) {
                if (chains[x][y] > 2) {
                    hadChains = true;
                    gaps[x]++;
                    removed.push({
                        x : x, y : y,
                        type : getJewel(x, y)
                    });

                    // Dodaje punkty do wyniku.
                    score += baseScore
                           * Math.pow(2, (chains[x][y] - 3));
                } else if (gaps[x] > 0) {
                    ...
                }
                ...
            }
        }
    }
    ...
})();
```

Za każdy składowy klejnot łańcucha gra dodaje punkty do ogólnego wyniku. Liczba otrzymanych punktów zależy od długości łańcucha. Każdy dodatkowy łańcuch podwaja wynik.

> **Pamiętaj:** Zmienna `score` nie zawiera całkowitego wyniku gracza; to jedynie suma punktów zebrana w rezultacie wykonania funkcji `check()`. Moduł planszy nie odzwierciedla reprezentacji idei gracza. Po prostu oblicza, jaką liczbę punktów należy nadać za wykonanie danego ruchu.

Łańcuchy zniknęły, a powstałe luki zostały wypełnione nowymi klejnotami. Trzeba jednak pamiętać, że nowe klejnoty mogą stworzyć nowe łańcuchy — dlatego przed nami jeszcze trochę pracy. Funkcja `check()` będzie wywoływać siebie rekurencyjnie do czasu, gdy nie wykryje żadnych łańcuchów. Jak pamiętamy, musi ona zwracać zachowane zmiany. Listing 4.17 przedstawia wprowadzone w niej zmiany.

Listing 4.17. Rekurencyjna analiza planszy

```
jewel.board = (function() {
    ...
    function check(events) {
        ...
```

```
        events = events || [];
        if (hadChains) {
            events.push({
                type : "remove",
                data : removed
            }, {
                type : "score",
                data : score
            }, {
                type : "move",
                data : moved
            });
            return check(events);
        } else {
            return events;
        }
    }
    ...
}
```

Teraz należy połączyć dane z tablic `removed`, `moved` i `score` z informacjami zwracanymi przez rekurencyjne wywołania. W tym celu dodaj opcjonalny argument zdarzenia w funkcji `check()`. Argument ten jest używany tylko w rekurencyjnych wywołaniach. Jeśli do funkcji nie zostanie przekazany żaden argument, zdarzenia są przypisywane do pustej tablicy. Po wykonaniu analizy planszy dodajemy wynik, po czym plansza zmienia się w tablicę zdarzeń poprzez użycie prostego formatu obiektów zdarzeń, który przedstawiłem w listingu 4.16. Wszystkie obiekty zdarzeń zawierają właściwości `type` i `data`. Jeżeli gra nie znajdzie żadnych łańcuchów, ułożenie kamieni na planszy nie ulegnie zmianie, a funkcja `check()` nie zostanie wywołana ponownie. W tym momencie zostanie zwrócona tablica zdarzeń, która odbierze pierwsze wywołanie i wykona żądaną operację. W ten sposób podmiot, który ją wywoła, uzyska listę wszystkich zmian, które zaszły, nim gracz wykona kolejny ruch.

Ponowne wypełnianie siatki

W trakcie rozgrywki prędzej czy później dojdzie do sytuacji, w której ułożenie kamieni na planszy zablokuje wszystkie ruchy. Gra musi rozpoznać taką sytuację i zainicjować ponowne wypełnienie planszy nowymi klejnotami, by gracz mógł kontynuować zabawę. Przygotujemy więc funkcję, która rozpozna, czy na planszy możliwe są jakiekolwiek ruchy. Listing 4.18 przedstawia treść funkcji `hasMoves()`.

Listing 4.18. Sprawdzanie dostępnych ruchów

```
jewel.board = (function() {
    ...
    // Zwraca wartość true, jeśli znajdzie przynajmniej jeden możliwy ruch.
    function hasMoves() {
        for (var x = 0; x < cols; x++) {
            for (var y = 0; y < rows; y++) {
                if (canJewelMove(x, y)) {
                    return true;
                }
            }
        }
        return false;
    }
    ...
})();
```

Funkcja `hasMoves()` zwraca wartość `true`, jeśli co najmniej jeden klejnot może zostać przeniesiony, tak by stworzył łańcuch; w przeciwnym razie zwraca wartość `false`. W listingu 4.19 znajdziesz pomocniczą metodę `canJewelMove()`, której zadaniem jest sprawdzenie współrzędnych dla ruchów.

Listing 4.19. Sprawdzanie wariantów ruchu dla pojedynczego kamienia

```
jewel.board = (function() {
    ...
    // Zwraca true, jeśli współrzędne (x,y) są odwzorowane na planszy i
    // jeśli kamień w tym punkcie może pozostać zamieniony z sąsiednim.
    canJewelMove(x, y).
    function canJewelMove(x, y) {
        return ((x > 0 && canSwap(x, y, x-1 , y)) ||
                (x < cols-1 && canSwap(x, y, x+1 , y)) ||
                (y > 0 && canSwap(x, y, x , y-1)) ||
                (y < rows-1 && canSwap(x, y, x , y+1)));
    }
    ...
})();
```

Aby sprawdzić, czy klejnot może zostać przeniesiony i stworzyć łańcuch, metoda `canJewelSwap()` używa funkcji `canSwap()` — to właśnie ona określa możliwość zamiany z jednym z czterech sąsiadujących kamieni. Do wywołania metody `canSwap()` dochodzi, jeśli sąsiadujący kamień znajduje się w obrębie planszy. W rezultacie funkcja ta dokonuje zamiany z kamieniem np. po lewej stronie, tylko jeśli jego współrzędna wynosi przynajmniej 1 i jest mniejsza od `(cols-1)`.

Jeżeli gra odkryje, iż nie da się wykonać żadnego ruchu (co nastąpi, gdy funkcja `hasMoves()` zwróci `false`), plansza zostanie automatycznie wypełniona nowymi brylantami. Wyzwalacz wypełniania umieściłem w funkcji `check()`. Po zidentyfikowaniu łańcuchów, usunięciu klejnotów i wygenerowaniu nowych kamieni wywoływana jest funkcja `hasMoves()` i w przypadku niestwierdzenia możliwości ruchu wprowadza nowy zestaw brylantów na planszy. Wszelkie opisane zmiany znajdziesz w listingu 4.20.

Listing 4.20. Wyzwalanie wypełnienia planszy

```
jewel.board = (function() {
    ...
    function check(events) {
        ...
        if (hadChains) {
            ...

            // Wypełnia planszę ponownie, jeśli gracz nie będzie miał żadnego ruchu.
            if (!hasMoves()) {
                fillBoard();
                events.push({
                    type : "refill",
                    data : getBoard()
                });
            }
            return check(events);
        } else {
            return events;
        }
    }
})();
```

Poza wyzwoleniem metody `fillBoard()` w listingu pojawia się zdarzenie wypełniania dodane do tablicy zdarzeń. Zawiera ono kopię planszy stworzoną przez funkcję `getBoard()` z listingu 4.21.

Listing 4.21. Kopiowanie danych z planszy

```
jewel.board = (function() {
    ...

    // Tworzy kopię planszy z klejnotami.
    function getBoard() {
        var copy = [],
            x;
        for (x = 0; x < cols; x++) {
            copy[x] = jewels[x].slice(0);
        }
        return copy;
    }

    return {
        ...
        getBoard : getBoard
    };
})();
```

Proste wywołanie metody `fillBoard()` nie gwarantuje, że na nowej planszy pojawi się możliwość wykonania ruchu. Istnieje szansa, że losowe wybieranie klejnotów uniemożliwi jakiekolwiek działanie gracza. Takie ułożenie planszy powinno wyzwolić kolejne, niewidoczne dla użytkownika wypełnienie. Najlepszym miejscem na umieszczenie tego mechanizmu jest funkcja `fillBoard()`. Pojedyncze wywołanie metody `hasMoves()` określi, czy plansza nadaje się do dalszej gry — jeśli okaże się, że nie, funkcja `fillBoard()` zacznie wywoływać siebie rekurencyjnie do czasu, aż uzyska pożądany wynik. W rezultacie plansza będzie wypełniana raz po raz, do czasu gdy uzyskana zostanie przynajmniej jedna para klejnotów. Listing 4.22 prezentuje mechanizm dodany do funkcji `fillBoard()`.

Listing 4.22. Rekurencyjne wypełnianie planszy klejnotami

```
jewel.board = (function() {
    ...
    function fillBoard() {
        ...

        // Rekurencyjnie wypełnia planszę, jeśli nie ma na niej żadnych ruchów.
        if (!hasMoves()) {
            fillBoard();
        }
    }
    ...
})();
```

W ten sposób mechanizm wypełniania poradzi sobie z hipotetycznym scenariuszem, w którym na wstępnej planszy nie byłoby żadnych ruchów. Jednakże istnieje drobne prawdopodobieństwo, że pierwsza wczytana plansza nie pozwoli graczowi na wykonanie żadnego dopasowania. Wywołanie rekurencyjne rozwiązuje ten problem.

Dopasowywanie brylantów

Przygotowaliśmy wszystkie funkcje zarządzające stanem planszy. Brakuje nam tylko metody zamieniającej klejnoty miejscami. Jej opracowanie nie jest trudne. Dysponujesz już funkcją `canSwap()`, która określa, czy gracz może wykonać danych ruch, oraz metodą `check()`, która zajmuje się wydarzeniami po wykonaniu dopasowania. Listing 4.23 przedstawia treść funkcji `swap()`.

Listing 4.23. Funkcja swap()

```
jewel.board = (function() {
    ...
    // Jeśli istnieje taka możliwość, zamienia miejscami klejnot w komórce (x1,y1)
    // z klejnotem w komórce (x2,y2).
    function swap(x1, y1, x2, y2, callback) {
        var tmp,
            events;

        if (canSwap(x1, y1, x2, y2)) {

            // Zamienia klejnoty miejscami.
            tmp = getJewel(x1, y1);
            jewels[x1][y1] = getJewel(x2, y2);
            jewels[x2][y2] = tmp;

            // Sprawdza planszę i pobiera listę zdarzeń.
            events = check();

            callback(events);
        } else {
            callback(false);
        }
    }

    return {
        ...
        swap : swap
    };
})();
```

Funkcja `swap()` zostanie odsłonięta dla pozostałej części kodu i może wpłynąć na stan planszy, toteż musi ona działać zgodnie z tym samym asynchronicznym mechanizmem co funkcja `initialize()`. Dlatego też poza dwoma zestawami współrzędnych metoda `swap()` przyjmuje jako parametr funkcję zwrotną. W zależności od tego, czy ruch użytkownika się powiedzie, wywołanie zwrotne otrzyma jako parametr albo listę zdarzeń, albo wartość `false` (w przypadku nieuznanego ruchu). Listing 4.24 przedstawia funkcje upublicznione poprzez moduł planszy.

Listing 4.24. Zwracanie metod publicznych

```
jewel.board = (function() {
    ...
    return {
        initialize : initialize,
        swap : swap,
        canSwap : canSwap,
        getBoard : getBoard,
        print : print
    };
})();
```

To wszystko. Dzięki odsłonięciu tych funkcji stan gry może zostać zmodyfikowany jedynie wskutek ustawienia nowej planszy lub wywołania metody `swap()`. Od tego momentu wszystkie zasady gry będą egzekwowane przez funkcję `swap()`, co jest gwarancją integralności planszy. Dodatkowo jedynym punktem wejściowym danych jest funkcja `getBoard()`, która nie pozwala nadpisywać informacji, co zmniejsza ryzyko „złamania" reguł przez resztę kodu.

Jak zwykle działanie metody `swap()` przetestujesz, wywołując ją w konsoli. Aby tego dokonać, wpisz:

```
jewel.board.initialize(function(){})
```

Następnie użyj komendy `jewel.board.print()`, aby znaleźć właściwą współrzędną, i wpisz np.:

```
jewel.board.swap(4,3,4,2, function(e){console.log(e)})
```

Pamiętaj przy tym, że funkcja `swap()` wymaga wywołania zwrotnego. Przygotuj własnoręcznie metodę, która będzie zwracać listę zdarzeń w konsoli.

Podsumowanie

W rozdziale tym wyłożyłem Ci sposób wdrożenia elementarnej mechaniki gry. Przeprowadziłem Cię przez implementację wszystkich jej reguł — w tym dotyczących przemieszczania klejnotów, usuwania łańcuchów i generowania dodatkowych klejnotów. Plansza gry została zrealizowana w obrębie jednego modułu, pozwalając na dostęp do informacji w zaledwie kilku miejscach, co gwarantuje, że wprowadzone modyfikacje będą działały według wyłożonych wcześniej zasad.

Ponadto rozdział ten bierze pod uwagę tryb dla wielu graczy, którego wdrożeniem zajmiemy się w dalszej części książki. Na razie zapewniliśmy, że gra będzie korzystać z kodu lokalnego lub specjalnego skryptu udostępnionego przez serwer. Użycie funkcji zwrotnych w kilku kluczowych metodach pozwoli dwóm opracowanym modułom współdzielić ten sam interfejs, ułatwiając dodanie asynchronicznego, serwerowego modułu planszy w dalszej fazie projektowania gry.

Skorowidz

Ś

T

U

V

W

Możliwości nowej odsłony języka HTML są nieprawdopodobne. Jego potencjał można wykorzystać do tworzenia atrakcyjnych stron WWW, funkcjonalnych aplikacji oraz... gier! Ten język świetnie sprawdza się nawet w takiej roli. WebGL, JavaScript oraz CSS3 pozwolą Ci rozwinąć skrzydła. Już dziś możesz podjąć wyzwanie i wraz z autorem tej wspaniałej książki stworzyć swoją pierwszą grę!

HTML5. Tworzenie gier napisano w przeświadczeniu, że książka pomoże Ci przenieść Twoje umiejętności tworzenia stron w świat projektowania gier komputerowych. Nie ma znaczenia, czy jesteś projektantem witryn, który pragnie spróbować swoich sił w nowej dziedzinie, twórcą aplikacji Flash zainteresowanym nową technologią, czy przyświeca Ci całkowicie inny cel. Książka pokaże Ci, jak użyć dobrze znanych narzędzi, by pokonać przepaść dzielącą tradycyjne witryny od ekscytujących gier. W trakcie lektury poznasz elementy składowe gry oraz charakterystykę urządzeń mobilnych. Zorientujesz się, jakie są kolejne etapy procesu wytwarzania nowej aplikacji. Nauczysz się tworzyć wielowątkowe rozwiązania dzięki technologii Web Workers oraz rysować atrakcyjne elementy graficzne przy użyciu elementu *canvas*. Z kolejnych rozdziałów dowiesz się, jak oprogramować sterowanie ruchem oraz jak przygotować efekty dźwiękowe dla Twojej gry, a potem poznasz tajniki tworzenia grafiki 3D oraz przechowywania danych lokalnie w przeglądarce użytkownika. Książka ta w sposób kompleksowy podchodzi do tematu tworzenia gier w języku HTML5. Jeżeli masz choć trochę zapału, dzięki niej z pewnością osiągniesz cel!

Już wkrótce będziesz mógł:

- zaprojektować swoją wyjątkową grę
- stworzyć zaawansowaną grafikę przy użyciu *canvas* i WebGL
- przechowywać dane lokalnie w bazie danych przeglądarki
- korzystać z wielowątkowości dzięki Web Workers

Od pustej strony do pełnowartościowej gry z HTML5!

helion.pl
księgarnia internetowa

Nr katalogowy: 10417

Księgarnia internetowa
http://helion.pl

Zamówienia telefoniczne:
0 801 339900
0 601 339900

Helion

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

Informatyka w najlepszym wydaniu

Cena 67,00 zł

ISBN 978-83-246-4480-3

9 788324 644803