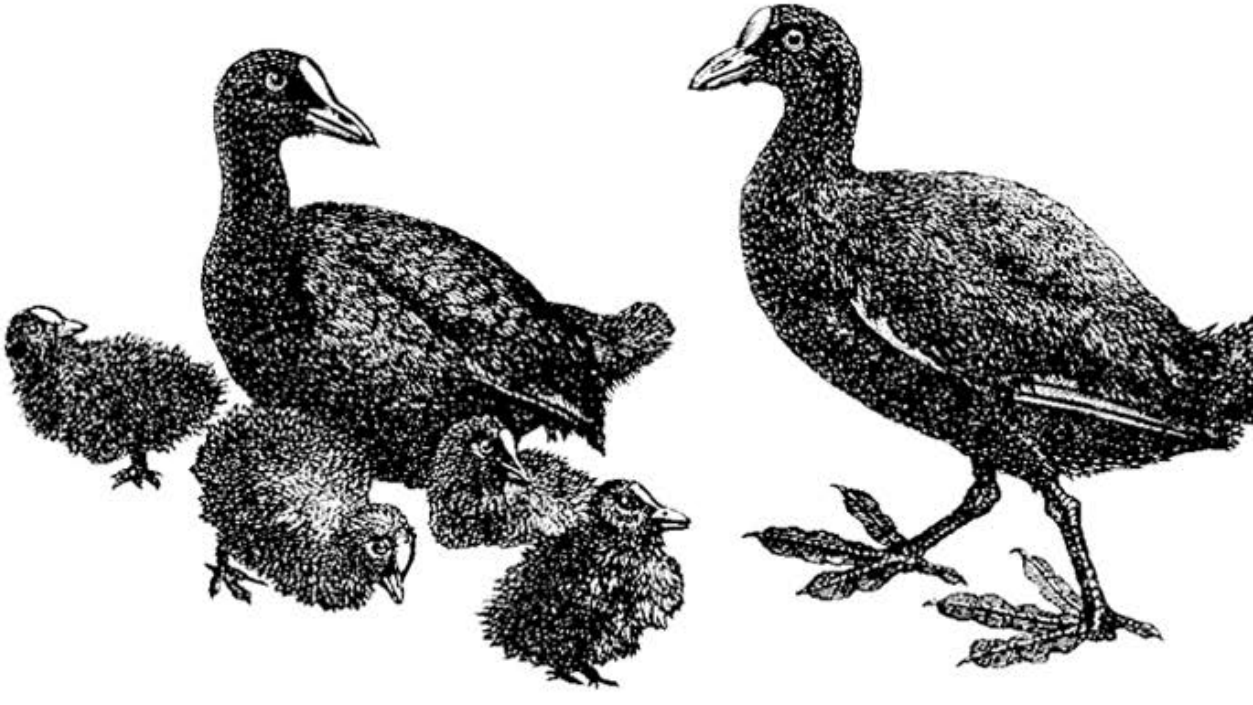
Opublikuj własną grę w serwisie Facebook!

Tworzenie

izometrycznych gier społecznościowych

w HTML5, CSS3 i JavaScript



HELION

O'REILLY®

Mario Andres Pagella

Tytuł oryginału: Making Isometric Social Real-Time Games with HTML5, CSS3, and JavaScript

Tłumaczenie: Aleksander Lamża

ISBN: 978-83-246-3888-8

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

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/twoizo
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

ROZDZIAŁ 5.

Niech świat pozna Twoją grę!

Stworzyłeś atrakcyjną grę z interaktywną grafiką oraz muzyką. Teraz wystarczy pokazać ją światu. Musisz więc umieścić logikę na serwerze, tak by uniemożliwić „grzebanie" w niej i uchronić się przed spowodowanymi tym uszkodzeniami, a następnie połączyć grę z miejscem, gdzie przebywa dużo ludzi, czyli Facebookiem.

Zabezpieczenie przed oszustwami i operacje na serwerze

Jednym z głównych problemów związanych z grami działającymi online jest skuteczna ochrona przed oszustwami. Podobnie jak w przypadku zwykłych witryn internetowych, również tu nie możemy ufać wszystkim użytkownikom, więc zabezpieczenie aplikacji przed szkodliwymi działaniami oraz właściwa obsługa niewłaściwych danych wejściowych i zwracanych wartości powinny mieć najwyższy priorytet.

W grach rozpowszechnianych na zasadach open source ryzyko jest nawet większe, zwłaszcza jeśli zostały utworzone w technologiach takich jak JavaScript i HTML, ponieważ łatwo w nich manipulować zmiennymi (oraz żądaniami `POST` i `GET`), a nawet modyfikować kod w kliencie w czasie działania aplikacji.

Niestety nie ma idealnego sposobu na wszystkie problemy — w każdej grze i aplikacji sprawa wygląda trochę inaczej. Można jednak wyróżnić dwa kluczowe (ale zwykle mało skuteczne) rozwiązania, które stosuje się we wstępnej fazie projektowania oraz później, w trakcie programowania:

- Postaraj się już w projekcie zminimalizować ryzyko popełnienia oszustwa na poziomie klienta (przeglądarki internetowej).
- Sprawdzaj *wszystkie* dane trafiające do serwera.

W przypadku naszej gry, jak w większości społecznościowych gier strategicznych, musimy pamiętać o kilku sprawach:

- Stan konta w grze każdego użytkownika powinien być zapisany w bazie danych (w polu lub osobnej tabeli, w zależności od tego, czy chcemy przechowywać informacje o pojedynczych transakcjach), a każda operacja sprzedaży i kupna powinna aktualizować tę wartość.
- Musimy regularnie synchronizować czas między serwerem i klientem.

- Być może najistotniejsze (i najskuteczniejsze w walce z oszustwami) jest takie zaprojektowanie gry, by w każdej chwili niezawodnie przewidywać punktację użytkownika *na serwerze*, bez konieczności kontaktowania się z klientem.

Jak to osiągnąć? Przyjrzyjmy się poniższemu scenariuszowi:

Początkowy stan posiadania użytkownika wynosi 2000 złotych monet, 0 budynków, czas utworzenia konta równy 1293861600 (co zgodnie z uniksowym czasem odpowiada dacie 1 stycznia 2011 roku i godzinie 00:00:00), a czas ostatniej modyfikacji to 1293861600 (czyli taki sam jak wyżej).

Użytkownik ma do wyboru trzy budynki:

- budkę z lodami za 250 złotych monet, która co 30 minut przynosi 5 złotych monet zysku,
- hotel za 1000 złotych monet przynoszący zysk 30 złotych monet co godzinę,
- kino za 500 złotych monet, które przynosi zysk 12 złotych monet co 30 minut.

Użytkownik zbudował budkę z lodami w chwili 1293861660 (1 stycznia 2011 o godzinie 00:01:00, czyli minutę po utworzeniu konta).

Kolejnym posunięciem było zbudowanie hotelu (1294084800 — 3 stycznia 2011 o godzinie 14:00:00).

W chwili 1294120800 (4 stycznia 2011 o godzinie 00:00:00) użytkownik zbudował kino.

W chwili 1294639200 (10 stycznia 2011 o godzinie 00:00:00) użytkownik powrócił do gry i postanowił sprawdzić stan konta.

Jednym z możliwych rozwiązań mogłoby być zapisywanie informacji o wszystkich budynkach stawianych przez użytkownika i dodanie do stanu konta pola przechowującego dane ostatniej modyfikacji, w którym zapisany byłby też czas operacji przeglądania stanu konta albo kupna bądź sprzedaży budynku. Dzięki temu w sytuacji, gdy użytkownik chce zobaczyć stan konta albo kupił bądź sprzedał jakiś budynek, można wykonać poniższy algorytm:

1. Uaktualnij pole ostatniej modyfikacji bieżącym czasem.
2. Rozpocznij przeglądanie wszystkich budynków użytkownika.
3. Przelicz różnicę (w sekundach) między bieżącym czasem a czasem ostatniej modyfikacji zapisanym w polu użytkownika.
4. Wynik podziel przez liczbę sekund przypadającą na płatności, a wynik dzielenia zaokrąglij w dół.
5. Wynik pomnóż przez liczbę złotych monet otrzymywanych w każdej jednostce czasu.
6. Uaktualnij stan konta, dodając wynik powyższych operacji.
7. W zależności od wykonywanej operacji (wyświetlanie, budowanie lub sprzedawanie) wyświetl stan konta, dodaj zysk z budynku albo odejmij jego wartość.
8. Jeśli budynek był sprzedawany, usuń powiązanie.

Po zastosowaniu tego algorytmu do powyższego scenariusza uzyskamy następujące efekty:

- W chwili 1293861660 użytkownik postanowił zbudować budkę z lodami. Ponieważ nie ma innych budynków, wystarczy, że uaktualnimy konto i czas ostatniej modyfikacji oraz utworzymy powiązanie. Czas ostatniej modyfikacji będzie równy 1293861660.

Bieżący stan konta użytkownika wynosi 1750 (2000 – 250).

- W chwili 1294084800 użytkownik zbudował hotel. Zanim utworzymy nowe powiązanie dla hotelu, musimy przeliczyć stan konta. Wiemy o budowie budki z lodami w chwili 1293861660, więc uaktualniamy pole ostatniej modyfikacji na wartość 1294084800. Aby obliczyć dotychczasowy przychód z budki z lodami, wykonujemy następującą operację:

  ```
  różnicaCzasu = 1294084800   1293861660 = 223140 sekund
  ```

  Budka generuje zysk 5 złotych monet co 30 minut (1800 sekund), więc musimy obliczyć:

  ```
  wynik = różnicaCzasu / 1800 sekund = 123.97
  ```

  czyli tylko 123, ponieważ zaokrąglamy w dół. Następnie wykonujemy operację:

  ```
  wynik = wynik * 5 = 615 złotych monet
  ```

  Uaktualniamy stan konta do wartości 1750 + 615 = 2365, tworzymy powiązanie dla hotelu (czas ostatniej modyfikacji to 1294084800) i pobieramy należność za hotel. Po tych operacjach stan konta użytkownika będzie wynosił 1365 (2365 – 1000).

- W chwili 1294120800 użytkownik kupuje hotel, więc wykonujemy te same operacje co wcześniej (tyle że dla dwóch budynków: budki z lodami i hotelu):

  ```
  różnicaCzasu = 1294120800 – 1294084800 = 36000 sekund
  wynikBudkaZLodami = różnicaCzasu / 1800 = 20
  wynikBudkaZLodami = wynikBudkaZLodami * 5 złotych monet = 100 złotych monet
  wynikHotel = różnicaCzasu / 3600 sekund (jedna godzina) = 10
  wynikHotel = wynikHotel * 30 złotych monet = 300 złotych monet
  stanKonta = stanKonta + wynikHotel + wynikBudkaZLodami = 1765
  stanKonta = stanKonta - 500 złotych monet (opłata za kino)
  ```

  Bieżący stan konta użytkownika wynosi teraz 1265.

- W chwili 1294639200 użytkownik sprawdza stan konta:

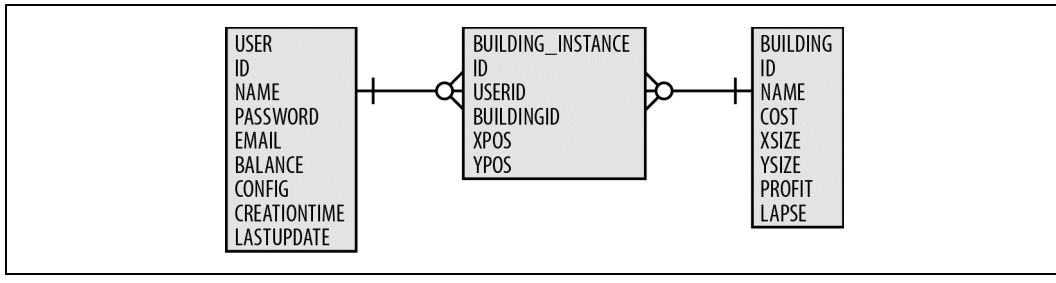
  ```
  różnicaCzasu = 1294639200 - 1294120800 = 518400 sekund
  wynikBudkaZLodami = różnicaCzasu / 1800 = 288
  wynikBudkaZLodami = wynikBudkaZLodami * 5 = 1440 złotych monet
  wynikHotel = różnicaCzasu / 3600 sekund (jedna godzina) = 144
  wynikHotel = wynikHotel * 30 = 4320 złotych monet
  kinoWynik = różnicaCzasu / 1800 = 288
  kinoWynik = kinoWynik * 12 = 3456 złotych monet
  stanKonta = stanKonta + kinoWynik + wynikHotel + wynikBudkaZLodami
  ```

  Oznacza to, że przewidywany stan konta użytkownika 10 stycznia 2011 roku o godzinie 00:00:00 wynosi 10 481.

Chociaż zabezpieczenie kodu aplikacji po stronie klienta przed modyfikacją jest bardzo trudne, o ile wręcz nie niemożliwe, zastosowanie opisanej wyżej metody pomaga ustrzec się przed szkodliwym działaniem użytkowników. Efekt wszelkich zmian jest jedynie lokalny i nie ma wpływu na serwer. Kolejnym przydatnym rozwiązaniem, które możemy zastosować, jest mechanizm używany przez firmę Zynga w niektórych grach — zamiast automatycznie generować zyski, można zmusić użytkownika do ręcznego „zbierania" zysków. Jeśli na przykład budynek generuje 500 monet zysku co 30 minut, a użytkownik nie grał przez trzy dni, gdy w końcu zbierze zysk, otrzyma jedynie 500 monet. Aby zaimplementować ten mechanizm, musimy tylko sprawdzić, czy różnica czasu jest większa niż zysk generowany przez budynek.

Jeśli tak, ustawiamy znacznik na wartość `true`. Po ręcznym zebraniu zysku przez użytkownika znacznik musimy ustawić na `false`.

Zrealizowanie *oryginalnej* metody opisanej wyżej wymaga zastosowania modelu danych przedstawionego na rysunku 5.1.



Rysunek 5.1. Model danych łączący użytkowników z budynkami

Z modelu danych przedstawionego na rysunku 5.1 wynika, że:

- Każdy użytkownik (tabela `USER`) może mieć zero lub wiele egzemplarzy budynków (tabela `BUILDING_INSTANCE`).
- Każdy egzemplarz budynku musi mieć dokładnie jednego użytkownika.
- Każdy budynek (tabela `BUILDING`) może mieć zero lub wiele egzemplarzy.
- Każdy egzemplarz budynku musi być powiązany z tylko jednym budynkiem.

W naszej grze zaimplementujemy ten model danych oraz logikę za niego odpowiedzialną za pomocą bazy danych MySQL i języka PHP. W tym celu na swoim komputerze musisz przygotować odpowiednie środowisko[1], składające się z:

- Bazy danych MySQL (*http://dev.mysql.com/usingmysql/get_started.html*).
- Języka PHP (*http://us.php.net/manual/en/install.php*). Aby można było korzystać z PHP, trzeba jeszcze zainstalować serwer WWW, taki jak Apache, Lighttpd czy nginx. Instrukcję instalacji i konfiguracji możesz znaleźć na stronie z opisem języka PHP.

Po zainstalowaniu i skonfigurowaniu składników środowiska (włącznie z ustaleniem hasła dla użytkownika root) możesz się połączyć z serwerem bazy danych. W tym celu otwórz okno terminala i wykonaj polecenie:

```
mysql –hlocalhost –uroot -phasło
```

Jeśli hasło użytkownika root nie zostało ustalone, należy wykonać polecenie:

```
mysql -hlocalhost -uroot
```

Po połączeniu z bazą danych pojawi się znak zachęty `mysql`:

```
$ mysql -uroot -hlocalhost
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 816
```

[1] Najszybszym i najbardziej niezawodnym sposobem zainstalowania oraz skonfigurowania takiego środowiska jest skorzystanie z gotowych „paczek" zawierających wszystkie niezbędne składniki. Dostępnych jest wiele tego typu rozwiązań, np. wieloplatformowy XAMPP (*http://www.apachefriends.org/en/xampp.html*) — *przyp. tłum.*

```
Server version: 5.1.45 MySQL Community Server (GPL)

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

Na Twoim komputerze może być zainstalowana inna wersja serwera MySQL.

Skoro jesteśmy już połączeni z serwerem, możemy utworzyć bazę danych dla naszej gry. Wykonujemy polecenie:

```
CREATE DATABASE mygame;
```

Jeśli wszystko zadziała prawidłowo, zostanie wyświetlony komunikat:

```
mysql> CREATE DATABASE mygame;
Query OK, 1 row affected (0.00 sec)
```

Przed utworzeniem tabel w bazie danych *mygame* musimy wskazać tę bazę jako aktywną:

```
USE mygame;
```

Jeśli nie pojawią się żadne błędy, zostanie wyświetlony komunikat:

```
mysql> USE mygame;
Database changed
```

W kodzie dołączonym do książki w katalogu *server* znajdziesz dwa pliki SQL: *model-empty.sql* i *model-filled.sql*. Drugi z nich zapisz w wybranym miejscu na komputerze i przejdź z powrotem do okna terminala. Wykonaj poniższe polecenie, podając pełną ścieżkę do zapisanego pliku:

```
source ścieżka_do_pliku_SQL;
```

Jeśli wszystko zadziała prawidłowo, zostanie wyświetlony komunikat:

```
mysql> source ~/Projekty/Gra/server/model-filled.sql
Query OK, 0 rows affected (0.00 sec)

Query OK, 0 rows affected (0.00 sec)

Query OK, 0 rows affected (0.00 sec)

Query OK, 4 rows affected (0.05 sec)

Query OK, 0 rows affected (0.05 sec)

Query OK, 0 rows affected (0.07 sec)

Query OK, 0 rows affected (0.00 sec)

Query OK, 0 rows affected (0.10 sec)

Query OK, 0 rows affected (0.09 sec)

mysql>
```

W bazie danych zostaną utworzone trzy tabele: `users`, `buildings` i `building_instances`. Tabela `buildings` zostanie również wypełniona danymi czterech budynków (z których będziemy korzystać z grze): budki z lodami, hotelu, kina i drzewa.

Aby móc korzystać z bazy *mygame* z poziomu skryptów PHP, musimy utworzyć użytkownika bazy MySQL, nadając mu prawa do wykonywania operacji `SELECT`, `INSERT`, `UPDATE` i `DELETE`. W tym celu wykonujemy polecenia:

```
CREATE USER 'mygameuser'@'localhost' IDENTIFIED BY 'game123';
GRANT SELECT, INSERT, UPDATE, DELETE ON mygame.* TO 'mygameuser'@'localhost';
```

Podobnie jak poprzednio, teraz także nie powinien się pojawić żaden komunikat o błędzie:

```
mysql> CREATE USER 'mygameuser'@'localhost' IDENTIFIED BY 'game123';
Query OK, 0 rows affected (0.09 sec)

mysql> GRANT SELECT, INSERT, UPDATE, DELETE ON mygame.* TO 'mygameuser'@'localhost';
Query OK, 0 rows affected (0.07 sec)

mysql>
```

Więcej informacji na temat korzystania z bazy danych MySQL możesz znaleźć na stronie *http://dev.mysql.com*.

W katalogu *server* w kodzie dołączonym do książki znajdziesz jeszcze kilka innych katalogów i plików:

- *config.php* — zawiera dane wymagane do połączenia z bazą danych oraz rozmiar siatki.
- *classes/class.dbutil.php* — jest to prosta klasa narzędziowa dla bazy MySQL.
- *classes/class.users.php* — odpowiada za obsługę operacji związanych z użytkownikami.
- *classes/class.buildings.php* — odpowiada za obsługę operacji związanych z budynkami.
- *classes/class.operations.php* — odpowiada za tworzenie egzemplarzy budynków i ich pobieranie.
- *classes/class.user.php* — zawiera klasę `user`.
- *classes/class.building.php* — zawiera klasę `building`.
- *classes/class.buildingInstance.php* — zawiera klasę `buildingInstance`.
- *test-database.php* — jest to przydatny skrypt łączący się z bazą danych i testujący możliwości zapisywania, pobierania i usuwania danych.
- *registration.php* — jest to skrypt ilustrujący sposób rejestracji nowych użytkowników z wykorzystaniem wcześniej wymienionych klas.
- *authentication.php* — jest to skrypt ilustrujący sposób uwierzytelniania użytkowników oraz inicjowania sesji z wykorzystaniem wcześniej wymienionych klas.

Ostatnia prosta

W poprzednim podrozdziale omówiliśmy sprawy związane z implementacją skryptów działających po stronie serwera oraz ze strukturą bazy danych. Teraz postaramy się połączyć wstępnie opracowaną grę ze skryptami na serwerze, tak by była możliwa rejestracja użytkowników, ich uwierzytelnianie, a także by wartości wyświetlane w panelu ze stanem posiadania użytkownika odpowiadały rzeczywistym danym zapisanym w bazie danych.

Struktura plików i katalogów całej aplikacji wygląda następująco:

- *index.php* — zawiera główną stronę; obsługuje uwierzytelnianie i rejestrację użytkowników.
- *game.php* — zawiera skrypt gry. Jeśli sesja nie jest aktywna, użytkownik zostanie przekierowany z powrotem na główną stronę w celu rejestracji lub uwierzytelnienia. Kod z tego pliku jest podobny do tego z przykładu 3.1, ale zawiera modyfikacje związane z wyświe-

tlaniem rzeczywistych wartości w panelu; dodatkowo kod JavaScript jest podzielony na kilka plików, aby łatwiej nim zarządzać.

- *async/* — zawiera skrypty PHP odpowiedzialne za obsługę asynchronicznych wywołań generowanych z poziomu kodu JavaScript za pomocą `XMLHttpRequest` (technologia AJAX).
- *css/* — zawiera pliki CSS: *site.css* (wykorzystywany w pliku *index.php*) oraz *ui-style.css* (używany w grze).
- *js/* — zawiera wszystkie pliki JavaScript wykorzystywane w grze.

Skrypt *game.php*, poza przypisywaniem rzeczywistych wartości do wszystkich pól, wykonuje jeszcze dodatkowe operacje wiążące egzemplarze budynków z bieżącym użytkownikiem oraz automatycznie wypełnia macierz `tileMap` budynkami należącymi do użytkownika. Plik *index.php* został uzupełniony o kod umieszczający na siatce drzewa w losowym układzie (10% pól siatki zostaje wypełnionych drzewami).

Za tworzenie egzemplarzy budynków na siatce odpowiada funkcja `initializeGrid()`, która korzysta z tablicy PHP zawierającej egzemplarze budynków oraz macierzy `tileMap` znajdującej się w kodzie JavaScript. Takie rozwiązanie jest dobre, pod warunkiem że użytkownik nie zapełni wszystkich pól siatki budynkami (co jest raczej niemożliwe, biorąc pod uwagę fakt, że siatka ma 62 500 pól). Alternatywnym rozwiązaniem byłoby utworzenie identycznej macierzy `tileMap` w PHP, zakodowanie jej do postaci JSON, a następnie wypełnienie oryginalnej macierzy `tileMap` (w kodzie JavaScript) zdekodowanymi danymi z JSON. Takie rozwiązanie ma jednak pewną wadę — jest mało wydajne, ponieważ dekodowanie dużych obiektów JSON w skrypcie JavaScript mocno obciąża przeglądarkę.

Jeśli w swoim projekcie chcesz wykorzystać większą siatkę, możesz zmodyfikować procedurę ładującą, tak by pobierała w danej chwili tylko wybrany fragment siatki, a pozostałe fragmenty pobierała dynamicznie tylko wtedy, gdy są potrzebne (czyli podczas przewijania). W omawianym tu przykładzie nie zastosowaliśmy takiego rozwiązania, ponieważ urządzenia przenośne charakteryzują się zwykle wysoką przepustowością, ale bardzo dużymi opóźnieniami (zwłaszcza w przypadku połączeń 3G). Wynika stąd, że lepiej pobrać wszystko za jednym razem, a nie wykonywać wiele żądań pobierających małe fragmenty danych.

Zmodyfikujemy również nieco klasę `Game`, aby siatka nie była wyświetlana bezpośrednio po jej utworzeniu. W tym celu trzeba usunąć dwa ostatnie wiersze konstruktora klasy `Game`:

```
this.doResize();
this.draw();
```

Kolejną sprawą, którą się zajmiemy, jest ekran tytułowy z jednego z pierwszych przykładów. Zmodyfikujemy go tak, by wyświetlał się podczas ładowania strony *game.php*. Dodanie tego ekranu wymaga użycia obiektu, który będzie przechowywał informację o bieżącym stanie gry (`LOADING`, `LOADED`, `PLAYING`). Z tego względu musimy utworzyć obiekt `GameState` (dostępny globalnie):

```
var GameState = {
   _current: null,
   LOADING: 0,
   LOADED: 1,
   TITLESCREEN: 2,
   PLAYING: 3
}
```

Dzięki temu, w zależności od bieżącego stanu gry ustawionego w polu `GameState._current`, niektóre obiekty i zdarzenia będą działały w różny sposób. Na przykład kliknięcie tytułowego ekranu, gdy pole `GameState._current` jest ustawione na `GameState.LOADING`, nie powinno wygenerować żadnego zdarzenia. Jeśli jednak stan gry jest ustawiony na `GameState.TITLE-SCREEN`, kliknięcie okna powinno spowodować rozpoczęcie gry.

Ekran tytułowy będzie wyświetlany podczas wstępnego ładowania w tle zasobów gry, takich jak obrazki i dźwięki. W tym celu skorzystamy z obiektu `ResourceLoader`, który będzie odpowiadał za pobranie wszystkich plików przed ich użyciem.

Kod klasy `ResourceLoader` znajduje się w pliku *resourceLoader.js*:

```
// Klasa ResourceLoader

var ResourceType = {
   IMAGE: 0,
   SOUND: 1
}

function ResourceLoader(onPartial, onComplete) {
   this.resources = [];
   this.resourcesLoaded = 0;

   if (onPartial !== undefined && typeof(onPartial) === "function") {
      this.onPartial = onPartial;
   }

   if (onComplete !== undefined && typeof(onComplete) === "function") {
      this.onComplete = onComplete;
   }
}

ResourceLoader.prototype.addResource = function(filePath, fileType, resourceType) {
   var res = {
      filePath: filePath,
      fileType: fileType,
      resourceType: resourceType
   };

   this.resources.push(res);
}

ResourceLoader.prototype.startPreloading = function() {
   for (var i = 0, len = this.resources.length; i < len; i++) {
      switch(this.resources[i].resourceType) {
         case ResourceType.IMAGE:
            var img = new Image();
            var rl = this;

            img.src = this.resources[i].filePath;
            img.addEventListener('load', function() { rl.onResourceLoaded(); }, false);
            break;
         case ResourceType.SOUND:
            var a = new Audio();

            // Pobiera tylko te pliki dźwiękowe, które można odtworzyć.
            if (a.canPlayType(this.resources[i].fileType) === "maybe" ||
               a.canPlayType(this.resources[i].fileType) === "probably") {

               a.src = this.resources[i].filePath;
               a.type = this.resources[i].fileType;
```

```
                var rl = this;
                a.addEventListener('canplaythrough', function() {
                   a.removeEventListener('canplaythrough', arguments.callee, false);
                   rl.onResourceLoaded();
                }, false);
             } else {
                // Nie można odtworzyć dźwięku. Zakładamy, że zasób został pobrany.
                this.onResourceLoaded();
             }

             break;
          }
       }
    }

    ResourceLoader.prototype.onResourceLoaded = function() {
       this.resourcesLoaded++;

       if (this.onPartial != undefined) {
          this.onPartial();
       }

       if (this.resourcesLoaded == this.resources.length) {
          if (this.onComplete != undefined) {
             this.onComplete();
          }
       }

       return;
    }

    ResourceLoader.prototype.isLoadComplete = function() {
       if (this.resources.length == this.resourcesLoaded) {
          return true;
       }

       return false;
    }
```

Z klasy `ResourceLoader` możemy skorzystać w następujący sposób:

```
var rl = new ResourceLoader();

rl.addResource('image1.png', null, ResourceType.IMAGE);
rl.addResource('image2.jpg', null, ResourceType.IMAGE);
rl.addResource('image3.gif', null, ResourceType.IMAGE);
rl.addResource('sound.ogg', 'audio/ogg', ResourceType.SOUND);
rl.addResource('sound.mp3', 'audio/mp3', ResourceType.SOUND);

rl.startPreloading();
```

Po wywołaniu metody `startPreloading()` klasy `ResourceLoader` mamy dostęp do dwóch bardzo użytecznych właściwości:

- *`instancja_klasy_ResourceLoader`*`.resources.length` — zwraca liczbę dodanych zasobów.
- *`instancja_klasy_ResourceLoader`*`.resourcesLoaded` — określa liczbę już pobranych zasobów.

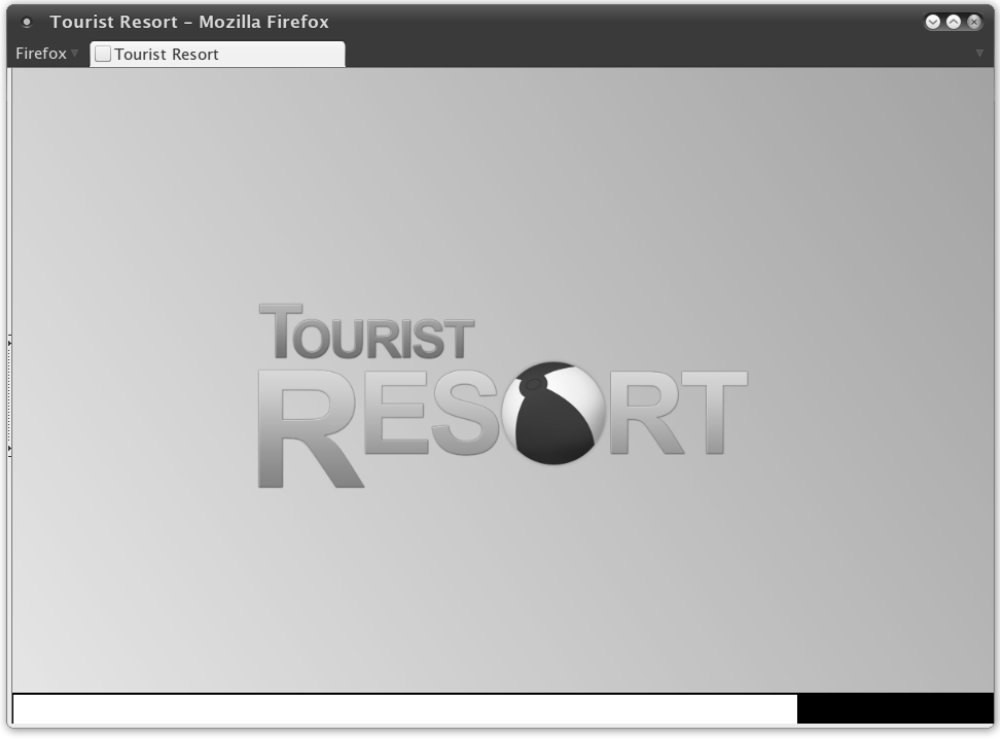
Wykorzystując obie właściwości, możemy obliczyć stan pobierania zasobów wyrażony w procentach:

```
var percentLoaded = Math.floor((instancja_klasy_ResourceLoader.resourcesLoaded * 100) /
instancja_klasy_ResourceLoader.resources.length);
```

Klasa `ResourceLoader` umożliwia również zdefiniowanie dwóch funkcji zwrotnych:

- `onPartial` — funkcja zwrotna jest wywoływana po pobraniu każdego pojedynczego zasobu.
- `onComplete` — funkcja jest wywoływana po pobraniu wszystkich zasobów.

Dzięki tym funkcjom zwrotnym możliwe jest wyświetlenie komunikatów o bieżącym statusie procesu pobierania zasobów (patrz rysunek 5.2).



Rysunek 5.2. Ekran tytułowy z paskiem postępu pobierania

Gra może się znaleźć w różnych stanach (zapisanych w obiekcie `GameState`). W zależności od bieżącego stanu wykonywany jest odpowiedni proces. Za przejścia między stanami gry będzie odpowiadała funkcja `handleGameState()`, która zostanie wywołana na przykład po pobraniu dokumentu. Funkcja ta korzysta z obiektu `GameState` opisanego wcześniej, a jej kod jest następujący:

```
function handleGameState(nextState) {
   if (nextState !== undefined) {
      GameState._current = nextState;
   }

   switch(GameState._current) {
      case GameState.LOADING:
         //...
         break;
```

```
            case GameState.LOADED:
               //...
               break;
            case GameState.TITLESCREEN:
               //...
               break;
            case GameState.PLAYING:
               //...
               break;
            default:
               //...
               break;
         }
      }
```

Aby zmienić bieżący stan gry, wystarczy przekazać funkcji jedną z wartości zdefiniowanych w klasie `GameState`. Możemy też w prosty sposób dodać nowy stan, na przykład `GameState.PAUSED`, który wstrzymuje wykonywanie gry po wciśnięciu jakiegoś klawisza, np. *Esc*. Po ponownym jego wciśnięciu wystarczy wywołać funkcję `handleGameState(GameState.PLAYING)`, aby wznowić działanie gry.

W stanie `GameState.LOADING` wywołamy funkcję `preloadResources()`, która tworzy obiekt klasy `ResourceLoader` odpowiedzialny za wstępne załadowanie wszystkich wymaganych zasobów (obrazków i dźwięków). Kod funkcji wygląda następująco:

```
function preloadResources(canvas, callback) {
   var c = canvas.getContext('2d');

   var rl = new ResourceLoader(printProgressBar, callback);

   rl.addResource('../img/tile.png', null, ResourceType.IMAGE);
   rl.addResource('../img/ui-icons.png', null, ResourceType.IMAGE);
   rl.addResource('../img/spritesheet.png', null, ResourceType.IMAGE);

   rl.addResource('../sounds/title.ogg', 'audio/ogg', ResourceType.SOUND);
   rl.addResource('../sounds/title.mp3', 'audio/mp3', ResourceType.SOUND);

   rl.startPreloading();

   printProgressBar();

   function printProgressBar() {
      var percent = Math.floor((rl.resourcesLoaded * 100) / rl.resources.length);

      var cwidth = Math.floor((percent * (canvas.width - 1)) / 100);

      c.fillStyle = '#000000';
      c.fillRect(0, canvas.height - 30, canvas.width, canvas.height);

      c.fillStyle = '#FFFFFF';
      c.fillRect(1, canvas.height - 28, cwidth, canvas.height - 6);
   }
}
```

Funkcja wyświetla również pasek postępu, który widać na rysunku 5.2.

Zwróć uwagę, że nie pobieramy plików takich jak *cinema.png* czy *tree.png*, ale jeden plik o nazwie *spritesheet.png*. Aby zmniejszyć liczbę wysyłanych żądań (co jest jednym ze skuteczniejszych sposobów optymalizacji aplikacji internetowych), nie pobieramy każdego obrazka z osobna, ale tworzymy z nich arkusz zapisany w jednym pliku, który przekażemy obiektowi `Sprite` opisanemu w jednym z poprzednich rozdziałów.

Obiektowi klasy `ResourceLoader` przekazujemy również parametr `callback`, który jest po prostu wywołaniem funkcji `handleGameState(GameState.LOADED)`. Posłuży ona do przejścia do następnego stanu po zakończeniu pobierania zasobów.

Stan `GameState.LOADED` będzie odpowiedzialny za utworzenie obiektu klasy `Game` oraz wypełnienie siatki budynkami, które posiada dany użytkownik. Po załadowaniu zawartości siatki ponownie zostanie wywołana funkcja `handleGameState()` z parametrem `GameState.TITLESCREEN`, co spowoduje przejście do kolejnego stanu gry.

W stanie `GameState.TITLESCREEN` będzie wyświetlany ekran tytułowy (na który w naszym przypadku składa się logo gry i tekst „Kliknij lub dotknij ekran, aby rozpocząć grę"). W tym samym czasie zostanie dodana funkcja obsługująca zdarzenie kliknięcia (jest ona dezaktywowana po kliknięciu ekranu). Po kliknięciu lub dotknięciu dowolnego obszaru ekranu zostanie wykonana seria wygaszeń obrazu do koloru białego i wyświetlenie tekstu „To Twoje dzieło!". Po odtworzeniu animacji jeszcze raz zostanie wywołana funkcja `handleGameState()`, by zmienić stan na `GameState.PLAYING`, co spowoduje wyświetlenie interfejsu użytkownika gry i siatki ze wszystkimi budynkami oraz aktywowanie funkcji nasłuchujących zdarzeń.

Po zgłoszeniu żądania do pliku *game.php*, który zawiera główny kod gry, serwer automatycznie wypełni panel dostępnymi budynkami wraz z określeniem ich kosztu, generowanych zysków, czasu itd. Wygląd wypełnionego panelu znajduje się na rysunku 5.3.

Rysunek 5.3. Możliwości oferowane przez panel budynków

Teraz, kiedy są już aktywne funkcje nasłuchujące zdarzeń wejściowych (takich jak przewijanie, wciskanie przycisku myszy i klawiszy), musimy skorzystać z obiektu podobnego do `GameState`, który będzie przechowywał informację o wybranym narzędziu, by prawidłowo obsługiwać zdarzenie kliknięcia:

```
var Tools = {
   current: 4, // Domyślne narzędzie
   MOVE: 0,
   ZOOM_IN: 1,
   ZOOM_OUT: 2,
```

```
    DEMOLISH: 3,
    SELECT: 4,
    BUILD: 5
}
```

Dzięki temu po wykryciu kliknięcia w obrębie siatki poniższa funkcja obsługi zdarzenia wykona odpowiednie działania:

```
Game.prototype.handleMouseDown = function(e) {
    e.preventDefault();
    switch (Tools.current) {
        case Tools.BUILD:
            //...
            break;
        // Pozostałe narzędzia...
    }
}
```

Niektóre akcje, jak budowanie czy burzenie, wymagają dostępu do bazy danych, by uaktualnić jej stan na serwerze. Obsługa komunikacji z serwerem jest realizowana za pomocą funkcji zawartych w pliku *comm.js*:

```
var SERVER_PATH_URL = "http://localhost:8080/";

function request(url, callback) {
    var req = false;

    if (window.XMLHttpRequest) {
        try {
            req = new XMLHttpRequest();
        } catch(e) {
            // Nic nie rób
        }
    }

    if (req) {
        req.open("GET", url, true);
        req.send(null);
        req.onreadystatechange = function() {
            switch(req.readyState) {
                case 2:
                    if (req.status !== 200) {
                        callback('ERROR');
                        return;
                    }
                    break;
                case 4:
                    callback (req.responseText);
                    break;
            }
        }
    } else {
        // Brak wsparcia dla XMLHttpRequest
        callback('ERROR');
    }
}

// Budowanie
function purchase(buildingId, col, row, callback) {
    var url = SERVER_PATH_URL + 'purchase.php';

    url += "?buildingId=" + buildingId;
    url += "&x=" + col;
```

```
    url += "&y=" + row;

    request(url, callback);
}

// Burzenie
function demolish(col, row) {
    var url = SERVER_PATH_URL + 'demolish.php';

    url += "?x=" + col;
    url += "&y=" + row;

    request(url, callback);
}

// Synchronizacja
function sync() {
    var url = SERVER_PATH_URL + 'sync.php';

    request(url, callback);
}
```

Korzystanie z tych funkcji jest bardzo proste. Aby postawić drzewo (`buildingId = 4`) w czwartym wierszu i piątej kolumnie, wystarczy wywołać kod:

```
purchase(4, 5, 4, function(resp) {
    if (resp.substr(0, 3) == 'OK:') {
        var buildingInstanceId = resp.substr(3, resp.length);
        alert("Budowa zakończona powodzeniem. Identyfikator egzemplarza: " +
              buildingInstanceId);
    } else {
        alert("Podczas tworzenia budynku wystąpił błąd!")
    }
});
```

Jednak przed umożliwieniem postawienia budynku musimy sprawdzić, czy na kliknięty polu siatki (i wszystkich sąsiednich polach) jest wolne miejsce, a więc czy nie znajduje się tam inny budynek lub jego część (`BuildPortion`). Za tę operację odpowiada metoda `checkIfTileIsBusy()` klasy `Game`:

```
Game.prototype.checkIfTileIsBusy = function(obj, col, row) {
    for (var i = (col + 1) - obj.tileWidth; i <= col; i++) {
        for (var j = (row + 1) - obj.tileHeight; j <= row; j++) {
            if (this.tileMap[i] != undefined && this.tileMap[i][j] != null) {
                return true;
            }
        }
    }

    return false;
}
```

Teraz mamy już wszystkie niezbędne elementy obsługujące zdarzenia, więc pozostało jeszcze uzupełnienie implementacji metody `Game.prototype.handleMouseDown`:

```
Game.prototype.handleMouseDown = function(e) {
    e.preventDefault();

    switch (Tools.current) {
        case Tools.BUILD:
            if (this.buildHelper.current != null) {
                var pos = this.translatePixelsToMatrix(e.clientX, e.clientY);
```

```
            // Czy w siatce możemy umieścić element?
            if (!this.checkIfTileIsBusy(this.buildHelper.current, pos.col, pos.row)) {

               var obj = this.buildHelper.current;
               var t = this;

               var processResponse = function(resp) {
                  if (resp.substr(0, 3) == 'OK:') {
                     var buildingInstanceId = resp.substr(3, resp.length);
                     for (var i = (pos.col + 1) - obj.tileWidth; i <= pos.col; i++) {
                        for (var j = (pos.row + 1) - obj.tileHeight; j <= pos.row; j++) {
                           t.tileMap[i] = (t.tileMap[i] == undefined) ? [] :
                                 t.tileMap[i];
                           t.tileMap[i][j] = (i === pos.col && j === pos.row) ? obj :
                                 new BuildingPortion(obj.buildingTypeId, i, j);
                        }
                     }
                  } else {
                     alert("Podczas tworzenia budynku wystąpił błąd!")
                  }

                  t.draw();
               }

               purchase(obj.buildingTypeId, pos.col, pos.row, processResponse);
            } else {
               alert("W tej lokalizacji nie można umieścić budynku");
            }
         }

         break;
      case Tools.MOVE:
         this.dragHelper.active = true;
         this.dragHelper.x = e.clientX;
         this.dragHelper.y = e.clientY;
         break;
      case Tools.ZOOM_IN:
         this.zoomIn();
         break;
      case Tools.ZOOM_OUT:
         this.zoomOut();
         break;
      case Tools.DEMOLISH:
         var pos = this.translatePixelsToMatrix(e.clientX, e.clientY);

         if (this.tileMap[pos.col] != undefined && this.tileMap[pos.col][pos.row] !=
                     undefined) {
            var obj = this.tileMap[pos.col][pos.row];

            // To nie jest budynek, tylko jego część. Pobierz referencję do oryginalnego budynku.
            if (obj instanceof BuildingPortion) {
               pos.col += obj.x;
               pos.row += obj.y;
               obj = this.tileMap[pos.col][pos.row];
            }

            var t = this;
            var processResponse = function(resp) {
               if (resp.substr(0, 2) == 'OK') {
                  // Sprawdź sąsiednie pola i usuń również części budynku.
                  for (var i = (pos.col + 1) - obj.tileWidth; i <= pos.col; i++) {
                     for (var j = (pos.row + 1) - obj.tileHeight; j <= pos.row; j++) {
                        t.tileMap[i][j] = null;
```

```
                    }
                }
            } else {
                alert("Podczas usuwania budynku wystąpił problem!");
            }

            t.draw();
        }

        demolish(pos.col, pos.row, processResponse);
    }

    break;
}

this.draw();
}
```

Mimo że możemy już budować i burzyć budynki (co jest realizowane poprzez dodawanie lub usuwanie egzemplarzy budynku w bazie danych), stan konta w ogóle się nie zmienia. Aby rozwiązać ten problem, zaimplementujemy funkcję `refresh()`, która będzie wywoływana co 15 sekund, a jej zadaniem będzie aktualizowanie stanu konta. Plik *sync.php* po stronie serwera będzie również odpowiadał za obliczanie bieżącego zysku generowanego przez budynki:

```
function refresh() {
    var processResponse = function(resp) {
        if (resp.substr(0, 5) == 'ERROR') {
            alert("Podczas próby zsynchronizowania serwisu wystąpił błąd.");
        } else {
            var balanceContainer = document.getElementById('balance');

            var currBalance = parseInt(balanceContainer.innerHTML);
            var balance = parseInt(resp.substr(3, resp.length));
            balanceContainer.innerHTML = balance;
        }
    }

    sync(processResponse);
    setTimeout(function() {
        refresh(ui);
    }, 15000);
}
```

Aktualny wygląd ekranu gry został przedstawiony na rysunku 5.4.

Ostatni szlif

Dzięki kosmetycznym poprawkom produkt (w naszym przypadku gra) zyskuje swój niepowtarzalny charakter. Chociaż w tej chwili gra jest w pełni funkcjonalna, nie wyróżnia się niczym szczególnym, może być nużąca i nieciekawa.

Jednym z możliwych rozwiązań tego problemu jest sprawienie, by gra miała w sobie więcej życia, poprzez wprowadzenie dynamicznych obiektów poruszających się po mieście lub ponad nim, jak na przykład chmur. Do ich wyświetlenia zamiast obiektu `canvas` (co wiązałoby się z koniecznością ciągłego przerysowywania całej planszy przy każdym ruchu takiego obiektu) użyjemy możliwości oferowanych przez animacje CSS3.

Rysunek 5.4. Działająca gra

Animacje CSS3 pozwalają na modyfikowanie jednej lub wielu właściwości CSS przez określoną liczbę klatek kluczowych:

```
@moveToRight {
   0% {
      left: 0px
   }

   50% {
      left: 100px
   }

   100% {
      left: 200px
   }
}
```

Po zdefiniowaniu klatek kluczowych animacji (w tym przypadku animowany element rozpoczyna ruch w pozycji `left: 0px` i dociera do pozycji `left: 200px`) musimy zająć się innymi właściwościami, takimi jak:

- `animation-timing-function` — steruje sposobem przechodzenia do następnej klatki kluczowej. Dostępne wartości to: `ease`, `linear`, `ease-in`, `ease-out`, `ease-in-out`, `cubic-bezier(`*`liczba`*`, `*`liczba`*`, `*`liczba`*`, `*`liczba`*`)`.
- `animation-name` — określa nazwę animacji (w tym przypadku będzie to `moveToRight`).
- `animation-duration` — definiuje długość trwania całej animacji.

- `animation-iteration-count` — określa liczbę powtórzeń animacji; jako wartość przyjmuje liczbę całkowitą lub stałą `infinite`.
- `animation-direction` — pozwala na zdefiniowanie „kierunku" animacji. Dopuszczalne wartości to: `normal` (animuje od klatki kluczowej 0 do 100) i `alternate` (animuje od klatki kluczowej 100 do 0).

Pełną listę właściwości możesz znaleźć w roboczej wersji specyfikacji W3C pod adresem *http://www.w3.org/TR/css3-animations/*.

W naszym przypadku animacja zawsze będzie wyświetlana w tym samym położeniu. Ten sam efekt można w prosty sposób uzyskać za pomocą skryptu JavaScript i zmiennych o losowych wartościach lub wartościach uwzględniających bieżące położenie siatki:

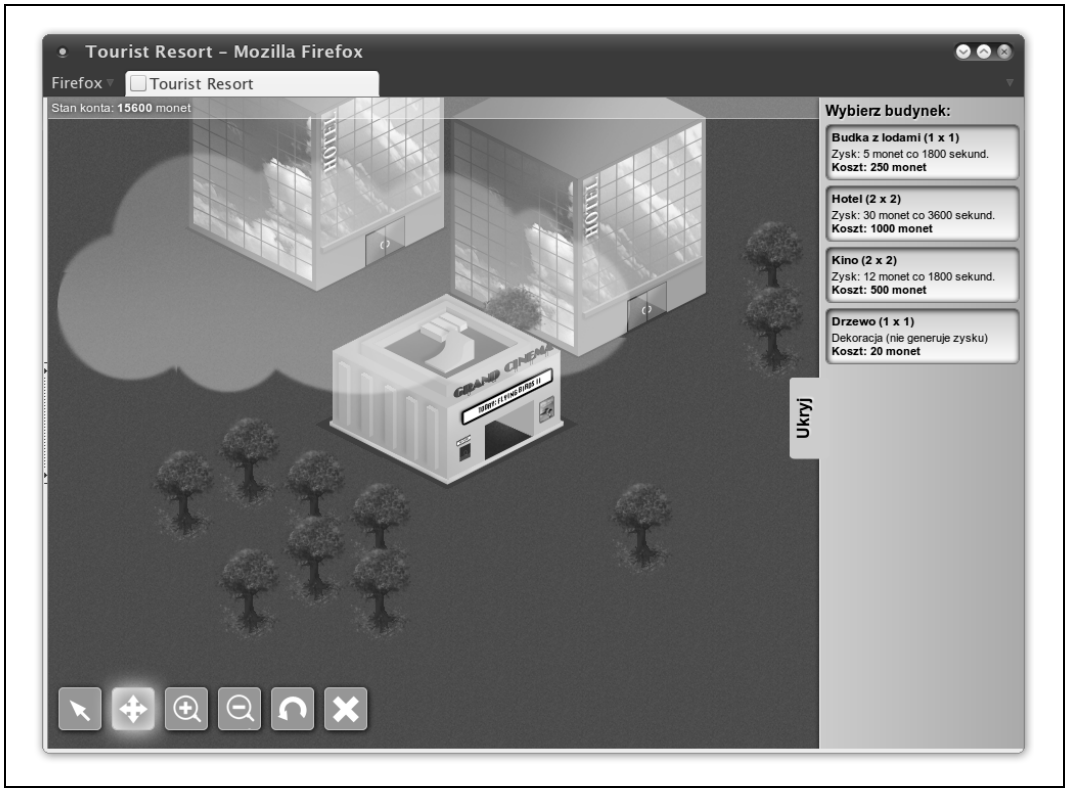
```
@-webkit-keyframes moveFromLeftToRight {
   0% {
      -webkit-transform: translateX(-500px) translateY(50px) translateZ(0px);
   }
   50% {
      -webkit-transform: translateX(0px) translateY(50px) translateZ(0px);
   }
   100% {
      -webkit-transform: translateX(500px) translateY(50px) translateZ(0px);
   }
}

@-moz-keyframes moveFromLeftToRight {
   0% {
      -moz-transform: translateX(-500px) translateY(50px);
   }
   50% {
      -moz-transform: translateX(0px) translateY(50px);
   }
   100% {
      -moz-transform: translateX(500px) translateY(50px);
   }
}

@-ms-keyframes moveFromLeftToRight {
   0% {
      -ms-transform: translateX(-500px) translateY(50px);
   }
   50% {
      -ms-transform: translateX(0px) translateY(50px);
   }
   100% {
      -ms-transform: translateX(500px) translateY(50px);
   }
}

@-o-keyframes moveFromLeftToRight {
   0% {
      -o-transform: translateX(-500px) translateY(50px);
   }
   50% {
      -o-transform: translateX(0px) translateY(50px);
   }
   100% {
      -o-transform: translateX(500px) translateY(50px);
   }
}
```

```
@keyframes moveFromLeftToRight {
   0% {
      transform: translateX(-5000px) translateY(50px);
   }
   50% {
      transform: translateX(0px) translateY(50px);
   }
   100% {
      transform: translateX(5000px) translateY(50px);
   }
}

div.cloud {
   position: absolute;
   top: 0px;
   left: 0px;
   z-index: 500;
   background: transparent url(../../img/spritesheet.png) no-repeat  10px  257px;
   width: 566px;
   height: 243px;

   pointer-events: none;

   -webkit-animation-timing-function: linear;
   -webkit-animation-name: moveFromLeftToRight;
   -webkit-animation-duration: 2s;
   -webkit-animation-iteration-count: infinite;
   -webkit-animation-direction: alternate;

   -moz-animation-timing-function: linear;
   -moz-animation-name: moveFromLeftToRight;
   -moz-animation-duration: 60s;
   -moz-animation-iteration-count: infinite;
   -moz-animation-direction: alternate;

   -ms-animation-timing-function: linear;
   -ms-animation-name: moveFromLeftToRight;
   -ms-animation-duration: 60s;
   -ms-animation-iteration-count: infinite;
   -ms-animation-direction: alternate;

   -o-animation-timing-function: linear;
   -o-animation-name: moveFromLeftToRight;
   -o-animation-duration: 60s;
   -o-animation-iteration-count: infinite;
   -o-animation-direction: alternate;

   animation-timing-function: linear;
   animation-name: moveFromLeftToRight;
   animation-duration: 60s;
   animation-iteration-count: infinite;
   animation-direction: alternate;
}
```

Osiąga się dzięki temu efekt przedstawiony na rysunku 5.5.

Kolejnym efektownym dodatkiem ożywiającym grę może być dodanie cieni. Istnieje kilka technik i algorytmów generujących cienie na dwuwymiarowej scenie. W większości wymagają one jednego lub kilku źródeł światła, a to wiąże się z wykonaniem wielu obliczeń, zwłaszcza jeśli mamy do czynienia z wieloma obiektami (a tak jest w naszym przypadku).

Ponieważ obiekty przez większość czasu są nieruchome, możemy zastosować jedno z dwóch poniższych rozwiązań:



Rysunek 5.5. Chmury przelatujące nad miastem

- rysowanie tekstury cienia dla każdego budynku z osobna,
- dynamiczne i samoczynne generowanie cienia.

Zastosujemy drugie podejście, ponieważ pozwala na zdefiniowanie „udawanego" źródła światła oraz dynamiczne sterowanie intensywnością cienia.

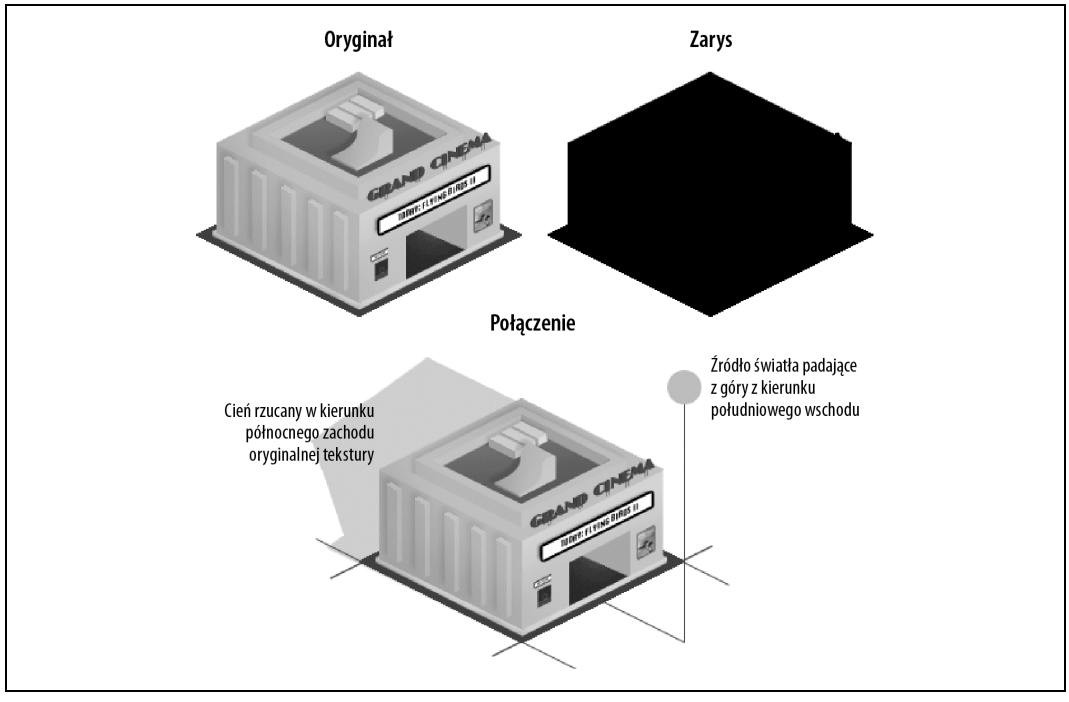
Ta metoda polega na narysowaniu zarysu oryginalnego budynku, wykrzywieniu go i obróceniu oraz zredukowaniu stopnia krycia (czyli zwiększeniu przezroczystości). Jeżeli rysujemy tekstury przy świetle padającym na przykład z południowego wschodu, cienie powinny być rzucane w kierunku północnego zachodu. Jeśli światło pada z południowego zachodu, cienie są rzucane w kierunku północnego wschodu itd. Rysunek 5.6 powinien wiele w tym względzie wyjaśnić.

Aby zaimplementować ten mechanizm, musimy skorzystać z dwóch funkcji obiektu `canvas`, z którymi się już zetknęliśmy: `getImageData()` i `putImageData()`. Całą funkcjonalność zamkniemy w klasie `Sprite`.

Musimy zacząć od zmodyfikowania klasy `Sprite` i dodania pustej właściwości:

```
this.shadow = null;
```

Następnie zmodyfikujemy metodę `draw()`, by przyjmowała opcjonalny parametr `drawShadow`, czyli wartość logiczną określającą, czy do rysowanego właśnie duszka dodać cień. Niewielka dodatkowa procedura sprawdzi, czy `this.shadow` jest `null`, i wykona wszystkie niezbędne



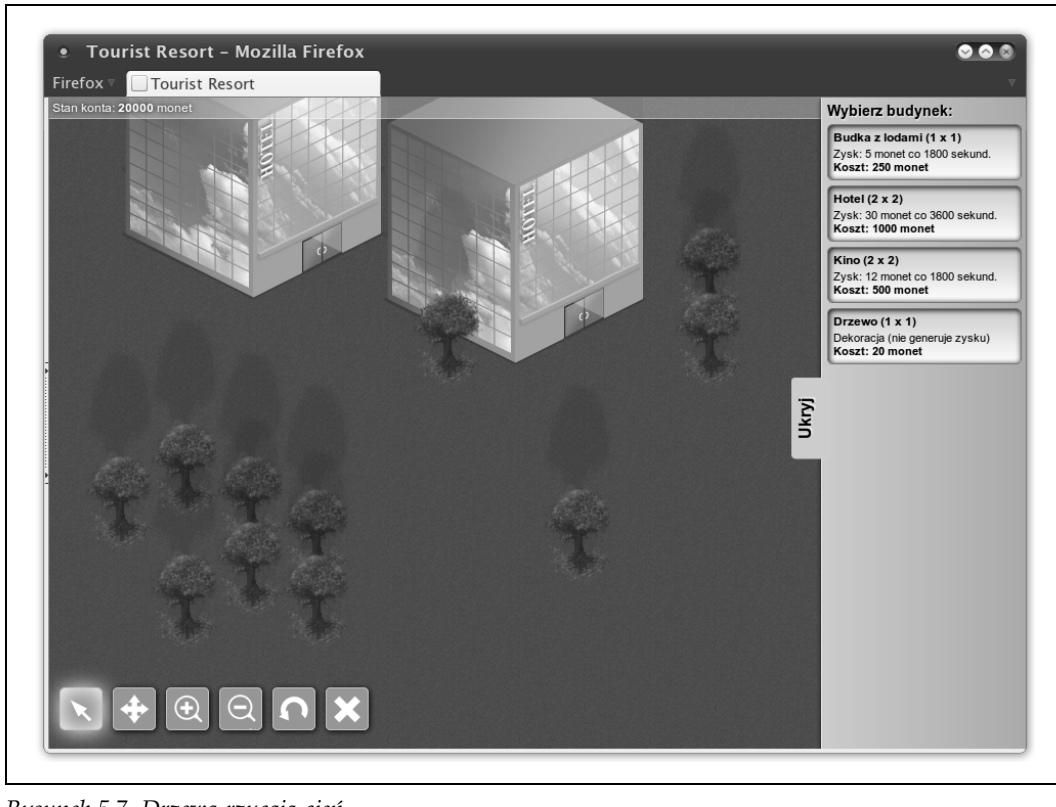
Rysunek 5.6. Dodawanie cienia do budynku

operacje przekształcające obraz na tablicę obrazów, które zostaną przekazane metodzie `putImageData()`. Wynik jest zapisywany we właściwości `this.shadow`, tak by następnym razem nie trzeba było powtórnie przeprowadzać wszystkich operacji związanych z przygotowaniem obrazu. Takie podejście może nie jest tak efektywne jak przygotowanie gotowych obrazów cieni, jednak daje znacznie większą kontrolę i jest bardziej elastyczne:

```
if (this.shown) {
   if (drawShadow !== undefined && drawShadow) {
      if (this.shadow === null) { // Cień nie został jeszcze utworzony
         var sCnv = document.createElement("canvas");
         var sCtx = sCnv.getContext("2d");

         sCnv.width = this.width;
         sCnv.height = this.height;

         sCtx.drawImage(this.spritesheet,
            this.offsetX,
            this.offsetY,
            this.width,
            this.height,
            0,
            0,
            this.width * this.zoomLevel,
            this.height * this.zoomLevel);

         var idata = sCtx.getImageData(0, 0, sCnv.width, sCnv.height);

         for (var i = 0, len = idata.data.length; i < len; i += 4) {
            idata.data[i] = 0; // R
            idata.data[i + 1] = 0; // G
            idata.data[i + 2] = 0; // B
```

```
            }

            sCtx.clearRect(0, 0, sCnv.width, sCnv.height);
            sCtx.putImageData(idata, 0, 0);

            this.shadow = sCtx;
         }

         c.save();
         c.globalAlpha = 0.1;
         var sw = this.width * this.zoomLevel;
         var sh = this.height * this.zoomLevel;
         c.drawImage(this.shadow.canvas, this.posX, this.posY - sh, sw, sh * 2);
         c.restore();
      }

      c.drawImage(this.spritesheet,
         this.offsetX,
         this.offsetY,
         this.width,
         this.height,
         this.posX,
         this.posY,
         this.width * this.zoomLevel,
         this.height * this.zoomLevel);
   }
```

Efekt końcowy został przedstawiony na rysunku 5.7.



Rysunek 5.7. Drzewa rzucają cień

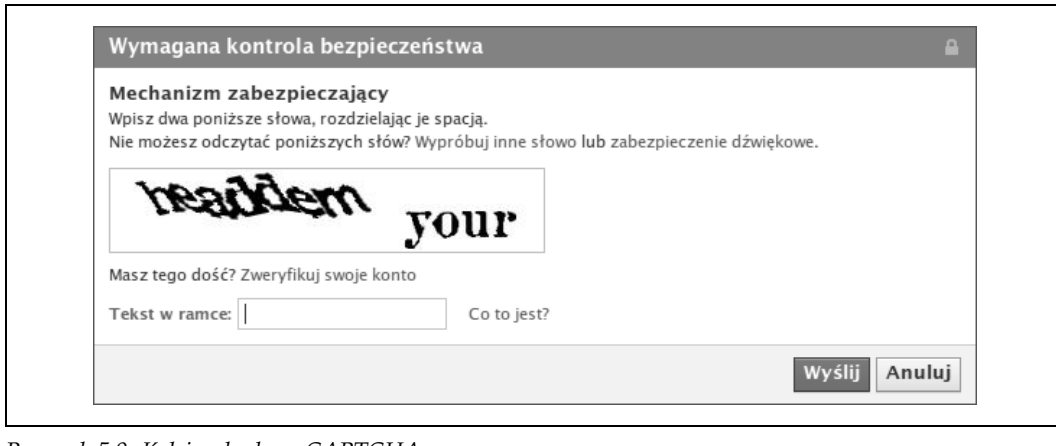
Gra trafia do społeczności — integracja z Facebookiem

Integracja gry z Facebookiem jest prosta — wystarczy wypełnić kilka formularzy i dodać trochę kodu. Przede wszystkim musisz utworzyć nową aplikację w serwisie Facebook. W tym celu przejdź na stronę *http://www.facebook.com/developers/* i kliknij przycisk *Utwórz aplikację*[2]. Zostanie wyświetlony kreator *Nowa aplikacja*, za pomocą którego możesz zintegrować grę z Facebookiem.

W pierwszym oknie kreatora (patrz rysunek 5.8) w polu *App Display Name* (wyświetlana nazwa aplikacji) musisz podać nazwę aplikacji (może być dowolna, ale weź pod uwagę, że to ona będzie wyświetlana użytkownikom), zgodzić się z warunkami korzystania z platformy i kliknąć przycisk *Kontynuuj*[3]. Ja nazwałem aplikację Tourist Resort. Możesz ją nazwać tak samo, bo nazwa aplikacji nie musi być unikalna.

Rysunek 5.8. Tworzenie aplikacji w Facebooku

Kolejnym krokiem (przedstawionym na rysunku 5.9) jest weryfikacja za pomocą mechanizmu zabezpieczającego CAPTCHA.



Rysunek 5.9. Kolejny krok — CAPTCHA

[2] Aby utworzyć aplikację, trzeba oczywiście mieć aktywne konto na Facebooku — *przyp. tłum.*

[3] W oknie kreatora znajduje się jeszcze jedno pole — *App Namespace*. Jego znacznie zostanie wyjaśnione w dalszej części rozdziału — *przyp. tłum.*

Po kliknięciu przycisku *Wyślij*, w zależności od ustawień konta i lokalizacji, może być wymagane przeprowadzenie dodatkowej weryfikacji konta za pomocą numeru telefonu lub karty kredytowej.

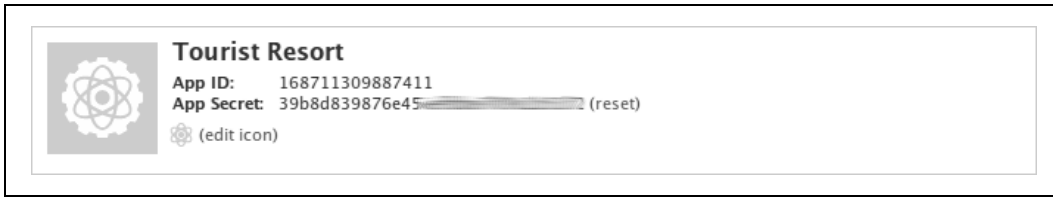
Po zakończeniu tego etapu zostanie wyświetlona strona zarządzania aplikacją, na której możesz zmodyfikować informacje kontaktowe, nazwę aplikacji, jej opis, logo, język i wiele innych szczegółów.

Aplikację możemy zintegrować z platformą Facebook, wyświetlając ją na stronie serwisu lub tworząc odrębną aplikację stosującą schemat uwierzytelniania Facebook Connect. Omówienie wszystkich dostępnych sposobów uwierzytelniania wykracza poza ramy książki, więc skupimy się na sposobie wyświetlania gry *w obrębie* strony Facebooka.

Umieszczenie zewnętrznej strony wewnątrz serwisu jest możliwe dzięki stosowanemu w Facebooku elementowi `canvas` (który nie ma *nic* wspólnego z płótnem z HTML5). Jest to po prostu pływająca ramka (`iframe`) o szerokości 765 pikseli, do której jest ładowana strona aplikacji wraz z parametrami umożliwiającymi zainicjowanie schematu uwierzytelniania OAuth 2.0, dostępnego z poziomu serwera lub klienta (za pomocą JavaScript i XBFML).

Skupiamy się tutaj na rozwiązaniu realizowanym po stronie serwera. Więcej na temat różnych sposobów uwierzytelniania dostępnych na platformie Facebook można znaleźć na stronie *http://developers.facebook.com/docs/authentication/*.

Aby zaimplementować stosowany przez nas schemat uwierzytelniania po stronie serwera, na stronie zarządzania aplikacją musisz kliknąć link *Basic* (podstawowe) znajdujący się w menu *Ustawienia*. Na górze strony są wyświetlane informacje przedstawione na rysunku 5.10.



Rysunek 5.10. Kluczowe informacje dla aplikacji

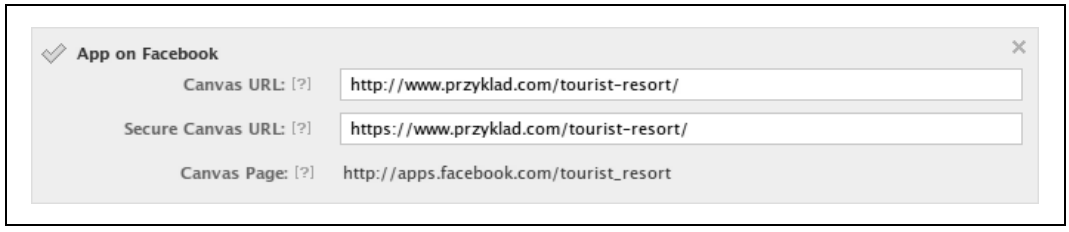
Zanotuj dane z pól *App ID* (identyfikator) oraz *App Secret* (tajny kod; na rysunku jest częściowo zamazany) i pobierz oficjalne SDK Facebooka dla PHP: *https://github.com/facebook/php-sdk/*.

Na stronie podstawowych ustawień aplikacji (*Ustawienia/Basic*) musimy wypełnić dwa dodatkowe pola:

- *App Namespace* (przestrzeń nazwy aplikacji) — adres w ramach serwisu Facebook, pod którym użytkownicy będą widzieć aplikację. Ustalony tu adres jest wyświetlany w innych miejscach jako *Canvas Page* (strona płótna).
- *Canvas URL* (URL płótna) i *Secure Canvas URL* (bezpieczny URL płótna)[4] w sekcji *App on Facebook* (aplikacja na Facebooku) — rzeczywisty adres, pod którym znajduje się aplikacja.

Panel z przykładowymi danymi został przedstawiony na rysunku 5.11.

[4] Od 1 października 2011 roku trzeba podać adres bezpiecznej strony HTTPS — *przyp. tłum.*



Rysunek 5.11. Definiowanie elementu canvas

W skrócie — gdy użytkownik przejdzie na stronę *http://apps.facebook.com/tourist-resort*, Facebook automatycznie wywoła adres podany w polu *Canvas URL* i przekaże kilka parametrów ułatwiających uwierzytelnianie.

Następnie musimy dołączyć pliki z API Facebooka oraz utworzyć nowy obiekt `Facebook`, korzystając z przypisanego do aplikacji identyfikatora i tajnego kodu. Dzięki temu uzyskamy dostęp do informacji o użytkowniku:

```
<?php
   require 'facebook/src/facebook.php';

   $fb = new Facebook(array(
         'appId' => 'identyfikator_aplikacji',
         'secret' => 'tajny_kod_aplikacji',
   ));

   $user = $fb->getUser();

   echo '<pre>';
   print_r($user);
   echo '</pre>';
?>
```

Jeśli użytkownik nie jest zalogowany lub nie został jeszcze uwierzytelniony w aplikacji, wartość zmiennej `$value` jest równa 0. Lepszym sposobem wykrycia tej sytuacji jest zastosowanie podejścia, które można znaleźć w przykładowym pliku *example.php* dołączonym do SDK Facebooka dla PHP:

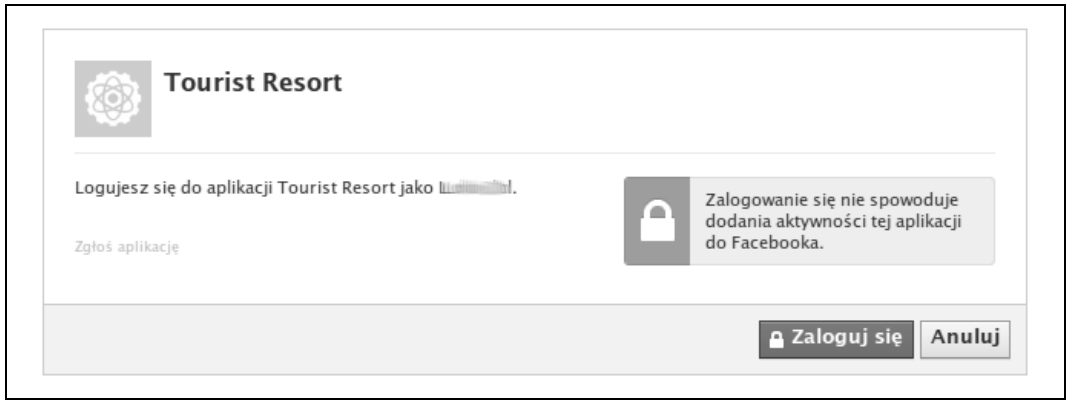
```
<?php
if ($user) {
   try {
      $user_profile = $fb->api('/me');
   } catch (FacebookApiException $e) {
      $user = null;
   }
}
?>
```

W bloku `try` do zmiennej `$user_profile` jest przypisywany wynik wywołania `$fb->api('/me')`, który wysyła zapytanie o własne konto użytkownika do Graph API Facebooka (więcej na ten temat możesz znaleźć na stronie *http://developers.facebook.com/docs/reference/api/*). Jeśli operacja się nie powiedzie, wyrzucany jest wyjątek.

Aby umożliwić zarejestrowanie się użytkownika w aplikacji, musisz dodać poniższy kod:

```
<?php
$loginUrl = $fb->getLoginUrl();
?>
<a href="<?=$loginUrl?>">Zarejestruj się</a>
```

Po kliknięciu linku *Zarejestruj się* zostanie wyświetlone okno autoryzacji Facebooka, przedstawione na rysunku 5.12.



Rysunek 5.12. Wyrażenie zgody na dostęp do danych

Jeśli użytkownik zgodzi się na udostępnienie danych, strona zostanie przeładowana i tym razem w zmiennej `$user` znajdzie się identyfikator użytkownika Facebooka, a w zmiennej `$user_profile` — odpowiedź na zapytanie o własne konto (`/me`) skierowane do Graph API, zwrócona w postaci obiektu JSON.

Facebook domyślnie pyta jedynie o „podstawowe informacje" o użytkowniku, do których zaliczają się: imię, nazwisko, identyfikator, lokalizacja, zainteresowania, płeć itd. Jeśli chcemy poprosić o dodatkowe zezwolenia, by na przykład mieć dostęp do adresu e-mail i znajomych oraz móc publikować komunikaty, musimy zmodyfikować wywołanie `$fb->getLoginUrl()` w następujący sposób:

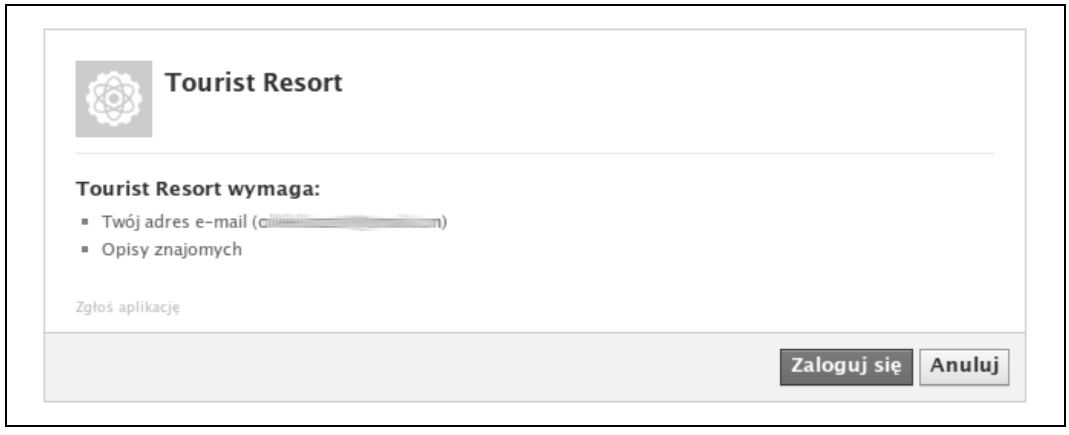
```
<?php
$loginUrl = $fb->getLoginUrl(array(
            "scope" => "email, publish_stream, friends_about_me"
));
?>
```

Pełną listę zezwoleń możesz znaleźć pod adresem *http://developers.facebook.com/docs/authentication/permissions/*.

Po zmodyfikowaniu zezwoleń okno uwierzytelniania wygląda jak na rysunku 5.13.

Jeśli użytkownik zgodzi się na udostępnienie tych danych, zmienna `$user_profile` będzie zawierała również adres e-mail, z którego możemy skorzystać do automatycznego rejestrowania użytkownika we własnej bazie danych.

Skoro mamy dostęp do wszystkich informacji, możemy w bazie danych naszej gry zastosować ten sam identyfikator użytkownika co na Facebooku. Dzięki temu możemy automatycznie zalogować użytkowników przechodzących do naszej gry bezpośrednio z Facebooka.



Rysunek 5.13. Wyrażenie zgody na dostęp do dodatkowych danych

Więcej informacji na temat tworzenia aplikacji dla Facebooka możesz znaleźć na stronie *http://developers.facebook.com/docs/guides/canvas/*.

Programowanie

O'REILLY®

Tworzenie izometrycznych gier społecznościowych w HTML5, CSS3 i JavaScript

Masz konto na Facebooku? Pewnie, że tak — wszyscy mają! Musiałeś więc zauważyć gry dostępne w tym serwisie. Może nie oszałamiają fotorealistyczną grafiką i superdynamiczną akcją, mają jednak to coś. Nazywa się to grywalność. Frajdy ze współzawodniczenia nie da się niczym zastąpić. Popularność tego typu gier gwałtownie rośnie, a firma, która wyprodukowała prawdopodobnie najbardziej znaną z nich — Farmville — właśnie wchodzi na giełdę!

Dzięki tej książce również i Ty będziesz mógł spróbować szczęścia. W trakcie lektury dowiesz się, jak wykorzystać nowości HTML5 i CSS3 do osiągnięcia interesujących efektów, takich jak rzut izometryczny. Ponadto zobaczysz, jak wzbogacić aplikację o efekty dźwiękowe oraz przygotować atrakcyjny interfejs użytkownika. Wiedza tutaj zawarta pozwoli Ci na przygotowanie kompletnej gry za pomocą HTML5, CSS3 i JavaScriptu, a następnie zintegrowanie jej z Facebookiem. Teraz to, czy zdobędziesz popularność i osiągniesz sukces finansowy, zależy tylko i wyłącznie od Twojej pomysłowości oraz kreatywności!

Przygotuj grę, która będzie:

- oparta na nowoczesnych technologiach
- zintegrowana z serwisem Facebook
- dostępna dla setek graczy
- zaopatrzona w grafikę 3D
- popularna!

Zaistniej na rynku gier komputerowych!

ISBN 978-83-246-3888-8