Rozdział 2

Gra w kości

W tym rozdziale omawiamy następujące zagadnienia:

- rysowanie na elemencie `canvas`;
- przetwarzanie liczb losowych;
- logika gry;
- wypełnianie informacji w polach formularza.

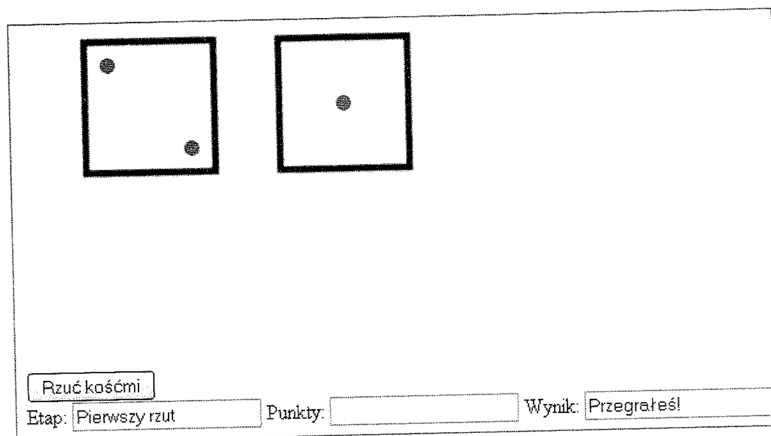
Wprowadzenie

Jedną z najważniejszych nowości wprowadzonych w HTML5 jest element `canvas`. To jest element wyposażony w tzw. kontekst graficzny, który pozwala na tworzenie rysunków, komponowanie ilustracji i pozycjonowanie tekstu w zupełnie swobodny sposób, co stanowi znaczące udoskonalenie w porównaniu z poprzednimi wersjami standardu HTML. Co prawda istniała możliwość formatowania, jednak układ elementów na stronie był raczej statyczny i wyraźnie dominowało uporządkowanie w formie prostokątów. W jaki sposób korzysta się z elementu `canvas`? Stosuje się do tego język skryptowy, jak JavaScript. W tym rozdziale pokażę, w jaki sposób rysować na elemencie `canvas`, omówię istotne cechy języka JavaScript, które będą potrzebne do implementacji gry w kości znanej jako craps. Wyjaśnię też, w jaki sposób definiować funkcje, jak wykorzystywać **liczby pseudolosowe**, jak zaimplementować logikę tej gry i w jaki sposób wyświetlać informacje użytkownikowi. Zanim przejdziemy dalej, musimy się jednak zastanowić nad podstawami gry.

Gra w kości o nazwie craps ma następujące zasady:

Gracz rzuca dwiema kostkami. Liczy się suma oczek, zatem 1 i 3 daje ten sam wynik, co 2 i 2. Suma oczek dwóch sześciennych kostek przyjmuje wartości od 2 do 12. Jeśli gracz uzyska sumę 7 lub 11 w pierwszym rzucie, wygrywa. Jeśli uzyska sumę 2, 3 lub 12, przegrywa. Każdy inny wynik (4, 5, 6, 8, 9 lub 10) powoduje przejście do dalszej gry. Suma oczek jest zapisywana jako tzw. ***punkty gracza*** *i wykonywany jest kolejny rzut. W kolejnym rzucie suma oczek 7 powoduje przegraną, a suma oczek identyczna z punktami gracza powoduje wygraną. Każda inna suma oczek wymusza kolejny rzut z tymi samymi zasadami.*

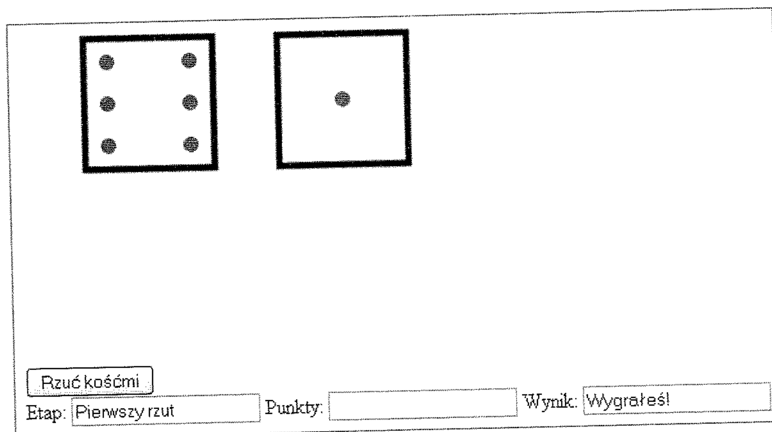
Zobaczmy, jak mógłby wyglądać interfejs naszej gry. Rysunek 2.1 przedstawia wynik rzutu dwiema kostkami na początku gry.



Rysunek 2.1. Przegrana gracza w pierwszym rzucie

Na rysunku 2.1 to nie jest oczywiste, ale nasza gra rysuje kostki w obszarze elementu `canvas`, co oznacza, że strona nie musi być przeładowywana przy każdym rzucie i przeglądarka nie pobiera żadnych obrazków.

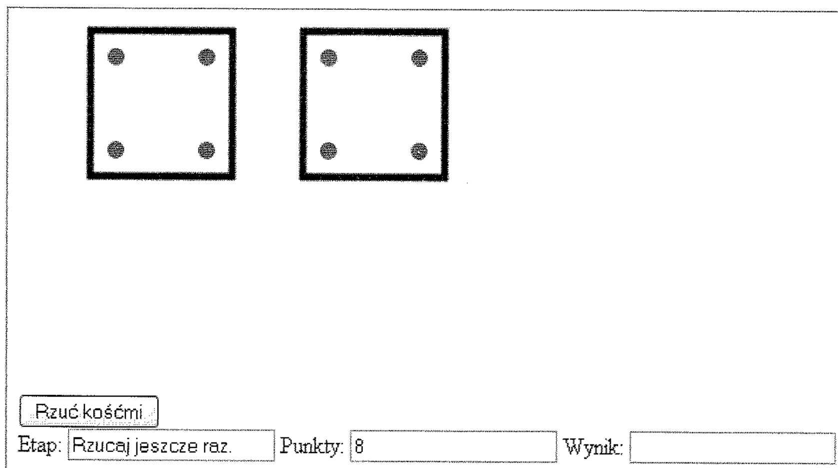
Z opisanych wyżej reguł wynika, że dwie jedynki w pierwszym rzucie oznaczają przegraną gracza. Następny przykład, rysunek 2.2., przedstawia wygraną gracza w pierwszym rzucie dzięki wyrzuceniu sumy 7 oczek.



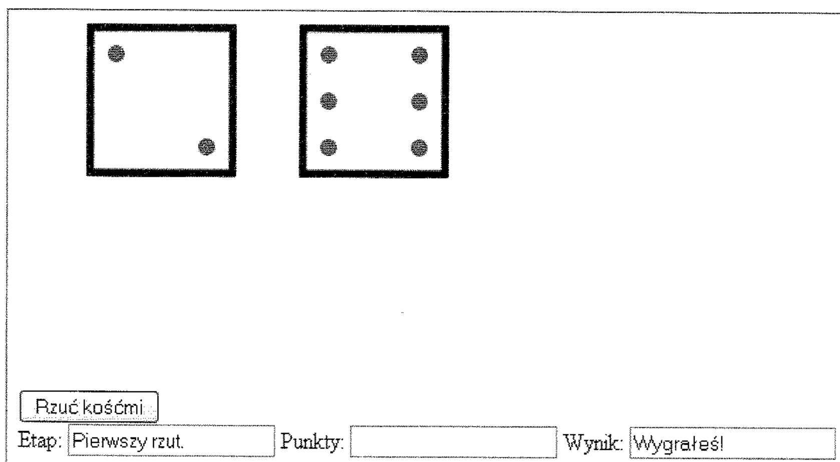
Rysunek 2.2. Suma siedmiu oczek w pierwszym rzucie powoduje wygraną gracza

Rysunek 2.3 prezentuje inną sytuację: suma oczek wynosi 8, co oznacza, że należy wykonać kolejny rzut.

Załóżmy, że gracz ponownie wyrzucił 8 oczek, co powoduje sytuację z rysunku 2.4.



Rysunek 2.3. Suma 8 oczek oznacza konieczność wykonania kolejnego rzutu



Rysunek 2.4. Gracz wygrywa, jeśli w kolejnym rzucie wyrzucił tę samą liczbę oczek, co w pierwszym

Jak widać w przykładowej sekwencji, istotna jest tylko suma oczek obydwu kostek. Punkty gracza zostały ustalone rzutem, w którym wypadły dwie czwórki, ale gra została wygrana kolejnym rzutem, w którym wypadła dwójka i szóstka.

Z reguł wynika, że liczba rund gry jest zmienna. Gracz może wygrać lub przegrać w pierwszym rzucie, ale może też wykonać dowolnie długą sekwencję kolejnych rzutów. Zadanie osoby tworzącej kod gry polega na stworzeniu działającej gry, to znaczy, że gra musi działać zgodnie z regułami, nawet jeśli miałaby nigdy się nie skończyć. Niektórzy moi studenci uważają, że jeśli udało im się wygrać, to dowód, że gra działa prawidłowo. W dobrze napisanej grze gracz powinien wygrywać i przegrywać.

Wymagania

Najważniejszym wymaganiem komputerowej gry symulującej grę w kości jest implementacja mechanizmu losowania wyników rzutu kostką. To może wydać się sprzeczne z logiką, w końcu programowanie komputerów polega na definiowaniu dokładnych instrukcji, co ma być wykonane. Na szczęście, JavaScript, jak prawie wszystkie inne języki programowania, posiada wbudowany mechanizm generowania liczb pseudolosowych. Niektóre języki programowania wykorzystują do tego celu fragment wartości znacznika czasu w milisekundach. Dokładny sposób realizacji tego zadania przez komputer nie ma jednak dla nas zupełnie żadnego znaczenia. Zakładamy, że JavaScript zaimplementowany w przeglądarce spisuje się w tym zakresie w sposób wystarczający do naszych potrzeb.

Załóżmy zatem, że jesteśmy w stanie wygenerować losową liczbę od 1 do 6; do obsługi dwóch kostek zrobimy to za każdym razem dwukrotnie. Następny krok to definicja reguł gry. Chodzi o sposób, w który będziemy mogli stwierdzić, czy jesteśmy w pierwszym, czy w kolejnym rzucie. W programowaniu tego typu logikę nazywa się **śledzeniem stanu** i stanowi ona zagadnienie istotne nie tylko w grach. Następnie musimy skonstruować logikę gry w oparciu o warunki. Konstrukcje warunkowe, jak `if` i `switch`, stanowią standardowe elementy języka programowania i wkrótce zrozumiesz, dlaczego nauczyciele programowania, którzy, jak ja, nigdy nie byli w kasynie, do nauki teorii liczb pseudolosowych chętnie stosują przykład gry w kości.

Musimy dać graczowi możliwość „rzucenia" kośćmi; do tego celu wykorzystamy przycisk na ekranie. Powinniśmy też przekazać graczowi informację o tym, co się stało w wyniku kliknięcia. W tym konkretnym przykładzie zdecydowałam się narysować na ekranie elementy graficzne reprezentujące kości do gry, a dodatkowo wyświetlimy odpowiednią informację tekstową, zawierającą etap gry, liczbę punktów i wynik. Dawniej, gdy do interakcji z użytkownikiem stosowało się tylko tekst, wykorzystywane były operacje **wejścia-wyjścia** (**I/O**). Obecnie wykorzystywana jest **koncepcja graficznego interfejsu użytkownika** (**GUI**), która obejmuje mnóstwo technik interakcji z systemami komputerowymi. Zaliczamy do tego wskazywanie myszką elementów na ekranie i klikanie ich czy też przeciągnie, symulujące operację przenoszenia elementów (co wykorzystamy w przykładzie z rozdziału 4.). Rysowanie na ekranie wymaga wykorzystania układu współrzędnych. W większości języków programowania współrzędne punktów na ekranie są reprezentowane w ten sam sposób, który opiszę niebawem.

Funkcje HTML5, CSS i JavaScriptu

Przyjrzyjmy się teraz bardziej szczegółowo funkcjom HTML5, CSS i JavaScriptu, które będziemy wykorzystywać w naszym kodzie gry w kości.

Przetwarzanie liczb pseudolosowych i wyrażenia matematyczne

Przetwarzanie liczb pseudolosowych w JavaScripcie jest realizowane z użyciem wbudowanej metody `Math.random()`. Z formalnego punktu widzenia `Math` jest **klasą**, a `random()` jest jej **metodą**. Metoda `Math.random()` generuje liczby większe lub równe 0 i mniejsze od 1, czyli otrzymujemy liczby dziesiętne, np. 0.253012. Do naszych celów taki wynik nie jest przydatny, ale przekształcenie

liczby z zakresu [0, 1) do liczby całkowitej z zakresu [1, 6] jest prostym zadaniem. Wynik wywołania `Math.random()` mnożymy przez górny zakres naszego wyniku, czyli 6, co daje wynik z zakresu [0, 6). Jeśli na przykład otrzymaliśmy liczbę 0.253012, po przemnożeniu jej przez 6 otrzymamy 1.518072. Jesteśmy już bliżej pożądanego wyniku. Następny krok polega na odrzuceniu części dziesiętnej i zatrzymaniu części całkowitej. W tym celu wykorzystamy inną metodę klasy `Math`: `Math.floor()`. Ta metoda zwraca liczbę całkowitą, odrzucając część dziesiętną. Jak sugeruje nazwa (ang. *floor* znaczy „podłoga"), metoda `Math.floor()` zaokrągla liczby w dół. W naszym przykładzie zaczęliśmy od 0.253012, następnie otrzymaliśmy 1.518072, zatem w tym momencie mamy liczbę 1. Wykonanie mnożenia przez 6 i wyliczenia wyniku metody `Math.floor()` da nam zawsze liczbę całkowitą z przedziału od 0 do 5. Wystarczy zatem dodać 1 i otrzymamy liczby od 1 do 6, czyli dokładnie to, o co nam chodzi.

Ten sam proces można wykorzystać do dowolnego zakresu. Na przykład, jeśli chcemy wylosować liczbę z przedziału od 1 do 13, wystarczy wyznaczyć liczbę losową, przemnożyć ją przez 13, wyliczyć `Math.floor()` i dodać 1. Takie obliczenia mogą się przydać w implementacji gry w karty. Podobne przykłady przewijają się w innych rozdziałach tej książki.

Wszystkie te obliczenia możemy połączyć w jedno **wyrażenie**. Wyrażenia są kombinacjami wartości, wywołań metod i funkcji oraz innych elementów, którymi zajmiemy się w dalszej części książki. Wszystkie składowe łączy się z użyciem operatorów, jak + w celu dodawania czy * w celu mnożenia.

Z rozdziału 1. wiemy, że elementy HTML można wzajemnie zagnieżdżać. W podobny sposób zagnieżdżaliśmy kod JavaScript:

```
document.write(Date());
```

W tym przypadku możemy zastosować taką samą konstrukcję. Zamiast zapisywać `Math.random()` i `Math.floor()` jako osobne wyrażenia, możemy przekazać pierwsze wywołanie jako parametr drugiego. W efekcie uzyskamy następujący kod:

```
1 + Math.floor(Math.random() * 6)
```

To **wyrażenie** w wyniku da liczbę całkowitą z przedziału od 1 do 6. Operatory + i * realizują operacje arytmetyczne i mają takie samo znaczenie, jak dodawanie i mnożenie w matematyce. W pierwszej kolejności wyznacza się wyrażenia zagnieżdżone. Innymi słowy, powyższy kod wykonuje następującą sekwencję operacji:

- Wywołuje `Math.random()` w celu wygenerowania liczby losowej z zakresu [0, 1).
- Mnoży powyższy wynik przez 6.
- Z użyciem `Math.floor()` odrzuca część dziesiętną powyższego wyniku, pozostawiając liczbę całkowitą.
- Do powyższego wyniku dodaje 1.

W naszym kodzie znajdziesz wyrażenia właśnie tego typu. Zanim jednak zajmiemy się programowaniem, musimy wyjaśnić kilka szczegółów.

Zmienne i instrukcje przypisania

JavaScript, podobnie jak inne języki programowania, wykorzystuje abstrakcję nazywaną **zmienną**, która służy jako pojemnik na wartość, jak np. liczba. Nazwy zmiennych po prostu wiąże się z wartościami. Taka związana wartość może być użyta w dalszej części kodu poprzez użycie nazwy zmiennej. Jako analogii można użyć sytuacji z życia: mówimy „prezydent kraju", ale z kadencji na kadencję może to być inna osoba. Zatem wartość zmiennej „prezydent kraju" jest różna w różnych punktach czasu. W programowaniu wartości zmiennych również mogą ulegać modyfikacjom w trakcie działania programu, stąd nazwa.

Do **zadeklarowania** zmiennej stosuje się instrukcję `var`.

Nazwy zmiennych i funkcji są dowolne, ich wybór zależy od programisty. Istnieją jednak reguły tworzenia tych nazw: nie wolno stosować białych znaków ani znaków przestankowych, nazwa musi zaczynać się literą, podkreślnikiem _ lub znakiem $. Nazwy nie powinny być zbyt długie, bo wiąże się to z koniecznością wpisywania dużej liczby znaków przy każdym użyciu zmiennej, ale nie powinny też być zbyt krótkie, ponieważ łatwo zapomnieć, do czego służyły. Warto zachować spójność nazewnictwa zmiennych, ale nie ma oczywiście konieczności stosowania reguł ortografii. Ważne jednak, aby nazwy te stosować w sposób konsekwentny. Jeśli wykorzystuje się standardowe elementy języka, jak nazwy obiektów (np. `document`) czy metod (jak `random`), należy stosować dokładnie taką pisownię nazw, jakiej oczekuje JavaScript.

We własnym kodzie JavaScript nie należy używać nazw zmiennych identycznych z nazwami wykorzystywanymi przez JavaScript (jak `random` czy `floor`). Warto zadbać o to, by te nazwy były unikalne, ale zrozumiałe. Jeden z powszechnie stosowanych sposobów tworzenia czytelnych nazw zmiennych określa się terminem **camel case**. Polega to na tym, że w przypadku nazw składających się z wielu wyrazów każdy kolejny wyraz rozpoczyna się wielką literą, na przykład `calkowitaLiczbaElementów`, `pierwszyRzutKostka`. Nazwa „camel case" wzięła się z tego, że wielkie litery wyrastają z nazwy, jak garby wielbłąda (ang. *camel* to wielbłąd). Nie ma konieczności stosowania tej konwencji nazewnictwa, ale jest popularna wśród wielu programistów.

Wyrażenie pobierające wartość pseudolosową i przekształcające ją do liczby całkowitej z interesującego nas zakresu będzie wykorzystane w instrukcji przypisania, na przykład:

```
var ch = 1 + Math.floor(Math.random() * 6);
```

Instrukcja ta spowoduje przypisanie zmiennej `ch` wyniku wyrażenia po prawej stronie znaku równości. Dodatkowa instrukcja `var` służy do **deklarowania** nowej zmiennej; w tym przypadku zmienna jest **inicjalizowana w ramach deklaracji**. Symbol równości `=` służy do przypisywania wartości zmiennym z zastosowaniem wyrażeń, jak również wartości stałych, co zademonstruję za chwilę. Nazwa zmiennej `ch` stanowi skrót od *choice* (wybór). Ta nazwa jest dla mnie czytelna i zrozumiała. Jeśli jednak zastanawiasz się, czy lepiej zastosować krótką, czy długą nazwę zmiennej, zawsze warto wybrać dłuższą. Instrukcja kończy się znakiem średnika `;`. Możesz zapytać: dlaczego nie kropką? Kropka jest używana do innych zastosowań: jako znak oddzielający część całkowitą liczby od jej części dziesiętnej oraz jako separator nazw obiektów i atrybutów lub metod, np. `document.write()`.

Instrukcje przypisania stanowią najpowszechniej stosowany typ instrukcji w programowaniu. Oto przykład instrukcji przypisania wykorzystującej stałą znakową (literał):

```
bookname = "Wprowadzenie do HTML5";
```

Użycie znaku równości może być mylące dla początkującego programisty. Zapis ten należy rozumieć jako deklarację, że nazwa po lewej stronie znaku jest równa wyrażeniu po prawej. Ze zmiennymi i różnorakimi sposobami wykorzystania operatorów i przypisań spotkasz się w tej książce wielokrotnie.

*Instrukcja `var` deklaruje zmienną jako **lokalną** w danym zakresie (ang. scope). Język JavaScript nie wymaga deklarowania nazw zmiennych przed ich użyciem, jak to ma miejsce w przypadku innych języków programowania. Pominięcie instrukcji `var` spowoduje, że zmienna zostanie zadeklarowana jako **globalna**, przez co jej zmiany będą widoczne na zewnątrz bieżącego zakresu. Staram się unikać tego typu praktyk, ale w internecie łatwo znaleźć mnóstwo tak niedbałego kodu.*

W grze w kości potrzebne nam są zmienne, które zdefiniują stan gry, a dokładniej, poinformują nas, czy to pierwszy, czy kolejny rzut kośćmi oraz ile punktów gracz zdobył w pierwszym rzucie. W naszej implementacji te wartości będą przechowywane przez **zmienne globalne**, to znaczy zmienne zadeklarowane z instrukcją `var` poza definicją funkcji. Takie zmienne zachowują swój stan między wywołaniami funkcji. Zmienne zdefiniowane z instrukcją `var` we wnętrzu funkcji przestają istnieć po zakończeniu jej wywołania.

Nie zawsze użycie zmiennych jest niezbędne. Na przykład w pierwszej przykładowej aplikacji zmienne zostały użyte do przechowywania pionowej i poziomej pozycji kostki. W miejsce zmiennych można jednak wpisać liczby, ponieważ nie ulegają one zmianie w trakcie działania programu. Jednak do tych wartości odwołuję się w wielu miejscach kodu. Dzięki użyciu zmiennych potencjalna decyzja o zmianie pozycji kostki na ekranie pociągnęłaby za sobą konieczność modyfikacji wartości zmiennej w tylko jednym miejscu kodu zamiast w wielu.

Funkcje użytkownika

JavaScript posiada dużą liczbę wbudowanych funkcji i metod, ale z pewnością nie zawiera wszystkiego, co może być potrzebne. Na przykład, o ile mi wiadomo, nie posiada funkcji symulujących rzut kostką do gry. Ale na szczęście pozwala nam tworzyć własne funkcje. Te funkcje mogą przyjmować **argumenty**, jak metoda `Math.floor()`, albo działać bez argumentów, jak `Math.random()`. Argumenty są wartościami przekazywanymi do funkcji. Można je traktować jak dodatkowe informacje. Definicja funkcji składa się ze słowa kluczowego `function`, po którym następuje nazwa, którą chcemy nadać funkcji, po niej nawiasy okrągłe, w których wylicza się listę argumentów, po czym następuje kod funkcji ujęty w nawiasy klamrowe. Nazwy funkcji, tak samo jak nazwy zmiennych, są definiowane przez programistę. Poniższy listing prezentuje definicję funkcji zwracającej wynik mnożenia jej dwóch argumentów. Jak sugeruje nazwa funkcji, służy ona do obliczania pola powierzchni prostokąta:

```
function areaOfRectangle(wd, ln) {
  return wd * ln;
}
```

Ważnym szczegółem powyższego kodu jest słowo kluczowe `return`. Informuje ono interpreter JavaScriptu o tym, jaką wartość ma zwrócić funkcja. Funkcję z powyższego przykładu możemy wykorzystać w przypisaniu `rect1 = areaOfRectangle(5, 10)`, co spowoduje, że zmiennej `rect1` zostanie

przypisana wartość 50 (czyli wynik działania 5 · 10). Definicja funkcji jest zapisana w postaci kodu w elemencie `script`. W rzeczywistych warunkach pisanie tak prostej funkcji może nie mieć wiele sensu, ponieważ łatwiej jest zapisać tę operację arytmetyczną bezpośrednio w wyrażeniu, ale to użyteczny przykład funkcji definiowanej przez użytkownika. Po wywołaniu tej definicji, co następuje najczęściej w ramach ładowania dokumentu HTML, inny kod może użyć tej funkcji, po prostu wywołując ją po nazwie, np. `areaOfRectangle(100, 200)` lub `areaOfRectangle(x2 - x1, y2 - y1)`.

Drugi z tych przykładów wykorzystuje zmienne `x1`, `x2`, `y1` i `y2`, które powinny być zadeklarowane w kodzie wcześniej w stosunku do wywołania.

Funkcje można wywoływać z poziomu niektórych atrybutów elementów HTML. Na przykład element `body` posiada atrybut `onLoad`:

```
<body onLoad="init();">
```

Mój kod JavaScript definiuje funkcję o nazwie `init()`. Powyższa deklaracja elementu `body` spowoduje, że po załadowaniu dokumentu HTML JavaScript wywoła funkcję `init()`. Taka sytuacja występuje zarówno przy pierwszym otwarciu dokumentu, jak również po kliknięciu przycisku przeglądarki *Odśwież stronę*.

Inny przykład przywiązania funkcji JavaScript do elementu HTML wykorzystuje nową funkcję standardu HTML 5 — element `button`:

```
<button onClick="throwdice();">Rzuć kośćmi</button>
```

Ten kod utworzy przycisk z etykietą *Rzuć kośćmi*. Gdy gracz kliknie ten przycisk, JavaScript wywoła funkcję `throwdice()` zdefiniowaną w elemencie `script`.

W podobny sposób można z JavaScriptem powiązać element formularza `form`, który opiszę za chwilę.

Kontrola logiki kodu: instrukcje if i switch

Gra craps ma zdefiniowany zestaw reguł. Można je podsumować następująco: jeśli to pierwszy rzut, sprawdzamy, czy wyrzucona liczba oczek należy do pewnych zbiorów wartości. Jeśli to drugi rzut, sprawdzamy inne zbiory wartości. Do realizacji tego typu warunków w JavaScripcie służą instrukcje `if` i `switch`.

Instrukcja `if` wykorzystuje **warunki**, które mogą być wykorzystane do porównania wartości, np. czy zmienna `temp` ma wartość większą od 85 albo czy zmienna `course` ma wartość równą `"Programowanie gier"`. Porównania zwracają wartości **boolowskie**, czyli `true` lub `false` (prawda lub fałsz). Do tej pory mieliśmy do czynienia głównie z wartościami liczbowymi oraz łańcuchami znaków. Wartości boolowskie są jeszcze jednym typem, z jakim spotykamy się w programowaniu w JavaScripcie. Nazwa **wartości boolowskie** pochodzi od nazwiska matematyka George'a Boole'a. Przykładowe warunki opisane w tym akapicie będą zapisane, odpowiednio, jako

```
temp > 85
```

oraz

```
course == "Programowanie gier"
```

Pierwsze z tych wyrażeń należy odczytywać jako: „czy wartość zmiennej `temp` jest większa niż 85?", drugie odczytamy jako: „czy wartość zmiennej `course` jest łańcuchem znaków o wartości `Programowanie gier`?".

Wyrażenia nierówności są proste w budowie: znaku większości > używamy po to, by sprawdzić, czy pierwsza wartość jest większa od drugiej, znaku mniejszości po to, by sprawdzić, czy pierwsza jest mniejsza. Wartością wyrażenia jest w tym przypadku odpowiedź na pytanie zdefiniowane przez wyrażenie: `true`, jeśli odpowiedź brzmi „tak", a `false` w przeciwnym wypadku.

Wyrażenie równości jest nieco bardziej skomplikowane. Można by się zastanawiać nad znaczeniem podwójnego znaku równości, użycie znaków cudzysłowu również wydaje się nieco nietypowe. Operator równości w JavaScripcie (i wielu innych językach programowania) składa się z dwóch znaków równości. Potrzebujemy dwóch znaków, ponieważ pojedynczy znak równości służy jako operator przypisania i nie może być użyty w tych dwóch rolach bez wywoływania niejasności. Gdybyśmy bowiem napisali `course = "Programowanie gier"`, spowodowalibyśmy zmianę wartości zmiennej `course`, a nie porównanie. Znaki cudzysłowu służą do definiowania łańcuchów znaków, w naszym przypadku znaki pomiędzy znakami cudzysłowów definiują pojedynczy łańcuch znaków, rozpoczynający się znakiem `P`, a kończący znakiem `r` i zawierający w środku znak spacji.

Posiadamy wiedzę o warunkach, możemy więc przystąpić do stworzenia kodu wykonywanego w przypadku, gdy warunek jest spełniony (przyjmuje wartość `true`):

```
if (warunek) {
   kod
}
```

Jeśli chcielibyśmy, żeby kod wykonał jedną czynność w efekcie spełnienia warunku, a inną w efekcie jego niespełnienia, zastosujmy następującą składnię:

```
if (warunek) {
  kod dla wartości true
} else {
  kod dla wartości false
}
```

W powyższych dwóch listingach zastosowaliśmy pochyloną czcionkę, ponieważ nie jest to rzeczywisty kod, a tzw. **pseudokod**, który opisuje składnię prawdziwego kodu JavaScript.

Oto prawdziwe przykłady. Wykorzystamy wbudowaną funkcję `alert()`, która powoduje wyświetlenie przez przeglądarkę niewielkiego okna zawierającego komunikat przekazany jako argument jej wywołania. Użytkownik musi kliknąć *OK* w celu kontynuowania.

```
if (temp > 30) {
  alert("Jest za gorąco!");
}
if (age > 18) {
  alert("Twój wiek upoważnia Cię do zakupu alkoholu.");
} else {
  alert("Jesteś za młody, żeby być obsłużonym w barze.");
}
```

Naszą grę w kości moglibyśmy napisać wyłącznie z użyciem instrukcji `if`. Jednak JavaScript oferuje nam dodatkową konstrukcję, która nieco upraszcza życie — `switch`. Ogólny format jest następujący:

```
switch(x) {
  case a:
    kod dla warunku a;
  case b:
    kod dla warunku b;
  default:
    kod dla innych wartości;
}
```

JavaScript wylicza wartość zmiennej `x` w pierwszym wierszu instrukcji `switch` i porównuje ją z wartościami dla każdej instrukcji `case`. Jeśli następuje dopasowanie, czyli w przypadku powyższego kodu wartość `x` jest równa `a` lub `b`, zostanie wywołany kod z odpowiedniego bloku. Jeśli dopasowanie nie zostało odnalezione, wywoływany jest kod z bloku `default`. Blok `default` nie jest wymagany. JavaScript będzie kontynuował przeglądanie kolejnych instrukcji `case`, nawet jeśli dopasował już zmienną i wykonał kod dla dopasowania. To może być pożądane zachowanie, ale jeśli chcemy przerwać instrukcję `switch` po pierwszym dopasowaniu, należy na końcu bloku `case` wywołać instrukcję `break`.

Prawdopodobnie już widzisz, w jaki sposób instrukcje `if` i `switch` mogą być użyte w naszej grze w kości. Jeśli nie, dowiesz się w kolejnym podrozdziale. Najpierw jednak przyjrzyjmy się przykładowi, który zwróci informację o liczbie dni miesiąca podanego z użyciem trzyliterowego skrótu (`sty`, `lut` itd.):

```
switch(mon) {
  case "kwi": case "cze": case "wrz": case "lis":
    alert("Ten miesiąc ma 30 dni.");
    break;
  case "lut":
    alert("Ten miesiąc ma 28 lub 29 dni.");
    break;
  default:
    alert("Ten miesiąc ma 31 dni.");
}
```

Jeśli zmienna `mon` ma wartość `"kwi"`, `"cze"`, `"wrz"` lub `"lis"`, zostanie wyświetlone okienko z informacją, że dany miesiąc ma 30 dni, po czym przetwarzanie bloku `case` się zakończy dzięki zastosowaniu instrukcji `break`. Jeśli zmienna `mon` ma wartość `"lut"`, zostanie wyświetlona informacja o tym, że ten miesiąc ma 28 lub 29 dni; również w tym przypadku przetwarzanie jest przerywane instrukcją `break`. Jeśli wartość zmiennej `mon` jest dowolnie inna, zostanie wyświetlona informacja o tym, że miesiąc ma 31 dni. Przy okazji: błędna nazwa miesiąca spowoduje wyświetlenie informacji o 31 dniach, co stanowi przykład sytuacji, której warto zapobiec.

HTML ignoruje znaki przejścia do nowego wiersza, podobnie JavaScript nie zważa na formatowanie kodu. Jeśli chcesz, możesz wpisać wszystko w pojedynczym wierszu. Formatowanie kodu w czytelny sposób znacznie jednak ułatwia pracę nad nim.

Rysowanie na elemencie canvas

Nadszedł czas, by zapoznać się z jedną z najważniejszych nowości w HTML 5 — elementem canvas. Wyjaśnię, jakie techniki programowania są niezbędne do pracy z canvas, po czym pokażę kilka prostych przykładów. Następnie wrócimy do naszego zadania i narysujemy kostkę do gry na elemencie canvas. Jak pamiętamy, szkielet dokumentu HTML wygląda następująco:

```
<html>
  <head>
    <title>...</title>
    <script>...</script>
  </head>
  <body>
    ...Tutaj znajduje się statyczna treść dokumentu...
  </body>
</html>
```

Do pracy z elementem canvas potrzebujemy elementu canvas, wstawionego za pomocą znacznika canvas w dokumencie HTML, oraz kodu JavaScript w elemencie script, definiującego, co i jak chcemy rysować. Na początek zdefiniujmy element canvas:

```
<canvas id="canvas" width="400" height="300">
  Twoja przeglądarka nie obsługuje elementu canvas standardu HTML5.
</canvas>
```

Jeśli tak zdefiniowany dokument zostanie otwarty w przeglądarce nieobsługującej elementu canvas, użytkownik zobaczy komunikat *„Twoja przeglądarka nie obsługuje elementu canvas standardu HTML5"*. Jeśli tworzona strona WWW jest pisana również z myślą o użytkownikach starszych przeglądarek, kod w elemencie canvas mógłby zawierać odnośnik do innej części strony WWW, przystosowanej do tego typu przeglądarek. Może też na przykład zawierać poradę, jak zdobyć odpowiednią przeglądarkę. W naszej książce zakładamy, że interesują nas wyłącznie użytkownicy nowoczesnych przeglądarek, obsługujących HTML5.

Element canvas z naszego przykładu posiada atrybut id o wartości canvas. Wartość ta może być dowolna, ale użyliśmy canvas dla uproszczenia. Jeśli chcemy użyć w dokumencie większej liczby elementów canvas, musimy im nadać odmienne atrybuty id. W tej aplikacji nie potrzebujemy większej liczby, nie musimy zatem się tym przejmować. Atrybuty width i height definiują odpowiednio szerokość i wysokość elementu canvas na ekranie.

Element canvas został umieszczony w elemencie body, rzućmy zatem okiem na JavaScript. Pierwszy krok w rysowaniu na elemencie canvas polega na zainicjalizowaniu odpowiedniego obiektu w kodzie JavaScript. Do tego celu przyda nam się zmienna, którą nazwiemy ctx:

```
var ctx;
```

Zmienną zadeklarujemy na zewnątrz funkcji, co spowoduje, że stanie się ona zmienną globalną, to znaczy dostępną w każdej funkcji naszego kodu. Zmienna ctx będzie zawierała obiekt kontekstu graficznego, niezbędny do realizacji operacji rysowania. Nazwa zmiennej ctx wzoruje się na wielu przykładach dostępnych na różnych stronach WWW, ale można użyć dowolnej.

We wszystkich przypadkach użycia przez nas kontekstu graficznego (które znajdziesz w przykładach w dalszej części książki, a ich kod źródłowy możesz pobrać z serwera FTP) zmienna ctx będzie zainicjalizowana w następujący sposób:

```
ctx = document.getElementById('canvas').getContext('2d');
```

Ten fragment kodu wydobywa z dokumentu element o atrybucie id równym canvas, a z tego elementu wydobywa kontekst graficzny pod nazwą 2d. Jak się można domyślić, element canvas obsługuje więcej kontekstów, ale na razie wystarczy nam ten pod nazwą 2d.

W kodzie JavaScript można rysować wielokąty, ścieżki składające się z odcinków i łuków, można też wyświetlać ilustracje ładowane z plików. Wielokąty i ścieżki można wypełniać. Zanim jednak zaczniemy bawić się tymi wszystkimi dobrodziejstwami, musimy opanować wiedzę z zakresu układów współrzędnych i miar łukowych w radianach.

Współrzędne na ekranie są definiowane w podobny sposób, co współrzędne na mapie. W przypadku ekranu jednostką układu współrzędnych jest piksel. W poprzednim rozdziale mieliśmy już okazję użyć tej jednostki do określenia szerokości i wysokości ilustracji oraz szerokości ramki. Piksel jest dość niewielką jednostką miary, co można łatwo wykazać w eksperymentach. Jednak sama jednostka nie wystarczy do wyznaczenia współrzędnych. Potrzebny jest też punkt odniesienia, względem którego mierzone są współrzędne, tak samo jak systemy GPS wykorzystują punkt przecięcia południka zerowego i równika. W przypadku dwuwymiarowego prostokąta, jakim jest element canvas, tym punktem odniesienia jest tzw. **punkt początkowy** (ang. *origin*), czyli lewy górny wierzchołek elementu canvas. Warto zwrócić uwagę, że w rozdziale 6., w którym będziemy definiować wzajemne położenie elementów HTML, wykorzystamy podobny układ współrzędnych dla całego dokumentu, a nie samego elementu canvas. Również w tym przypadku punktem początkowym będzie lewy górny wierzchołek okna.

Taka organizacja układu współrzędnych różni się od tego, który pamiętamy z lekcji matematyki w szkole. Wprawdzie w poziomie wartości zwiększają się od lewej do prawej, ale za to w pionie zwiększają się od góry do dołu. Standardowy zapis współrzędnych wykorzystuje najpierw współrzędną poziomą, po której podaje się współrzędną pionową. W niektórych sytuacjach współrzędna pozioma jest określana jako współrzędna x, a pionowa jako y. W innych sytuacjach współrzędną poziomą określa się jako left (co można tłumaczyć jako „od lewej"), a pionową jako top (czyli „od góry").

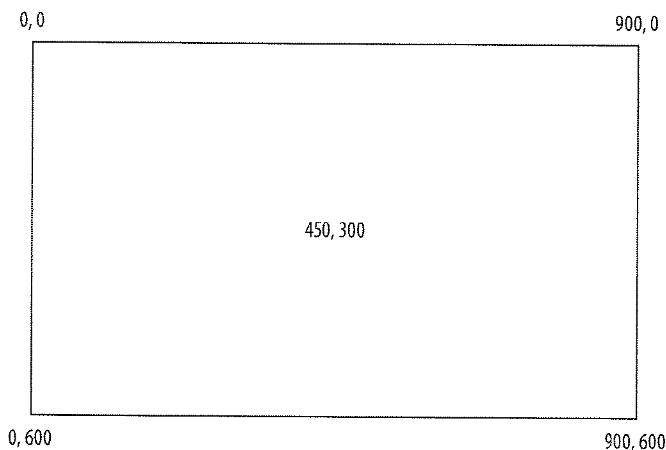
Na rysunku 2.5 przedstawiamy układ współrzędnych okna przeglądarki o wymiarach 900 (szerokość) na 600 (wysokość). Liczby określają wartości współrzędnych narożników oraz środka ekranu.

Teraz przyjrzymy się kilku instrukcjom rysowania, a następnie połączymy je w celu narysowania prostych kształtów (rysunki od 2.6 do 2.10). W dalszej części rozdziału zastanowimy się, w jaki sposób najlepiej będzie rysować ściany kostki i oczka określające liczbę wyrzuconych punktów.

Wywołanie JavaScriptu rysujące prostokąt ma następującą postać:

```
ctx.strokeRect(100, 50, 200, 300);
```

To wywołanie narysuje niewypełniony prostokąt z wierzchołkiem o współrzędnych 100 od lewej, 50 od góry okna o szerokości 200 i wysokości 300 pikseli. To wyrażenie wykorzysta aktualne ustawienia grubości, kształtu i koloru linii.



Rysunek 2.5. Współrzędne w oknie przeglądarki

Kolejny listing demonstruje sposób definiowania grubości linii (ustawimy ją na 5 pikseli) oraz jej koloru (czerwony). Prostokąt zostanie narysowany z tymi ustawieniami linii, ze współrzędnymi oraz długością i szerokością określonymi w zmiennych `x`, `y`, `w` i `h`.

```
ctx.lineWidth = 5;
ctx.strokeStyle = "rgb(255, 0, 0)";
ctx.strokeRect(x, y, w, h);
```

Poniższy listing spowoduje narysowanie niebieskiego prostokąta o zadanych współrzędnych i wymiarach:

```
ctx.fillStyle = "rgb(0, 0, 255)";
ctx.fillRect(x, y, w, h);
```

Jeśli chcesz narysować niebieski prostokąt z czerwoną ramką, możesz wykorzystać następujące dwa wiersze kodu:

```
ctx.fillRect(x, y, w, h);
ctx.strokeRect(x, y, w, h);
```

HTML5 pozwala rysować tak zwane ścieżki (ang. *path*), składające się z łuków i odcinków. Odcinki są rysowane za pomocą kombinacji wywołań `ctx.moveTo()` i `ctx.lineTo()`. Technikę tę będziemy wykorzystywać w kilku kolejnych rozdziałach: w rozdziale 4. przy okazji strzelania z procy, w rozdziale 5. przy grze w zapamiętywanie kształtów oraz w rozdziale 9. przy grze w wisielca. W rozdziale 4. zademonstruję, w jaki sposób można przechylić prostokąt, w rozdziale 9. pokażę, jak rysować kształty owalne. W tym rozdziale skupimy się natomiast na łukach.

Na początku ścieżki należy wywołać następującą metodę:

```
ctx.beginPath();
```

Na końcu ścieżki należy zasygnalizować jej zamknięcie, a następnie wywołać rysowanie ścieżki:

```
ctx.closePath();
ctx.stroke();
```

Można też wypełnić ścieżkę:

```
ctx.closePath();
ctx.fill();
```

Łuk może być pełnym okręgiem lub jego częścią. W przypadku gry w kości chcemy rysować pełne okręgi, reprezentujące kropki na ścianie kostki (oczka), ale najpierw wytłumaczę ogólną zasadę działania łuków, co pozwoli nieco rozjaśnić działanie kodu. Metoda rysująca łuki ma następujący format wywołania:

```
ctx.arc(cx, cy, promień, kąt_początkowy, kąt_końcowy, kierunek);
```

gdzie `cx` i `cy` to współrzędne środka łuku, promień określa promień okręgu (jeśli rysujemy niepełny okrąg, łuk będzie fragmentem okręgu o zadanym promieniu). Kolejne dwa parametry wymagają nieco wprowadzenia z zakresu miar łukowych. Ze szkoły pamiętamy zasadę mierzenia kątów w stopniach: mówimy o obrocie o 180 stopni (gdy ktoś „zawróci"), wiemy, że kąt 90 stopni powstaje wówczas, gdy dwie przecinające się linie są wzajemnie prostopadłe. Większość języków programowania wykorzystuje jednak miary łukowe w radianach. Kąt ma wartość jednego radiana, gdy weźmiemy promień okręgu i odmierzymy ją po jego obwodzie. Gdy spróbujesz odświeżyć sobie trochę wiedzy z matematyki, może przypomnisz sobie, że obwód okręgu wynosi $2 \cdot \pi$ **radianów**, czyli nieco więcej niż 6. Jeśli zatem chcemy narysować pełny okrąg, musimy określić `kąt_początkowy` jako 0, a `kąt_końcowy` jako $2 \cdot \pi$. Na szczęście, nie musimy pamiętać ani wpisywać wartości π (która, jak pamiętamy, jest nieskończona), ponieważ moduł `Math` definiuje zmienną `PI` (o takiej dokładności, aby obliczenia były wystarczająco precyzyjne). W naszym kodzie JavaScript zastosujemy wyrażenie `2 * Math.PI`. Jeśli potrzebujemy półokręgu, zastosujemy `Math.PI`, natomiast ćwiartka okręgu (90 stopni) to wartość `0.5 * Math.PI`.

Metoda `arc()` wymaga jeszcze jednego argumentu — kierunku. W jaki sposób rysuje się łuki? Wyobraź sobie rysowanie okręgu cyrklem. W HTML 5 rysowanie w kierunku zgodnym z ruchem wskazówek zegara określa się jako `false`, natomiast w przeciwnym jako `true` (nie pytaj dlaczego, po prostu tak to zostało zdefiniowane w specyfikacji standardu HTML5). Kierunek ma znaczenie w przypadku, gdy chcemy narysować fragment okręgu. Sposób definiowania parametrów metody `arc()` w przypadku rysowania fragmentów okręgu jest zależny od konkretnego zadania.

Poniższe listingi prezentują kilka przykładów użycia metody `arc()`. Proponuję, abyś przepisał je samodzielnie (za pomocą programu TextPad albo TextWrangler), dzięki czemu lepiej je zrozumiesz. Pierwszy z nich rysuje łuk, przypominający uproszczony uśmiech:

```
<html>
  <head>
    <title>Uśmiech</title>
    <script type="text/javascript">
    function init() {
      var ctx =document.getElementById("canvas").getContext('2d');
      ctx.beginPath();
      ctx.strokeStyle = "rgb(200,0,0)";
      ctx.arc(200, 200, 50, 0, Math.PI, false);
      ctx.stroke();
    }
    </script>
```

```
  </head>
  <body onLoad="init();">
    <canvas id="canvas" width="400" height="300">
    Twoja przeglądarka nie obsługuje elementu canvas standardu HTML5.
    </canvas>
  </body>
</html>
```

Wynik tego kodu prezentuje rysunek 2.6.

Rysunek 2.6. „Uśmiech" narysowany z użyciem wyrażenia ctx.arc(200, 200, 50, 0, Math.PI, false);

Spróbuj zmodyfikować wartości parametrów metody `arc()`, żeby lepiej zapoznać się z jej działaniem oraz mechanizmem układu współrzędnych w celu wyczucia rozmiaru jednego piksela na ekranie.

Zanim narysujemy smutną minę, spróbuj zmienić szerokość lub wysokość uśmiechu, nadaj mu też kolor. Spróbuj przesunąć cały rysunek w górę, w dół, w lewo lub w prawo. Wskazówka: parametry niezbędne do realizacji tego ostatniego polecenia znajdziesz w tym jednym wywołaniu:

```
ctx.arc(200, 200, 50, 0, Math.PI, false);
```

Zmiana fragmentu 200, 200 spowoduje przesunięcie środka okręgu, natomiast zmiana wartości 50 zmodyfikuje jego promień.

Przejdźmy teraz do modyfikacji naszego rysunku. Każdą z nich powinieneś wprowadzić w swoim komputerze i wykonać na niej eksperymenty. Zmiana ostatniego parametru metody `arc` na `true` spowoduje, że łuk zostanie narysowany w kierunku zgodnym z ruchem wskazówek zegara:

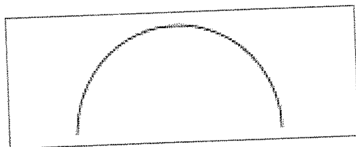
```
ctx.arc(200, 200, 50, 0, Math.PI, true);
```

Oto kompletny kod przykładu:

```
<html>
  <head>
    <title>Smutek</title>
    <script type="text/javascript">
    function init() {
      var ctx =document.getElementById("canvas").getContext('2d');
      ctx.beginPath();
      ctx.strokeStyle = "rgb(200, 0, 0)";
      ctx.arc(200, 200, 50, 0, Math.PI, true);
      ctx.stroke();
    }
    </script>
  </head>
  <body onLoad="init();">
    <canvas id="canvas" width="400" height="300">
```

```
    Twoja przeglądarka nie obsługuje elementu canvas standardu HTML5.
    </canvas>
  </body>
</html>
```

Zmieniłam też tytuł dokumentu na bardziej właściwy. Wynik działania tego kodu prezentuje rysunek 2.7.



Rysunek 2.7. „Smutna mina" narysowana za pomocą wywołania ctx.arc(200, 200, 50, 0, Math.PI, true);

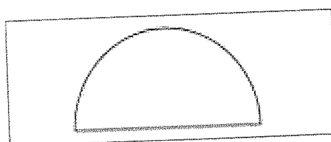
Spróbujmy zamknąć ścieżkę teraz, przed wywołaniem metody `stroke()`:

```
ctx.closePath();
ctx.stroke();
```

Spowoduje to, że łuk zostanie „domknięty" odcinkiem w kształcie cięciwy. Oto pełny kod przykładu:

```
<html>
  <head>
    <title>Smutek</title>
    <script type="text/javascript">
    function init() {
      var ctx =document.getElementById("canvas").getContext('2d');
      ctx.beginPath();
      ctx.strokeStyle = "rgb(200, 0, 0)";
      ctx.arc(200, 200, 50, 0, Math.PI, true);
      ctx.closePath();
      ctx.stroke();
    }
    </script>
  </head>
  <body onLoad="init();">
    <canvas id="canvas" width="400" height="300">
    Twoja przeglądarka nie obsługuje elementu canvas standardu HTML5.
    </canvas>
  </body>
</html>
```

Wynik tego kodu prezentuje rysunek 2.8.



Rysunek 2.8. Smutna mina staje się półkolem dzięki wywołaniu ctx.closePath(); przed wywołaniem ctx.stroke();

Wywołanie closePath()nie zawsze jest niezbędne, ale jego użycie należy do zalecanych praktyk. Poeksperymentuj z tym kodem, więcej przykładów użycia ścieżek znajdziesz natomiast w rozdziale 5. przy okazji rysowania procy i w rozdziale 9. w grze w wisielca. Jeśli chcesz wypełnić ścieżkę, zamiast ctx.stroke() użyj ctx.fill(), co spowoduje, że ścieżka zostanie wypełniona czarnym kolorem, jak to prezentuje rysunek 2.9. Oto kompletny kod przykładu:

```
<html>
 <head>
  <title>Uśmiech</title>
  <script type="text/javascript">
  function init() {
    var ctx =document.getElementById("canvas").getContext('2d');
    ctx. beginPath();
    ctx.strokeStyle = "rgb(200, 0, 0)";
    ctx.arc(200, 200, 50, 0, Math.PI, false);
    ctx. closePath();
    ctx.fill();
  }
  </script>
 </head>
 <body onLoad="init();">
  <canvas id="canvas" width="400" height="300">
  Twoja przeglądarka nie obsługuje elementu canvas standardu HTML5.
  </canvas>
 </body>
</html>
```

Czarny jest kolorem domyślnym.

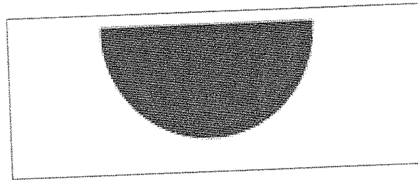
Rysunek 2.9. Wypełnienie ścieżki półokręgu z użyciem metody ctx.fill()

Jeśli ścieżka ma być wypełniona innym kolorem i obrysowana innym, należy użyć metody fill() oraz metody stroke(), a wywołania te poprzedzić odpowiednio wywołaniami metod fillStyle() i strokeStyle(). Definicja koloru wykorzystuje ten sam schemat kombinacji czerwonego, zielonego i niebieskiego, który poznaliśmy w rozdziale 1. Kolor można dobrać metodą eksperymentalną, można też dobrać kolor za pomocą programu graficznego. Oto kompletny kod przykładu:

```
<html>
 <head>
  <title>Uśmiech</title>
  <script type="text/javascript">
  function init() {
    var ctx =document.getElementById("canvas").getContext('2d');
    ctx.beginPath();
    ctx.strokeStyle = "rgb(200, 0, 0)";
```

```
      ctx.arc(200, 200, 50, 0, Math.PI, false);
      ctx.fillStyle = "rgb(200, 0, 200)";
      ctx.closePath();
      ctx.fill();
      ctx.strokeStyle="rgb(255, 0, 0)";
      ctx.lineWidth=5;
      ctx.stroke();
    }
    </script>
  </head>
  <body onLoad="init();">
    <canvas id="canvas" width="400" height="300">
    Twoja przeglądarka nie obsługuje elementu canvas standardu HTML5.
    </canvas>
  </body>
</html>
```

Ten kod w efekcie daje półkole wypełnione kolorem fioletowym (kombinacja składowych czerwonej i niebieskiej) z czerwoną obwódką, co prezentuje rysunek 2.10. Kod definiuje ścieżkę, następnie wypełnia ją, a na końcu rysuje obwódkę.



Rysunek 2.10. Użycie metod fill() i stroke() do uzyskania różnych kolorów wypełnienia i obwódki

Do narysowania pełnego koła możemy użyć różnych kombinacji parametrów, na przykład:

```
ctx.arc(200, 200, 50, 0, 2 * Math.PI, true);
ctx.arc(200, 200, 50, 0, 2 * Math.PI, false);
ctx.arc(200, 200, 50, .5 * Math.PI, 2.5 * Math.PI, false);
```

Warto jednak podkreślić, że pierwsze z tych wywołań da dokładnie taki sam efekt, jak pozostałe. Warto zwrócić uwagę na to, że w celu wypełnienia okręgu nadal należy używać metody `closePath()`. Okrąg jest zamkniętą figurą z punktu widzenia geometrii, ale ten fakt nie ma znaczenia w języku JavaScript.

Jeśli element `canvas` potraktujesz jak płótno, na którym malujesz obraz, zorientujesz się, że w celu namalowania nowego obrazu musisz wymazać starą zawartość. Do tego celu w HTML 5 służy odpowiednia metoda:

```
ctx.clearRect(x, y, szerokość, wysokość);
```

Przykłady użycia tej metody znajdziesz również w dalszych rozdziałach: w rozdziale 4. przy okazji rysowania procy, w rozdziale 5. przy rysowaniu wielokątów na potrzeby gry w zapamiętywanie, w rozdziale 7. przy okazji rysowania labiryntu oraz w rozdziale 9. przy okazji gry w wisielca. Teraz jednak wróćmy do naszej gry w kości.

Wyświetlanie tekstu w polach formularza

Element canvas pozwala również na wyświetlanie tekstu (co zobaczymy w rozdziale 5.), ale na potrzeby gry w kości wymyśliłam sposób polegający na wyświetlaniu informacji w tekstowych polach formularza, dostępnych we wszystkich wersjach standardu HTML. Formularz nie jest tutaj wykorzystany do przyjmowania danych od użytkownika, a jedynie do prezentowania użytkownikowi komunikatów na temat stanu gry w kości. Specyfikacja HTML 5 dodaje wiele nowych funkcji obsługi formularzy, jak **walidacja** typu i zakresu wprowadzonych danych. Tymi nowinkami zajmiemy się przy okazji aplikacji prezentowanej w kolejnym rozdziale.

Do prezentowania wyników gry wykorzystamy następujący kod HTML:

```
<form name="f">
  Etap: <input name="stage" value="Pierwszy rzut"/>
  Punkty: <input name="pv" value=""/>
  Wynik: <input name="outcome" value=""/>
</form>
```

Element form zawiera atrybut name (nazwa). Obok pól tekstowych wyświetlone są etykiety *Etap:*, *Punkty:* i *Wynik:*. Pola input (są to znaczniki pojedyncze) posiadają atrybuty name i value. Te atrybuty będą wykorzystane w kodzie JavaScript. W elemencie form można umieszczać dowolny kod HTML, a formularze można zagnieżdżać w dowolnym innym kodzie.

Gra w kości wykorzystuje element button (przycisk), do formularza dodamy również element tego typu, natomiast formularz nie będzie posiadał elementu typu submit (przycisk wysyłania formularza). Można również użyć elementu typu submit (rezygnując z elementu button). Wówczas wykorzystamy taki kod:

```
<form name="f" onSubmit="throwdice();">
  Etap: <input type="text" name="stage" value="Pierwszy rzut"/>
  Punkty: <input type="text" name="pv" value=""/>
  Wynik: <input type="text" name="outcome" value=""/>
  <input type="submit" value="Rzuć kośćmi"/>
</form>
```

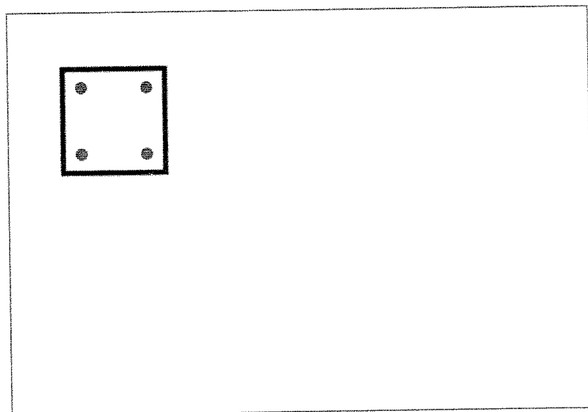
Element submit również generuje przycisk na ekranie. Dzięki temu otrzymujemy wszystkie elementy niezbędne do obsługi gry w kości. Zostało nam tylko napisać nasz kod.

Zbuduj własną aplikację

Zapewne masz już niewielką wprawę w wykorzystaniu standardów HTML5, CSS i JavaScriptu do tworzenia prostych wprawek, jak przykłady z pierwszej części tego rozdziału. Jeśli nie, mocno do tego zachęcam. Jedyny sposób na dogłębne nauczenie się czegokolwiek polega na wykorzystaniu tej wiedzy we własnych projektach. W ramach etapowej implementacji gry w kości napiszemy jej 3 wersje:

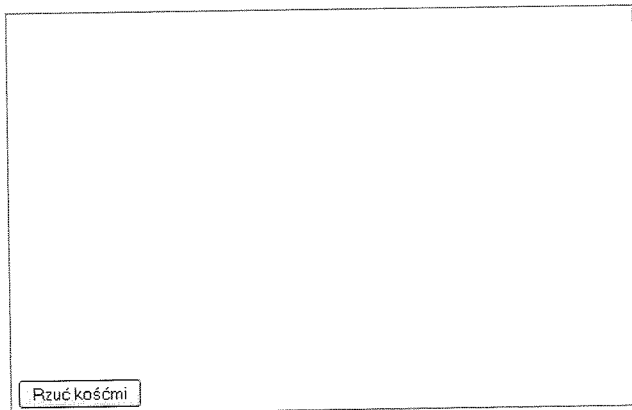
- pojedynczy rzut kostką i narysowanie wyniku; kolejny rzut kostką polega na odświeżeniu strony;
- rzucanie dwiema kostkami z użyciem przycisku formularza;
- kompletna gra w kości z obsługą reguł gry craps.

Rysunek 2.11 prezentuje możliwy wygląd ekranu pierwszej z aplikacji. Możliwy, ponieważ nie zawsze wynikiem będzie 4. W tym przypadku zrzut ekranu obejmuje duży obszar okna, dzięki czemu zorientujesz się, w którym miejscu rysowana jest kostka.



Rysunek 2.11. Rzut jedną kostką

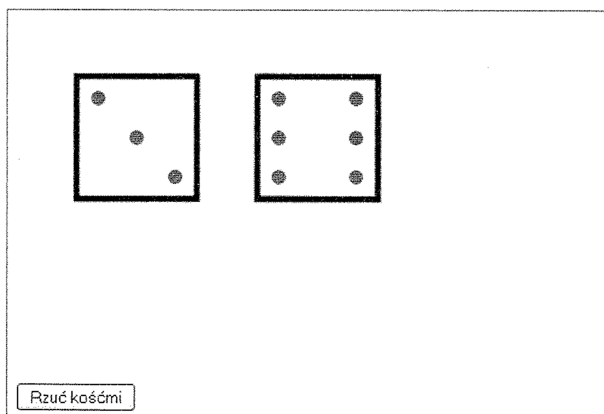
Rysunek 2.12 przedstawia początkowy ekran aplikacji polegającej na rzucie dwiema kostkami. W tym przypadku widać tylko przycisk.



Rysunek 2.12. Początkowy ekran aplikacji obsługującej rzut dwiema kostkami

Wreszcie, rysunek 2.13 przedstawia ekran drugiej aplikacji po kliknięciu przycisku.

Etapowe budowanie aplikacji stanowi bardzo dobre podejście do tego typu problemów. Nasze aplikacje piszemy w edytorze tekstu, jak TextPad czy TextWrangler. Pamiętaj o tym, aby zapisać dokument jako typ *.html*. Zapisuj często, nie ma konieczności zapisywania po skończeniu wprowadzania całej treści. Po wprowadzeniu kodu pierwszej aplikacji i zapisaniu jej możesz zapisać ją ponownie, pod inną nazwą. Wówczas wystarczy w tej kopii wprowadzić niezbędne modyfikacje. Tak samo możesz postąpić z trzecią aplikacją.



Rysunek 2.13. Kliknięcie przycisku powoduje narysowanie wyniku rzutu kośćmi

Rzut pojedynczą kością

Celem pierwszej aplikacji jest wyświetlenie na elemencie `canvas` wyniku rzutu pojedynczą kością do gry, z kropkami (oczkami) uporządkowanymi tak jak na standardowej kostce.

Jak w przypadku większości aplikacji, do dyspozycji mamy kilka sposobów realizacji tego zadania. Pisząc ten kod, zorientowałam się, że niektóre z fragmentów kodu mogę wykorzystać wielokrotnie, ponieważ oczka wyniku 3 stanowią kombinację oczek wyników 2 i 1. Podobnie układ oczek wyniku 5 stanowi kombinację układów 4 i 1. Układ wyniku 6 stanowi natomiast kombinację układu wyniku 4 oraz dodatkowego, unikalnego układu. Cały kod mogłam umieścić w funkcji `init()`, mogłam też użyć dodatkowej funkcji `drawface()`. Po rozpoznaniu problemu zaimplementowanie kodu i usunięcie błędów zajęło mi stosunkowo mało czasu. Tabela 2.1 zawiera listę funkcji użytych w aplikacji z informacją o tym, przez co jest wywoływana dana funkcja i co sama wywołuje. Tabela 2.2 przedstawia kompletny kod z objaśnieniami.

Tabela 2.1. Funkcje zdefiniowane w aplikacji „Rzut pojedynczą kością"

Funkcja	**Wywoływana przez**	**Wywołuje**
`init()`	zdarzenie `onLoad` elementu `body`	`drawface()`
`drawface()`	`init()`	`draw1()`, `draw2()`, `draw4()`, `draw6()`, `draw2mid()`
`draw1()`	`drawface()` do narysowania jedynki, trójki i piątki	
`draw2()`	`drawface()` do narysowania dwójki i trójki	
`draw4()`	`drawface()` do narysowania czwórki, piątki i szóstki	
`draw2mid()`	`drawface()` do narysowania szóstki	

Tabela 2.2. Pełny kod aplikacji „Rzut pojedynczą kością"

Kod	**Omówienie**
<html>	Początek elementu html
<head>	Początek elementu head
<title>Rzut pojedynczą kością</title>	Kompletny element title
<script>	Początek elementu script
var cwidth = 400;	Zmienna przechowująca szerokość elementu canvas, wykorzystywana do wymazywania zawartości elementu przed ponownym rysowaniem
var cheight = 300;	Zmienna przechowująca wysokość elementu canvas, wykorzystywana do wymazywania zawartości elementu przed ponownym rysowaniem
var dicex = 50;	Zmienna przechowująca poziome położenie rysunku kostki w elemencie canvas
var dicey = 50;	Zmienna przechowująca pionową pozycję kostki w elemencie canvas
var dicewidth = 100;	Zmienna przechowująca szerokość rysunku kostki
var diceheight = 100;	Zmienna przechowująca wysokość rysunku kostki
var dotrad = 6;	Zmienna przechowująca promień rysunku oczka kostki
var ctx;	Zmienna przechowująca kontekst elementu canvas, używany w instrukcjach rysowania
function init() {	Początek definicji funkcji init(), która jest wywoływana w ramach obsługi zdarzenia onLoad dokumentu
var ch = 1 + Math.floor(Math.random() * 6);	Deklaracja zmiennej ch i ustawienie jej wartości na liczbę losową ze zbioru 1, 2, 3, 4, 5, 6
drawface(ch);	Wywołanie funkcji drawface() z parametrem ch
}	Koniec definicji funkcji init()
function drawface(n) {	Początek definicji funkcji drawface(), która przyjmuje parametr określający liczbę oczek do narysowania
ctx = document.getElementById('canvas').getContext('2d');	Uzyskanie dostępu do obiektu używanego do rysowania na elemencie canvas
ctx.lineWidth = 5;	Ustawienie szerokości linii na 5 pikseli
ctx.clearRect(dicex, dicey, dicewidth, diceheight);	Wyczyszczenie obszaru, w którym będzie rysowana kostka. Przy pierwszym wywołaniu nie powoduje żadnego efektu
ctx.strokeRect(dicex, dicey, dicewidth, diceheight)	Rysowanie obrysu kostki
ctx.fillStyle = "#009966";	Ustawienie koloru kropek. Ja użyłam programu graficznego do ustalenia koloru; można też poeksperymentować z wartościami szesnastkowymi
switch(n) {	Początek instrukcji switch analizującej liczbę kropek

Kod	Omówienie
`case 1:`	Jeśli wartość wynosi 1
`draw1();`	Wywołanie funkcji `draw1()`
`break;`	Wyjście z instrukcji `switch`
`case 2:`	Jeśli wartość wynosi 2
`draw2();`	Wywołanie funkcji `draw2()`
`break;`	Wyjście z instrukcji `switch`
`case 3:`	Jeśli wartość wynosi 3
`draw2();`	Wywołanie funkcji `draw2()`, a następnie
`draw1();`	… funkcji `draw1()`
`break;`	Wyjście z instrukcji `switch`
`case 4:`	Jeśli wartość wynosi 4
`draw4();`	Wywołanie funkcji `draw4()`
`break;`	Wyjście z instrukcji `switch`
`case 5:`	Jeśli wartość wynosi 5
`draw4();`	Wywołanie funkcji `draw4()`, a następnie
`draw1();`	… funkcji `draw1()`
`break;`	Wyjście z instrukcji `switch`
`case 6:`	Jeśli wartość wynosi 6
`draw4();`	Wywołanie funkcji `draw4()`, a następnie
`draw2mid();`	… funkcji `draw2mid()`
`break;`	Wyjście z instrukcji `switch` (w tym miejscu nie jest wymagane)
`}`	Koniec instrukcji `switch`
`}`	Koniec definicji funkcji `drawface()`
`function draw1() {`	Początek definicji funkcji `draw1()`
`var dotx;`	Zmienna przechowująca poziomą pozycję kropki
`var doty;`	Zmienna przechowująca pionową pozycję kropki
`ctx.beginPath();`	Początek ścieżki
`dotx = dicex + .5 * dicewidth;`	Środek tego oczka jest wycentrowany w stosunku do rysunku ściany kostki poziomo…
`doty = dicey + .5 * diceheight;`	… i pionowo
`ctx.arc(dotx, doty, dotrad, 0, Math. PI * 2, true);`	Rysowanie okręgu (jego wypełnienie następuje po wywołaniu funkcji `fill()`)
`ctx.closePath();`	Zamknięcie ścieżki
`ctx.fill();`	Wypełnienie narysowanego okręgu
`}`	Koniec definicji funkcji `draw1()`

Kod	Omówienie
`function draw2() {`	Początek definicji funkcji `draw2()`
`var dotx;`	Zmienna przechowująca pozycję poziomą dla dwóch oczek
`var doty;`	Zmienna przechowująca pozycję pionową dla dwóch oczek
`ctx.beginPath();`	Początek ścieżki
`dotx = dicex + 3 * dotrad;`	Środek tego oczka znajduje się w odległości trzech promieni oczka od lewego górnego wierzchołka ściany kostki poziomo…
`doty = dicey + 3 * dotrad;`	… i pionowo
`ctx.arc(dotx, doty, dotrad, 0, Math. PI * 2, true);`	Rysowanie pierwszej kropki
`dotx = dicex + dicewidth - 3 * dotrad;`	Środek tego oczka znajduje się w odległości trzech promieni oczka od prawego dolnego wierzchołka ściany kostki poziomo…
`doty = dicey + diceheight - 3 * dotrad;`	… i pionowo
`ctx.arc(dotx, doty, dotrad, 0, Math. PI * 2, true);`	Rysowanie drugiej kropki
`ctx.closePath();`	Zamknięcie ścieżki
`ctx.fill();`	Rysowanie obydwu kropek
`}`	Koniec definicji funkcji `draw2()`
`function draw4() {`	Początek definicji funkcji `draw4()`
`var dotx;`	Zmienna przechowująca poziomą pozycję oczek
`var doty;`	Zmienna przechowująca pionową pozycję oczek
`ctx.beginPath();`	Początek ścieżki
`dotx = dicex + 3 * dotrad;`	Środek pierwszego oczka znajduje się w odległości trzech promieni oczka od lewego górnego wierzchołka ściany kostki poziomo…
`doty = dicey + 3*dotrad;`	… i pionowo
`ctx.arc(dotx, doty, dotrad, 0, Math. PI * 2, true);`	Rysowanie okręgu
`dotx = dicex + dicewidth - 3 * dotrad;`	Środek drugiego oczka znajduje się w odległości trzech promieni oczka od prawego dolnego wierzchołka ściany kostki poziomo…
`doty = dicey + diceheight - 3 * dotrad;`	… i pionowo
`ctx.arc(dotx, doty, dotrad, 0, Math. PI * 2, true);`	Rysowanie kropek
`ctx.closePath();`	Zamknięcie ścieżki
`ctx.fill();`	Rysowanie 2 kropek
`ctx.beginPath();`	Początek ścieżki

Kod	Omówienie
dotx = dicex + 3 * dotrad;	Środek trzeciego oczka znajduje się w odległości trzech promieni oczka od lewego dolnego wierzchołka ściany kostki poziomo…
doty = dicey + diceheight - 3 * dotrad;	… i pionowo (to ta sama wartość y, która była użyta poprzednio)
ctx.arc(dotx, doty, dotrad, 0, Math. PI * 2, true);	Rysowanie okręgu
dotx = dicex + dicewidth - 3 * dotrad;	Środek pierwszego oczka znajduje się w odległości trzech promieni oczka od prawego górnego wierzchołka ściany kostki poziomo…
doty = dicey + 3 * dotrad;	… i pionowo
ctx.arc(dotx, doty, dotrad, 0, Math. PI * 2, true);	Rysowanie okręgu
ctx.closePath();	Zamknięcie ścieżki
ctx.fill();	Rysowanie 2 kropek
}	Koniec definicji funkcji draw4()
function draw2mid() {	Początek definicji funkcji draw2mid()
var dotx;	Zmienna przechowująca poziomą pozycję oczek
var doty;	Zmienna przechowująca pionową pozycję oczek
ctx.beginPath();	Początek ścieżki
dotx = dicex + 3 * dotrad;	Pozioma pozycja pierwszego oczka znajduje się w odległości trzech promieni oczka od lewej krawędzi ściany kostki poziomo…
doty = dicey + .5 * diceheight;	… i jest wyśrodkowana pionowo
ctx.arc(dotx, doty, dotrad, 0, Math.PI * 2, true);	Rysowanie okręgu
dotx = dicex + dicewidth - 3 * dotrad;	Pozioma pozycja drugiego oczka znajduje się w odległości trzech promieni oczka od prawej krawędzi ściany kostki poziomo…
doty = dicey + .5 * diceheight; *//brak zmiany*	… i jest wyśrodkowana pionowo
ctx.arc(dotx, doty, dotrad, 0, Math.PI * 2, true);	Rysowanie okręgu
ctx.closePath();	Zamknięcie ścieżki
ctx.fill();	Rysowanie kropek
}	Koniec definicji funkcji draw2mid()
</script>	Koniec elementu script
</head>	Koniec elementu head
<body onLoad="init();">	Początek elementu body, z atrybutem onLoad wymuszającym wywołanie funkcji init()

Kod	Omówienie
`<canvas id="canvas" width="400" height="300">`	Początek elementu canvas
`Twoja przeglądarka nie obsługuje elementu canvas standardu HTML5.`	Komunikat wyświetlany użytkownikom przeglądarek nieobsługujących elementu canvas
`</canvas>`	Koniec elementu canvas
`</body>`	Koniec elementu body
`</html>`	Koniec elementu html

W kodzie warto umieszczać komentarze. Komentarze to informacje tekstowe, które są pomijane przez interpreter JavaScriptu, ale służą jako przypomnienie dla programisty lub informacja dla innych osób czytających kod. Jedna z dopuszczalnych form komentarzy polega na wpisaniu dwóch ukośników // przed tekstem komentarza. Cały tekst od tych znaków do końca wiersza jest ignorowany przez przeglądarkę. W przypadku komentarzy rozciągających się przez wiele wierszy początek komentarza oznaczany jest sekwencją znaków ukośnik i gwiazdka /*, natomiast koniec komentarza przez sekwencję gwiazdka ukośnik */:

```
/*
To jest komentarz
*/
```

Komentarze są jednym z elementów kodu, których nie stosowałam w przykładach. Ponieważ kody są prezentowane w wąskiej kolumnie tabeli, komentarze zmniejszałyby ich czytelność, poza tym cały kod jest szczegółowo udokumentowany w prawej kolumnie. Tym niemniej Ty powinieneś stosować komentarze w pisanym przez siebie kodzie.

Pisząc ten kod (lub dowolny kod wykorzystujący zdarzenia losowe), nie chciałam losować wszystkich możliwych kombinacji rzutów kostką w celu przetestowania, czy aplikacja zachowuje się prawidłowo dla każdej z nich. Aby tego uniknąć, po wierszu

```
var ch = 1 + Math.floor(Math.random() * 6);
```

dopisałam wartość, którą chciałam przetestować, np:

```
ch = 1;
```

Po zakończeniu testu zamieniałam tę wartość na kolejną:

```
ch = 2;
```

Po zakończeniu testów usunęłam te wiersze (można też zamienić je na komentarz za pomocą znaków //). Tego typu praktyki są szczególnie użyteczne w przypadku testowania rozbudowanych gier, ponieważ dzięki nim można uniknąć konieczności spędzenia ogromnej ilości czasu na testowaniu sposobem wielogodzinnego grania w grę.

Rzut dwiema kośćmi

Kolejna aplikacja wykorzystuje element przycisku `button`, za pomocą którego gracz może wykonać „rzut dwiema kośćmi", zamiast przeładowywać stronę. Zanim zaczniemy analizować kod, należy zastanowić się, które elementy pierwszej aplikacji nadają się do ponownego użycia. Jak się okazuje

— większość. Druga aplikacja musi obsłużyć pozycjonowanie dwóch rysunków kostek; do tego celu wykorzystamy dwie zmienne, dx i dy. Kod losujący wykorzystujący Math.random() musi wystąpić dwukrotnie, podobnie funkcja drawface() rysująca kostki. Musimy też zmodyfikować miejsce, w którym wywoływana jest funkcja wykonująca „rzut kośćmi". Tabela 2.3 zawiera zestawienie funkcji wykorzystanych w drugiej aplikacji z informacją, przez co są wywoływane i co wywołują. Tabela ta jest bardzo podobna do tabeli 2.1, z tą różnicą, że tu mamy dodatkową funkcję pod nazwą throwdice(), wywoływaną przez zdarzenie onClick elementu button. Tabela 2.4 zawiera pełny kod HTML dokumentu aplikacji rzutu dwiema kośćmi.

Tabela 2.3. Funkcje w aplikacji „Rzut dwiema kośćmi"

Funkcja	**Wywoływana przez**	**Wywołuje**
throwdice()	obsługę zdarzenia onClick elementu button	drawface()
drawface()	init()	draw1(), draw2(), draw4(), draw6(), draw2mid()
draw1()	drawface() do narysowania jedynki, trójki i piątki	
draw2()	drawface() do narysowania dwójki i trójki	
draw4()	drawface() do narysowania czwórki, piątki i szóstki	
draw2mid()	drawface() do narysowania szóstki	

Tabela 2.4. Pełny kod aplikacji „Rzut dwiema kośćmi"

Kod	**Omówienie**
<html>	Początek elementu html
<head>	Początek elementu head
<title>Rzut dwiema kośćmi</title>	Element title z zawartością
<script>	Początek elementu script
var cwidth = 400;	Zmienna przechowująca szerokość elementu canvas
var cheight = 300;	Zmienna przechowująca wysokość elementu canvas, wykorzystywana do wymazywania zawartości elementu przed ponownym rysowaniem
var dicex = 50;	Zmienna przechowująca poziome położenie rysunku kostki w elemencie canvas
var dicey = 50;	Zmienna przechowująca pionową pozycję kostki w elemencie canvas
var dicewidth = 100;	Zmienna przechowująca szerokość rysunku kostki
var diceheight = 100;	Zmienna przechowująca wysokość rysunku kostki
var dotrad = 6;	Zmienna przechowująca promień rysunku oczka kostki
var ctx;	Zmienna przechowująca kontekst elementu canvas, używany w instrukcjach rysowania
var dx;	Zmienna przechowująca poziomą pozycję kostki; jej wartość jest modyfikowana w celu narysowania każdej z kostek

Kod	**Omówienie**
`var dy;`	Zmienna przechowująca pionową pozycję, taka sama dla obydwu kostek
`function throwdice() {`	Początek definicji funkcji `throwdice()`
`var ch = 1 + Math.floor(Math.random() * 6);`	Deklaracja zmiennej `ch` i ustawienie jej wartości na losową
`dx = dicex;`	Ustawienie zmiennej `dx` dla pierwszej kostki
`dy = dicey;`	Ustawienie wartości `dy`
`drawface(ch);`	Wywołanie funkcji `drawface()` z wartością `ch` w charakterze liczby oczek
`dx = dicex + 150;`	Modyfikacja wartości `dx` dla drugiej kostki
`ch=1 + Math.floor(Math.random()*6);`	Ustawienie zmiennej `ch` na kolejną wartość losową
`drawface(ch);`	Wywołanie funkcji `drawface()` z wartością `ch` w charakterze liczby oczek
`}`	Koniec definicji funkcji `throwdice()`
`function drawface(n) {`	Początek definicji funkcji `drawface()`, która przyjmuje parametr liczby oczek
`ctx = document.getElementById('canvas').getContext('2d');`	Uzyskanie dostępu do obiektu używanego do rysowania na elemencie `canvas`
`ctx.lineWidth = 5;`	Ustawienie szerokości linii na 5 pikseli
`ctx.clearRect(dx, dy, dicewidth, diceheight);`	Wyczyszczenie obszaru, w którym będzie rysowana kostka. Przy pierwszym wywołaniu nie powoduje żadnego efektu
`ctx.strokeRect(dx, dy, dicewidth, diceheight)`	Rysowanie obrysu kostki
`var dotx;`	Zmienna przechowująca poziomą pozycję oczka
`var doty;`	Zmienna przechowująca pionową pozycję oczka
`ctx.fillStyle = "#009966";`	Ustawienie koloru
`switch(n) {`	Początek bloku instrukcji `switch` analizującego liczbę oczek
`case 1:`	Jeśli wartość wynosi 1
`draw1();`	Wywołanie funkcji `draw1()`
`break;`	Wyjście z instrukcji `switch`
`case 2:`	Jeśli wartość wynosi 2
`draw2();`	Wywołanie funkcji `draw2()`
`break;`	Wyjście z instrukcji `switch`
`case 3:`	Jeśli wartość wynosi 3
`draw2();`	Wywołanie funkcji `draw2()`, a następnie...
`draw1();`	... wywołanie funkcji `draw1()`
`break;`	Wyjście z instrukcji `switch`
`case 4:`	Jeśli wartość wynosi 4
`draw4();`	Wywołanie funkcji `draw4()`

Kod	Omówienie
`break;`	Wyjście z instrukcji `switch`
`case 5:`	Jeśli wartość wynosi 5
`draw4();`	Wywołanie funkcji `draw4()`, a następnie…
`draw1();`	… wywołanie funkcji `draw1()`
`break;`	Wyjście z instrukcji `switch`
`case 6:`	Jeśli wartość wynosi 6
`draw4();`	Wywołanie funkcji `draw4()`, a następnie…
`draw2mid();`	… wywołanie funkcji `draw2mid()`
`break;`	Wyjście z instrukcji `switch` (niewymagane)
`}`	Koniec bloku instrukcji `switch`
`}`	Koniec definicji funkcji `drawface()`
`function draw1() {`	Początek definicji funkcji `draw1()`
`var dotx;`	Zmienna przechowująca poziomą pozycję kropki
`var doty;`	Zmienna przechowująca pionową pozycję kropki
`ctx.beginPath();`	Początek ścieżki
`dotx = dx + .5*dicewidth;`	Środek tego oczka jest wycentrowany w stosunku do rysunku ściany kostki poziomo…
`doty = dy + .5*diceheight;`	… i pionowo
`ctx.arc(dotx, doty, dotrad, 0, Math.PI * 2, true);`	Rysowanie okręgu (jego wypełnienie następuje po wywołaniu funkcji `fill()`)
`ctx.closePath();`	Zamknięcie ścieżki
`ctx.fill();`	Wypełnienie narysowanego okręgu
`}`	Koniec definicji funkcji `draw1()`
`function draw2() {`	Początek definicji funkcji `draw2()`
`var dotx;`	Zmienna przechowująca pozycję poziomą dla dwóch oczek
`var doty;`	Zmienna przechowująca pozycję pionową dla dwóch oczek
`ctx.beginPath();`	Początek ścieżki
`dotx = dx + 3 * dotrad;`	Środek tego oczka znajduje się w odległości trzech promieni oczka od lewego górnego wierzchołka ściany kostki poziomo…
`doty = dy + 3 * dotrad;`	… i pionowo
`ctx.arc(dotx, doty, dotrad, 0, Math.PI * 2, true);`	Rysowanie pierwszej kropki
`dotx = dx+dicewidth – 3 * dotrad;`	Środek tego oczka znajduje się w odległości trzech promieni oczka od prawego dolnego wierzchołka ściany kostki poziomo…
`doty = dy+diceheight-3*dotrad;`	… i pionowo

Kod	Omówienie
`ctx.arc(dotx, doty, dotrad, 0, Math.PI * 2, true);`	Rysowanie drugiej kropki
`ctx.closePath();`	Zamknięcie ścieżki
`ctx.fill();`	Rysowanie obydwu kropek
`}`	Koniec definicji funkcji `draw2()`
`function draw4() {`	Początek definicji funkcji `draw4()`
`var dotx;`	Zmienna przechowująca poziomą pozycję oczek
`var doty;`	Zmienna przechowująca pionową pozycję oczek
`ctx.beginPath();`	Początek ścieżki
`dotx = dx + 3 * dotrad;`	Środek pierwszego oczka znajduje się w odległości trzech promieni oczka od lewego górnego wierzchołka ściany kostki poziomo…
`doty = dy + 3 * dotrad;`	… i pionowo
`ctx.arc(dotx, doty, dotrad, 0, Math.PI * 2, true);`	Rysowanie okręgu
`dotx = dx + dicewidth - 3 * dotrad;`	Środek drugiego oczka znajduje się w odległości trzech promieni oczka od prawego dolnego wierzchołka ściany kostki poziomo…
`doty = dy + diceheight - 3 * dotrad;`	… i pionowo
`ctx.arc(dotx, doty, dotrad, 0, Math.PI * 2, true);`	Rysowanie kropek
`ctx.closePath();`	Zamknięcie ścieżki
`ctx.fill();`	Rysowanie 2 kropek
`ctx.beginPath();`	Początek ścieżki
`dotx = dx + 3 * dotrad;`	Środek trzeciego oczka znajduje się w odległości trzech promieni oczka od lewego dolnego wierzchołka ściany kostki poziomo…
`doty = dy + diceheight - 3 * dotrad;` *//brak zmiany*	… i pionowo (to ta sama wartość y, która była użyta poprzednio)
`ctx.arc(dotx, doty, dotrad, 0, Math.PI * 2, true);`	Rysowanie okręgu
`dotx = dx + dicewidth - 3 * dotrad;`	Środek pierwszego oczka znajduje się w odległości trzech promieni oczka od prawego górnego wierzchołka ściany kostki poziomo…
`doty = dy + 3 * dotrad;`	… i pionowo
`ctx.arc(dotx, doty, dotrad, 0, Math.PI * 2, true);`	Rysowanie okręgu
`ctx.closePath();`	Zamknięcie ścieżki
`ctx.fill();`	Rysowanie 2 kropek
`}`	Koniec definicji funkcji `draw4()`

Kod	Omówienie
function draw2mid() {	Początek definicji funkcji draw2mid()
var dotx;	Zmienna przechowująca poziomą pozycję oczek
var doty;	Zmienna przechowująca pionową pozycję oczek
ctx.beginPath();	Początek ścieżki
dotx = dx + 3 * dotrad;	Pozioma pozycja pierwszego oczka znajduje się w odległości trzech promieni oczka od lewej krawędzi ściany kostki poziomo...
doty = dy + .5 * diceheight;	... i jest wyśrodkowana pionowo
ctx.arc(dotx, doty, dotrad, 0, Math.PI * 2, true);	Rysowanie okręgu
dotx = dx + dicewidth - 3 * dotrad;	Pozioma pozycja drugiego oczka znajduje się w odległości trzech promieni oczka od prawej krawędzi ściany kostki poziomo...
doty = dy + .5 * diceheight; *//brak zmiany*	... i jest wyśrodkowana pionowo
ctx.arc(dotx, doty, dotrad, 0, Math.PI * 2, true);	Rysowanie okręgu
ctx.closePath();	Zamknięcie ścieżki
ctx.fill();	Rysowanie kropek
}	Koniec definicji funkcji draw2mid()
</script>	Koniec elementu script
</head>	Koniec elementu head
<body>	Początek elementu body
<canvas id="canvas" width="400" height="300">	Początek elementu canvas
Twoja przeglądarka nie obsługuje elementu canvas z HTML5	Komunikat wyświetlany użytkownikom przeglądarek nieobsługujących elementu canvas
</canvas>	Koniec elementu canvas
 	Przejście do nowego wiersza
<button onClick="throwdice();">Rzuć kośćmi</button>	Element button (atrybut onClick powoduje wywołanie funkcji throwdice())
</body>	Koniec elementu body
</html>	Koniec elementu html

Kompletna gra w kości

Trzecią aplikacją jest pełna gra w kości według zasad gry craps. Również w tym przypadku dużą część kodu możemy przenieść z poprzedniego etapu. Tym razem musimy dodać obsługę reguł gry. Oznacza to między innymi, że musimy użyć instrukcji warunkowych if i switch oraz zmiennych

globalnych (to znaczy zadeklarowanych poza funkcjami) do obsługi przechowywania stanu gry (runda rzutów, zmienna firstturn) oraz punktów (point). Tabela funkcji jest identyczna z tą dla drugiej aplikacji (tabela 2.3), nie będziemy jej zatem powtarzać. Pełny kod aplikacji znajdziesz w tabeli 2.5. Cała dodatkowa logika została zaimplementowana w funkcji throwdice. Omówione zostaną tylko różnice.

Tabela 2.5. Kompletna gra w kości

Kod	**Omówienie**
<html>	
<head>	
<title>Gra w kości</title>	
<script>	
var cwidth = 400;	
var cheight = 300;	
var dicex = 50;	
var dicey = 50;	
var dicewidth = 100;	
var diceheight = 100;	
var dotrad = 6;	
var ctx;	
var dx;	
var dy;	
var firstturn = true;	Zmienna globalna, zainicjalizowana na wartość true
var point;	Zmienna globalna, nie musi być inicjalizowana, ponieważ jej wartość będzie ustawiona przed użyciem
function throwdice() {	Początek definicji funkcji throwdice()
var sum;	Zmienna przechowująca sumę wartości rzutów dwiema kostkami
var ch = 1 + Math.floor(Math.random() * 6);	Deklaracja zmiennej ch i ustawienie jej wartości na losową
sum = ch;	Przypisanie tej wartości zmiennej sum
dx = dicex;	Ustawienie wartości dx
dy = dicey;	Ustawienie wartości dy
drawface(ch);	Narysowanie pierwszej kostki
dx = dicex + 150;	Modyfikacja wartości dx dla drugiej kostki
ch=1 + Math.floor(Math.random() * 6);	Ustawienie zmiennej ch na kolejną wartość losową dla drugiej kostki
sum += ch;	Aktualizacja wartości sum o wartość ch
drawface(ch);	Narysowanie drugiej kostki
if (firstturn) {	W tym miejscu zaczyna się implementacja reguł Czy to pierwszy rzut?

Kod	Omówienie
`switch(sum) {`	Jeśli tak, sprawdzamy wyrzuconą liczbę kostek
`case 7:`	dla sumy 7…
`case 11:`	… lub 11…
`document.f.outcome.value="Wygrałeś!";`	… wyświetl „Wygrałeś!”
`break;`	Wyjście z warunku `switch`
`case 2:`	Dla sumy 2…
`case 3:`	… lub 3…
`case 12:`	… lub 12…
`document.f.outcome.value="Przegrałeś!";`	… wyświetl „Przegrałeś!”
`break;`	Wyjście z warunku `switch`
`default:`	Dla wszystkich innych przypadków…
`point = sum;`	… zachowaj sumę w zmiennej `point`…
`document.f.pv.value=point;`	… wyświetl liczbę punktów…
`firstturn = false;`	… ustaw wartość `firstturn` na `false`…
`document.f.stage.value="Rzucaj jeszcze raz.";`	… wyświetl informację „Rzucaj jeszcze raz.”…
`document.f.outcome.value="";`	… wyczyść wynik
`}`	Koniec instrukcji `switch`
`}`	Koniec warunku `if true`
`else {`	W przeciwnym wypadku (to nie był pierwszy rzut)
`switch(sum) {`	Początek instrukcji `switch` z użyciem zmiennej `sum`
`case point:`	Jeśli zmienna `sum` ma wartość równą zmiennej `point`
`document.f.outcome.value="Wygrałeś!";`	Wyświetl „Wygrałeś!”…
`document.f.stage.value="Pierwszy rzut.";`	… wyświetl „Pierwszy rzut.”…
`document.f.pv.value="";`	… wyczyść liczbę punktów…
`firstturn = true;`	… ustaw zmienną `firstturn` na wartość `true`
`break;`	Wyjście z bloku `switch`
`case 7:`	Jeśli suma oczek wynosi 7…
`document.f.outcome.value="Przegrałeś!";`	… wyświetl „Przegrałeś!”…
`document.f.stage.value="Pierwszy rzut.";`	… wyświetl „Pierwszy rzut.”…
`document.f.pv.value=" ";`	… wyczyść liczbę punktów…
`firstturn = true;`	… ustaw zmienną `firstturn` na wartość `true`
`}`	Wyjście z bloku `switch`
`}`	Koniec instrukcji `if`
`}`	Koniec funkcji `throwdice()`
`function drawface(n) {`	

Kod	Omówienie
`ctx = document.getElementById('canvas').getContext('2d');`	
`ctx.lineWidth = 5;`	
`ctx.clearRect(dx, dy, dicewidth, diceheight);`	
`ctx.strokeRect(dx, dy, dicewidth, diceheight)`	
`var dotx;`	
`var doty;`	
`ctx.fillStyle = "#009966";`	
`switch(n) {`	
`case 1:`	
`draw1();`	
`break;`	
`case 2:`	
`draw2();`	
`break;`	
`case 3:`	
`draw2();`	
`draw1();`	
`break;`	
`case 4:`	
`draw4();`	
`break;`	
`case 5:`	
`draw4();`	
`draw1();`	
`break;`	
`case 6:`	
`draw4();`	
`draw2mid();`	
`break;`	
`}`	
`}`	
`function draw1() {`	
`var dotx;`	

Kod	Omówienie
var doty;	
ctx.beginPath();	
dotx = dx + .5 * dicewidth;	
doty = dy + .5 * diceheight;	
ctx.arc(dotx, doty, dotrad, 0, Math.PI * 2, true);	
ctx.closePath();	
ctx.fill();	
}	
function draw2() {	
var dotx;	
var doty;	
ctx.beginPath();	
dotx = dx + 3 * dotrad;	
doty = dy + 3 * dotrad;	
ctx.arc(dotx, doty, dotrad, 0, Math.PI * 2, true);	
dotx = dx + dicewidth - 3 * dotrad;	
doty = dy + diceheight - 3 * dotrad;	
ctx.arc(dotx, doty, dotrad, 0, Math.PI * 2, true);	
ctx.closePath();	
ctx.fill();	
}	
function draw4() {	
var dotx;	
var doty;	
ctx.beginPath();	
dotx = dx + 3 * dotrad;	
doty = dy + 3 * dotrad;	
ctx.arc(dotx, doty, dotrad, 0, Math.PI * 2, true);	
dotx = dx + dicewidth - 3 * dotrad;	
doty = dy + diceheight - 3 * dotrad;	
ctx.arc(dotx, doty, dotrad, 0, Math.PI * 2, true);	
ctx.closePath();	

Kod	**Omówienie**
ctx.fill();	
ctx.beginPath();	
dotx = dx + 3*dotrad;	
doty = dy + diceheight - 3 * dotrad; *//brak zmiany*	
ctx.arc(dotx, doty, dotrad, 0, Math.PI * 2, true);	
dotx = dx + dicewidth - 3 * dotrad;	
doty = dy + 3 * dotrad;	
ctx.arc(dotx, doty, dotrad, 0, Math.PI * 2, true);	
ctx.closePath();	
ctx.fill();	
}	
function draw2mid() {	
var dotx;	
var doty;	
ctx.beginPath();	
dotx = dx + 3*dotrad;	
doty = dy + .5*diceheight;	
ctx.arc(dotx, doty, dotrad, 0, Math.PI * 2, true);	
dotx = dx + dicewidth - 3 * dotrad;	
doty = dy + .5*diceheight; *//brak zmiany*	
ctx.arc(dotx, doty, dotrad, 0, Math.PI * 2, true);	
ctx.closePath();	
ctx.fill();	
}	
</script>	
</head>	
<body>	
<canvas id="canvas" width="400" height="300">	
Twoja przeglądarka nie obsługuje elementu canvas standardu HTML5.	
</canvas>	

Kod	Omówienie
<button onClick="throwdice();">Rzuć kośćmi</button>	
<form name="f">	Początek formularza f
Etap: <input name="stage" value="Pierwszy rzut"/>	Tekst *Etap:* oraz pole tekstowe informujące o etapie gry
Punkty: <input name="pv" value=""/>	Tekst *Punkty:* oraz pole tekstowe informujące o liczbie punktów wyrzuconej w pierwszym rzucie
Wynik: <input name="outcome" value=""/>	Tekst *Wynik:* oraz pole tekstowe informujące o wyniku gry
</form>	Koniec elementu formularza
</body>	Koniec elementu body
</html>	Koniec elementu html

Wykorzystanie aplikacji do własnych potrzeb

Przystosowanie tej aplikacji do własnych potrzeb nie jest tak oczywiste, jak w przypadku aplikacji z ulubionymi odnośnikami z rozdziału 1. Reguły gry w kości pod nazwą craps są dość specyficzne. Jednak istnieje kilka rzeczy, które można zrobić. Można zmodyfikować kształt i kolory rysunków kostek, wykorzystując metody fillRect() i ustawiając różne kolory za pomocą fillStyle(). Można też zmienić kolor i wymiary całego elementu canvas. Teksty wyników mogłyby być bardziej kolorowe. Można też wykorzystać mechanikę losowania i rysowania kości do zaimplementowania innych gier.

W następnym rozdziale omawiam sposób wyświetlania gotowych ilustracji na elemencie canvas. Można go wykorzystać do wyświetlania gotowych obrazków, zamiast rysować każdą kostkę. Wada tego rozwiązania polega na tym, że musimy obsługiwać pliki zewnętrzne.

Można również zaimplementować kod przechowujący wynik ogólny. W przypadku gry hazardowej gracz może dysponować pewną kwotą „pieniędzy", np. 100, i w każdej rundzie gry obstawiać jakąś mniejszą kwotę, powiedzmy 10, które straci w przypadku przegranej, a w przypadku wygranej do „banku" wróci 20. Informację o banku można wyświetlić tak samo, jak inne komunikaty od gry, czyli w polu formularza:

```
<form name="f" id="f">
 Etap: <input name="stage" value="Pierwszy rzut"/>
 Punkty: <input name="pv" value=""/>
 Wynik: <input name="outcome" value=""/>
 Bank: <input name="bank" value="100"/>
</form>
```

JavaScript (tak jak inne języki programowania) rozróżnia między liczbami i łańcuchami znaków. Z tego powodu "100" to łańcuch trzech znaków: "1", "0", "0". Wartość 100 natomiast jest liczbą. W każdym z tych przypadków wartości te są reprezentowane wewnętrznie przez komputer w postaci sekwencji zer i jedynek. W przypadku łańcuchów znaków i pojedynczych znaków każdy znak jest reprezentowany w standardowym systemie kodowania, jak ASCII lub UNICODE. W niektórych sytuacjach JavaScript podejmuje próbę przekształcenia wartości jednego typu na inny, ale nie należy opierać na tym logiki swojej aplikacji. Zawsze zalecam stosowanie wbudowanych funkcji String() i Number() do jawnego przekształcania tego typu wartości.

W funkcji `throwdice()` przed instrukcją `if(firstturn)` dodaj kod przedstawiony w tabeli 2.6.

Tabela 2.6. Obsługa banku gracza

Kod	Omówienie
`var bank = Number(document.f.bank.value);`	Inicjalizacja stanu banku na wartość zdefiniowaną w polu formularza
`if (bank < 10) {`	Sprawdzenie stanu banku
`alert("Skończyły Ci się pieniądze! Zmodyfikuj wartość i spróbuj ponownie.");`	Jeśli w banku znajduje się mniej niż 10, wyświetl komunikat
`return;`	Zakończenie funkcji z pominięciem wykonania rzutu
`}`	Zakończenie warunku `if true`
`bank = bank - 10;`	Zmniejszenie banku o 10. Do tego miejsca kod dojdzie wyłącznie wówczas, gdy `bank` ma wartość większą lub równą 10
`document.f.bank.value = String(bank);`	Umieszczenie w polu formularza reprezentacji tekstowej stanu banku

W każdym miejscu, w którym następuje wygrana użytkownika (w pierwszym rzucie, gdy wypadnie 7 lub 11, lub w drugim rzucie, gdy wypadnie ta sama liczba oczek, co w pierwszym), dodaj kod przedstawiony w tabeli 2.7.

Tabela 2.7. Zwiększenie kwoty w banku

Kod	Omówienie
`bank = Number(document.f.bank.value);`	Ustawienie w zmiennej `bank` kwoty zapisanej w polu formularza. Ten fragment kodu pozwala użytkownikowi ręcznie modyfikować stan banku w czasie trwania gry
`bank += 20;`	Zwiększenie kwoty w banku z użyciem operatora inkrementacji += o kwotę 20
`document.f.bank.value = String(bank);`	Umieszczenie w polu formularza reprezentacji tekstowej stanu banku

Gdy gracz przegrywa lub w przypadku kolejnego rzutu nie trzeba dodawać kodu. Bank zmniejsza swoją wartość na początku każdej rundy gry.

Testowanie aplikacji i wrzucenie jej na serwer

Te wszystkie aplikacje mieszczą się w całości w dokumencie HTML. Nie są potrzebne jakiekolwiek inne pliki. Ilustracje kostek są rysowane bezpośrednio na elemencie `canvas` przez kod JavaScript. Alternatywna aplikacja napisana przeze mnie dla starszych wersji HTML wykorzystywała dwa elementy `img`. W celu podmiany rysunków kostek kod JavaScript modyfikował atrybuty `src` tych elementów. Do prawidłowego działania ta wersja aplikacji wymagała plików ilustracji kostek.

W celu przetestowania wystarczy załadować pliki HTML w przeglądarce. Pierwsza aplikacja wymaga przeładowania w celu odświeżenia widoku pojedynczej kostki. Druga aplikacja nie wymaga przeładowania, wystarczy kliknąć przycisk.

Jak wspomniałam wcześniej, solidne przetestowanie tego typu aplikacji wymaga sprawdzenia wielu kombinacji warunków. Nie wystarczy wygrać, stawiając się w roli gracza. Typowe problemy, jakie mogą wystąpić w tego typu kodzie obejmują:

- Niedopasowane znaczniki początkowe i końcowe lub ich brak.
- Niedopasowane nawiasy klamrowe { oraz } otaczające kod funkcji, instrukcji `switch` i `if`.
- Brakujące znaki cudzysłowów. W rozwiązaniu tego problemu pomocna może okazać się funkcja kolorowania składni w edytorze TextPad i innych.
- Niespójna konwencja nazw zmiennych i funkcji. Nazwy mogą być stosunkowo dowolne, ale ich użycie musi być spójne. Funkcja nazwana `draw2mid` nie może być wywołana jako `drawmid2()`.

Wszystkie powyższe problemy (oprócz ostatniego) są w rzeczywistości błędami składni, co stanowi analogię do błędów gramatycznych i interpunkcyjnych języka naturalnego. Błędy semantyki (czyli znaczeniowe) są znacznie trudniejsze do wykrycia. Jeśli instrukcja `switch` została napisana tak, że gracz wygrywa po wyrzuceniu siódemki lub przegrywa przy tej samej wartości, gra będzie działać błędnie nawet wówczas, gdy kod JavaScript jest idealny z punktu widzenia składni.

Jeśli dokładnie skopiujesz mój kod, wszystko powinno działać prawidłowo, ale pisząc własne gry, możesz na przykład zapomnieć, że współrzędne pionowe zwiększają się z góry do dołu.

Podsumowanie

W tym rozdziale nauczyłeś się następujących zagadnień:

- deklarowania zmiennych i używania zmiennych globalnych do reprezentowania stanu aplikacji;
- pisania kodu wykonującego obliczenia arytmetyczne;
- definiowania i używania funkcji programisty;
- wykorzystania funkcji wbudowanych języka JavaScript, między innymi metod `Math.random()` i `Math.floor()`;
- instrukcji `if` i `switch`;
- tworzenia elementu `canvas` w dokumencie HTML;
- rysowania prostokątów i okręgów.

Ten rozdział wprowadził jedną z kluczowych nowości standardu HTML5 — element `canvas` — jak również sposób generowania zdarzeń losowych i elementy interaktywności. Miałeś też okazję zapoznać się z kilkoma z technik programowania, które będą wykorzystane w innych przykładach z tej książki. Szczególnie użyteczna jest technika etapowego budowania aplikacji. Następny rozdział zajmuje się animacją piłki odbijającej się w pudełku. Jest to wprowadzenie do rzeczywistej gry z rozdziału 4., w którym zaimplementujemy symulację balistyczną w postaci gier polegających na strzelaniu z armaty i procy.