helion.pl

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

Flash MX. Ćwiczenia zaawansowane



Autor: Daniel Bargieł
ISBN: 83-7361-044-8
Format: B5, stron: 156
Przykłady na ftp: 6722 kB

Książka „Flash MX. Ćwiczenia zaawansowane" została napisana z myślą o tych osobach, które tworząc we Flashu animacje, najczęściej korzystają z języka skryptowego, jakim jest ActionScript. Zawarty w niej materiał został dobrany tak, aby po jej przeczytaniu każdy mógł stworzyć dowolnie złożony pod względem programistycznym projekt animacji. Książka ta jest przeznaczona dla osób, które nie tylko znają dobrze Flasha MX, ale także potrafią korzystać z języka skryptowego, jakim jest ActionScript.

Ćwiczenia obejmują szeroki zakres zastosowania ActionScriptu. Dzięki nim, będziesz w stanie tworzyć zaawansowane gry we Flashu (także trójwymiarowe), a także wykorzystasz możliwości w zakresie transmisji strumieniowych i aplikacji interaktywnych typu czat. Jednym słowem — staniesz się prawdziwym ekspertem Flasha.

Omówiono:

- Animację modelu 3D na podstawie danych zapisanych w pliku XML
- Tworzenie i animowanie postaci w grach
- Tworzenie plansz i edytorów plansz
- Wykrywanie kolizji w grach
- Tworzenie inteligentnych przeciwników
- Wykorzystywanie w grach praw fizyki
- Użycie Flash Communication Server MX do tworzenia transmisji wideo i aplikacji typu chat

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

Wprowadzenie 5

Rozdział 1. Animacja modelu 3D 7
Tworzenie sześcianu 8
Podsumowanie 22

Rozdział 2. Zapisywanie geometrii w pliku XML 23
Podsumowanie 35

Rozdział 3. Kontrola oraz animacja postaci 37
Sterowanie prostą postacią 37
Sterowanie złożoną postacią 42
Sterowanie postacią za pomocą myszy 49
Podsumowanie 57

Rozdział 4. Tworzenie plansz w grach 59
Ładowanie mapy z pliku XML 60
Edytor map 64
Podsumowanie 76

Rozdział 5. Detekcja kolizji 77
Prosta detekcja kolizji metodą hitTest 77
Złożona detekcja kolizji metodą hitTest 83
Alternatywny sposób detekcji kolizji 91
Podsumowanie 96

Rozdział 6. Inteligentni przeciwnicy 97
Zachowania spontaniczne 97
Algorytm odszukiwania ścieżki 102
Podsumowanie 113

Rozdział 7. Fizyka w animacjach 115
Bezwładność obiektu w przestrzeni 115
Grawitacja oraz zderzenie 119
Podsumowanie 128

Rozdział 8. Transmisja strumieni wideo — Flash Communication Server MX 129
Co potrafi Flash Player 6 130
Serwer Flash Communication Server MX 133
Wymiana strumieni wideo 136
Podsumowanie 144

Rozdział 9. Aplikacja Chat — Flash Communication Server MX 145
Projekt aplikacji Chat 145
Narzędzia administracyjne 152
Podsumowanie 155
Zakończenie 155

Rozdział 3.

Kontrola oraz animacja postaci

Podczas tworzenia gier komputerowych — niezależnie od tego, czy korzystamy z jakiegoś języka programowania czy też z aplikacji takiej jak Flash MX — wcześniej czy później napotkamy problem kontroli i animowania postaci występujących w grze.

Może on dotyczyć prostego wyświetlania różnych animacji obrazujących różne stany postaci i przesuwania jej w lewo i w prawo. Na przykład, sterując samolotem, wyświetlamy inny obraz, gdy znajduje się on w stanie „spoczynku" (nie kontrolujemy go w żaden sposób), inny, gdy leci w lewo (np. obraz samolotu przechylonego na lewe skrzydło), a inny, gdy leci w prawo (obraz samolotu przechylonego na prawe skrzydło).

Problem może być też bardzo złożony, gdy np. sterujemy postacią z kreskówki, która posiada animację chodu, obrotu w lewo, w prawo, schylania się itd. Do tego kontrolujemy ją za pomocą myszy lub kombinacji klawiszy.

W tym rozdziale zapoznamy się dokładnie z tym zagadnieniami, wykonując ćwiczenia polegające na kontrolowaniu i animowaniu prostych oraz złożonych postaci, które mogą występować w grach.

Sterowanie prostą postacią

Ćwiczenie 3.1.

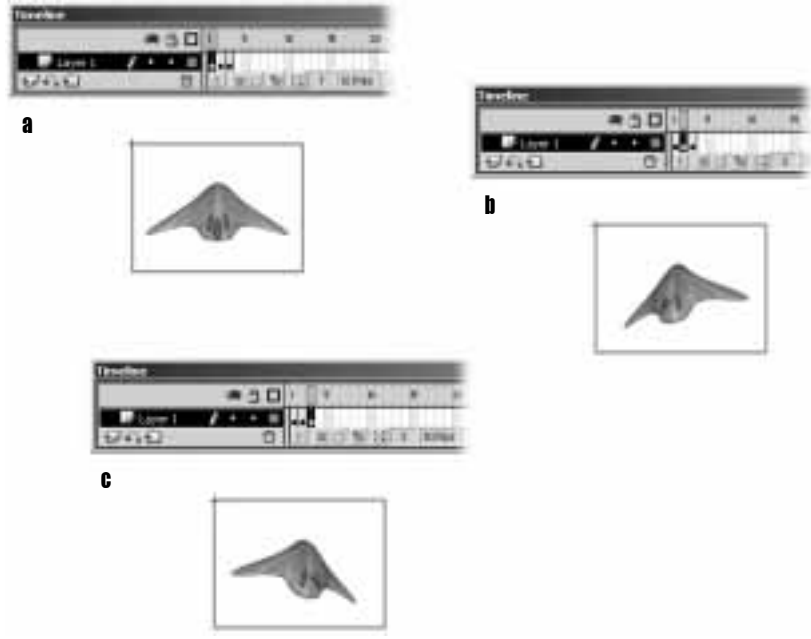
W pierwszym ćwiczeniu poznamy sposób tworzenia animacji, w której za pomocą klawisza lewej strzałki lub prawej strzałki będziemy mogli sterować rakietą przesuwającą się po ekranie. Gotową animację SWF, którą stworzymy w tym ćwiczeniu, można znaleźć w katalogu Rozdzial03/Gotowe/SterowanieRakieta.swf.

1. Otwórzmy Flash MX lub stwórzmy w nim nowy projekt.
2. Z menu *Insert* wybierzmy polecenie *New Symbol*, a następnie za pomocą okna *Create New Symbol* stwórzmy symbol typu *Movie Clip* o nazwie *Rakieta*.

   Symbol *Rakieta* będzie stanowił obiekt, który chcemy kontrolować w tej animacji
3. Upewniwszy się, że Flash jest w trybie edycji symbolu *Rakieta*, z menu *File* wybierzmy polecenie *Import*, a następnie z katalogu *Rozdzial03/Cwiczenia/Rakieta* załadujmy plik *Rakieta00.bmp*.

   Flash MX powinien zapytać, czy załadować pozostałe dwa pliki *Rakieta01.bmp* oraz *Rakieta02.bmp*, które jako całość mogą stanowić animację. Kliknijmy przycisk *Tak*.

   Gdy to zrobimy, program w kolejnych klatkach animacji symbolu *Rakieta* umieści bitmapy *Rakieta00*, *Rakieta01* oraz *Rakieta02* (patrz rysunek 3.1).

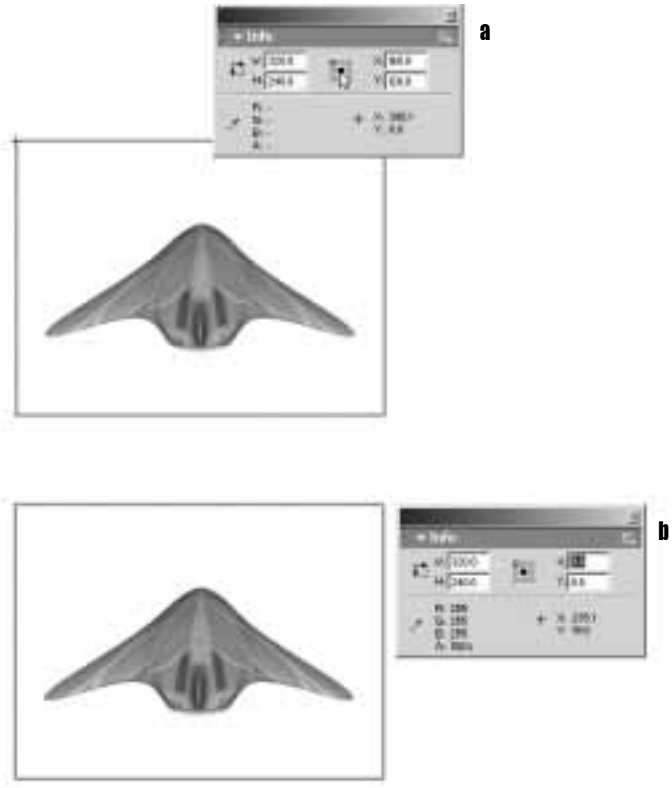


Rysunek 3.1. *Trzy ujęcia symbolu Rakieta po zaimportowaniu do niego sekwencji bitmap*
a) Pierwsze ujęcie kluczowe — rakieta w położeniu neutralnym
b) Drugie ujęcie kluczowe — rakieta skręca w lewo
c) Trzecie ujęcie kluczowe — rakieta skręca w prawo

Po umieszczeniu bitmap w symbolu *Rakieta* lewy górny róg bitmapy w każdym z ujęć symbolu będzie znajdował się w centrum symbolu (patrz rysunek 3.2a). Nam bardziej może odpowiadać sytuacja, w której centrum symbolu znajduje się w centrum bitmapy (patrz rysunek 3.2b).

Aby dokładnie pokryć centrum bitmapy z centrum symbolu, najlepiej posłużyć się panelem *Info*. W pola tekstowe *X* oraz *Y* należy wypisać 0.0, upewniwszy się wcześniej, że operacje będą dotyczyć punktu centralnego bitmapy, a nie jej lewego górnego rogu. Operację tę powinniśmy wykonać dla każdego z trzech ujęć symbolu *Rakieta*.

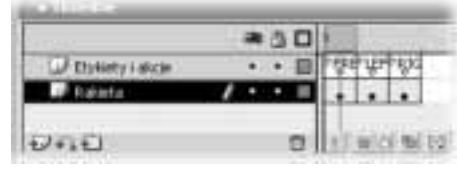
Przy tworzeniu całej gry a nie jedynie animacji, w której kontrolujemy rakietę, powinniśmy zamienić bitmapy na wypełnienia (polecenie *Modify/Break Apart*), po czym usunąć białe tło i zostawić jedynie obraz rakiety.



Rysunek 3.2.
Zmiana położenia centralnego punktu bitmapy za pomocą panelu Info
a) Przed zmianą położenia
b) Po zmianie położenia

4. Ponieważ każde z trzech ujęć symbolu *Rakieta* przedstawia inny stan postaci, powinniśmy znaleźć prosty sposób odwoływania się do tych ujęć. W tym celu nadamy każdemu z nich odpowiednią etykietę.

Stwórzmy więc nową warstwę i nadajmy jej nazwę *Etykiety i akcje*. Następnie w każdej klatce nowej warstwy wstawmy puste ujęcie kluczowe i, korzystając z panelu *Properties*, ustalmy kolejno ich nazwy jako *FREE*, *LEFT* oraz *RIGHT* (patrz rysunek 3.3).



Rysunek 3.3.
Nazwy ujęć pozwolą na szybkie odwołanie się do każdego z nich

5. Teraz za pomocą panelu *Actions* umieśćmy w każdym z ujęć warstwy *Etykiety i akcje* prosty skrypt:

```
//Zatrzymanie animacji rakiety
stop();
```

Dzięki temu będziemy mieli pewność, że animacja rakiety nie zostanie odtworzona.

6. Symbol *Rakieta* jest już gotowy, możemy więc powrócić do trybu edycji animacji głównej. Powinniśmy teraz umieścić odnośnik do symbolu *Rakieta* w obszarze roboczym, tak jak na rysunku 3.4.

Rysunek 3.4.
Odnośnik do symbolu Rakieta w obszarze roboczym



Ponieważ odnośnik będzie dość duży, za pomocą panelu *Transform* powinniśmy go zmniejszyć do około 45% pierwotnego rozmiaru.

7. Teraz, korzystając z panelu *Properties*, nadajmy odnośnikowi do symbolu *Rakieta* nazwę *Rakieta*.

8. Obiekt, którym będziemy sterować, jest już gotowy. Przystępujemy więc do tworzenia głównego kodu sterującego. W animacji głównej stwórzmy warstwę o nazwie *Akcje*, po czym wyselekcjonujmy jej pierwsze i jedyne ujęcie kluczowe, a następnie otwórzmy panel *Actions*.

9. Rozpoczynamy wprowadzanie kodu ActionScript sterującego zachowaniem rakiety:

```
//Status rakiety
PLAYER_STATUS = "FREE";
```

Zmienna globalna `PLAYER_STATUS` informuje, w jakim stanie znajduje się aktualnie rakieta. Istnieją trzy możliwości: wartość domyślna `FREE` oznacza, że obiekt znajduje się w spoczynku, wartość `LEFT` — rakieta przemieszcza się w lewo, natomiast wartość `RIGHT` — rakieta przemieszcza się w prawo.

10. Teraz musimy stworzyć metody, które będą reagowały na wciśnięcie oraz zwolnienie odpowiednich klawiszy:

```
/Obiekt nasłuchu klawiatury
Klawiatura = new Object();
```

Obiekt `Klawiatura` przypiszemy jako obiekt nasłuchu do klawiatury, czyli obiektu `Key`.

```
//Kod wykonywany, gdy użytkownik
//naciśnie klawisz
Klawiatura.onKeyDown = function () {
  if (Key.isDown(Key.LEFT))
  {
        PLAYER_STATUS = "LEFT";
  } else if (Key.isDown(Key.RIGHT))
        PLAYER_STATUS = "RIGHT";
} //Koniec metody onKeyDown
```

Kod metody onKeyDown, jak łatwo się domyślić, zostanie wykonany, gdy wciśniemy dowolny klawisz podczas odtwarzania animacji SWF. Jeśli wciśniemy klawisz *strzałki w lewo*, zmiennej PLAYER_STATUS zostanie przypisana wartość LEFT, jeśli natomiast wciśniemy klawisz *strzałki w prawo*, zmiennej PLAYER_STATUS zostanie przypisana wartość RIGHT. Dalej piszemy:

```
//Kod wykonywany, gdy użytkownik
//zwolni klawisz
Klawiatura.onKeyUp = function() {
  if (PLAYER_STATUS == "LEFT")
  {
        if (Key.getCode() == 37)
              PLAYER_STATUS = "FREE";
  } else if (PLAYER_STATUS == "RIGHT")
        if (Key.getCode() == 39)
              PLAYER_STATUS = "FREE";
} //Koniec metody onKeyUp
```

Metoda onKeyUp jest wywoływana, gdy zwolnimy wciśnięty wcześniej klawisz. Nie możemy jednak po prostu przypisać zmiennej PLAYER_STATUS wartości FREE (która oznacza, że rakieta się nie porusza), ponieważ metoda byłaby wywoływana nawet wtedy, kiedy zwolnilibyśmy inny klawisz, np. *spację*. Najpierw należy sprawdzić, który klawisz został zwolniony oraz jaka jest aktualna wartość zmiennej PLAYER_STATUS.

Jeśli wartość zmiennej PLAYER_STATUS to LEFT, a kod ASCII zwolnionego klawisza 37 (klawisz *strzałki w lewo*), oznacza to, że zwolniliśmy wciśnięty wcześniej klawisz *strzałki w lewo*. Podobnie ma się sprawa z klawiszem *strzałki w prawo*.

Musimy jeszcze przypisać obiekt Klawiatura do klawiatury komputerowej jako obiekt nasłuchu:

```
//Przypisanie obiektu nasłuchu do klawiatury
Key.addListener(Klawiatura);
```

11. Aby kod był kompletny, musimy jeszcze stworzyć funkcję, która przekształci wartość zmiennej PLAYER_STATUS w konkretne działanie, czyli ruch rakiety:

```
//Funkcja animująca rakietę
function render() {

  //Wyświetlenie odpowiedniego obrazka rakiety
  Rakieta.gotoAndStop(PLAYER_STATUS);
```

Pamiętamy etykiety w symbolu Rakieta? Ich nazwy odpowiadają dokładnie wartościom, jakie przyjmuje zmienna PLAYER_STATUS, dzięki czemu możemy od razu przekształcić wartość tej zmiennej w konkretny obrazek rakiety.

```
  switch (PLAYER_STATUS)
  {
        case "LEFT": //Ruch w lewo
              Rakieta._x -= 7;
              if (Rakieta._x < 50) Rakieta._x = 50;
        break;
        case "RIGHT": //Ruch w prawo
              Rakieta._x += 7;
              if (Rakieta._x > (Stage.width-50))
                    Rakieta._x = (Stage.width-50);
        break;
  }
} //Koniec funkcji render
```

W zależności od wartości zmiennej `PLAYER_STATUS` przemieszczamy odnośnik do symbolu *Rakieta* w lewo lub w prawo. Instrukcje warunkowe `if` zapewniają, że rakieta nie wyjedzie poza obszar ekranu animacji (ograniczenie 50 pikseli od lewej oraz prawej krawędzi).

12. Mamy już wszystkie niezbędne funkcje i obiekty, dzięki którym możemy kontrolować animację rakiety. Teraz musimy jeszcze stworzyć samą animację. We Flashu 5 wiązałoby się to z dodaniem kolejnych klatek do projektu oraz zapętleniem animacji, tak jak zrobiliśmy w poprzednich dwóch rozdziałach, tworząc animację obiektów 3D.

    Flash MX ma jednak bardzo przydatną funkcję `setInterval`, dzięki której możemy wywoływać funkcje oraz metody obiektów w ściśle określonych odstępach czasu. Ponieważ w naszym przypadku funkcją animującą jest `render`, wystarczy wywoływać ją z odstępem, powiedzmy, 50 ms (milisekund, 1ms = 1/1000 sekundy). Piszemy więc:

```
//Cykliczne wywołania funkcji render
setInterval(render, 50);
```

13. Wartość 50 ms oznacza, że funkcja `render` będzie wywoływana 20 razy na sekundę. Musimy jeszcze ustalić taką samą prędkość odtwarzania animacji Flasha, aby była płynna. Możemy zrobić to poprzez kliknięcie w obszarze roboczym, a następnie ustawienie w panelu *Properties* wartości *Frame Rate* na 20.

    Możemy również otworzyć okno *Document Properties* i tam dokonać stosownej zmiany.

Projekt jest już gotowy i możemy go opublikować. Przykładowe wykonanie tego ćwiczenia znajdziemy w katalogu *Rozdzial03/Gotowe*.

W podobny sposób komputer może kontrolować przeciwników w grze. Jednak o tym opowiemy w rozdziale 6. *Inteligentni przeciwnicy*.

Sterowanie złożoną postacią

Animacja rakiety, którą wykonaliśmy w poprzednim rozdziale, była doskonałym przykładem sterowania postacią niezbyt skomplikowaną. Jednak podczas tworzenia ciekawych gier mamy do czynienia z postaciami, które są o wiele bardziej złożone.

Przez postacie złożone będziemy rozumieli takie, które w celu oddania poszczególnych stanów dysponują nie pojedynczym obrazkiem, ale całą animacją, którą musimy wyświetlić w odpowiednim momencie. Powracając do przykładu z rakietą, moglibyśmy tak zmienić symbol rakiety, aby przy skręcie w lewo pokazana była animacja pochylania się obiektu na lewe skrzydło, a po zwolnieniu klawisza jego powrót do stanu początkowego.

Ćwiczenie 3.2.

W tym rozdziale wykonamy animację postaci bałwana, który jest częścią gry Snow Wars. Grę możemy znaleźć w katalogu Gry. Proponujemy zagrać w nią, aby zobaczyć, jak wygląda animacja bałwana.

1. Otwórzmy we Flashu MX projekt *Balwan.fla*, który znajdziemy w katalogu *Rozdzial03/Cwiczenia*.

   Po otwarciu projektu zobaczymy na środku obszaru roboczego postać, którą będziemy animować (patrz rysunek 3.5).

Rysunek 3.5. *Postać bałwana, którą będziemy animować*



Bałwan będzie posiadał kilka faz ruchu. Faza typu *STAND* oznacza, że bałwan stoi w miejscu. Będzie temu towarzyszyła animacja podrygującego bałwana powtarzana bez przerwy, jeśli użytkownik nie naciśnie żadnego klawisza sterującego.

Druga faza typu *WALK* oznacza, że bałwan porusza się w prawo lub w lewo (ponieważ postać jest narysowana w rzucie pseudoizometrycznym, rzeczywiście będzie się poruszać po skosie ekranu). Fazie tej towarzyszy animacja poruszającego się bałwana. Jest ona odtwarzana bez przerwy, jeśli gracz trzyma wciśnięte klawisze kierunkowe (klawisz *strzałki w lewo* lub *strzałki w prawo*).

Trzecia faza ruchu nazywa się fazą *TAKE* i oznacza, że bałwan podnosi śnieżkę z ziemi. Fazie tej towarzyszy animacja bałwana schylającego się po śnieżkę. Animacja ta jest odgrywana tylko raz, gdy gracz naciśnie spację.

Zanim bałwan będzie mógł po raz kolejny podnieść śnieżkę, musi najpierw wyrzucić trzymaną, co jest czwartą i ostatnią fazą ruchu bałwana o nazwie *THROW*. Fazie tej towarzyszy animacja, w której bałwan wyrzuca śnieżkę (wykonuje odpowiedni ruch, animacją samej śnieżki nie będziemy się zajmować). Animacja ta jest wyświetlana tylko jeden raz, po naciśnięciu spacji.

Pomiędzy fazami *TAKE* oraz *THROW* zachodzi następujący związek: naciskamy *spację* — bałwan podnosi śnieżkę (faza *TAKE*), naciskamy *spację* po raz drugi — bałwan wyrzuca śnieżkę (faza *THROW*), znów naciskamy *spację* — bałwan podnosi śnieżkę (faza *TAKE*) itd. Jak więc widać, do *spacji* są przypisane na zmianę dwie animacje.

2. Przejdźmy do trybu edycji symbolu bałwana poprzez wybranie z podręcznego menu odnośników polecenia *Edit*.

Zobaczymy, że animacje poszczególnych faz ruchu są już gotowe oraz umieszczone kolejno po sobie (patrz rysunek 3.5). Naszym zadaniem będzie więc jedynie odpowiednia synchronizacja poszczególnych faz animacji.



Gdy tworzymy animowane postacie, poszczególne fazy animacji są zazwyczaj zapisane w jednej dużej animacji, tak jak w naszym przypadku. Zadaniem programisty jest odpowiednie ich rozdzielenie.

3. Jeśli opublikujemy teraz animację, zobaczymy, że wyświetlane są po kolei poszczególne fazy ruchu. Wiemy jednak, że podczas bezczynności gracza powinna być wyświetlana jedynie faza *STAND*.

   Aby tak się stało, będąc w trybie edycji symbolu bałwana, wyselekcjonujmy w panelu *Timeline* piątą klatkę animacji na warstwie *Akcje* (patrz rysunek 3.6), a następnie za pomocą panelu *Actions* dodajmy do ujęcia kluczowego znajdującego się w tej klatce następujący prosty skrypt:

```
//Zapętlenie animacji w fazie STAND
gotoAndPlay("STAND");
```

Rysunek 3.6.
Należy zapętlić animację fazy STAND, aby podczas braku aktywności gracza bałwan nie wykonywał zbędnych ruchów



4. Powróćmy do animacji głównej, wybierając z menu *Edit* polecenie *Edit Document*. Wyselekcjonujmy odnośnik bałwana znajdujący się w obszarze roboczym i za pomocą panelu *Properties* nadajmy mu nazwę *SNOWMAN*.

5. Stwórzmy teraz nową warstwę i nadajmy jej nazwę *Akcje*. Na warstwie umieścimy cały kod ActionScript, który będzie kontrolował bałwana.

6. Wyselekcjonujmy pierwsze ujęcie kluczowe na warstwie *Akcje*, otwórzmy panel *Actions*, a następnie rozpocznijmy wprowadzanie kodu:

```
//Zmienna określająca status bałwana
PLAYER_STATUS = "STAND";

//Marker informujący, czy bałwan posiada śnieżkę czy też nie
SNOWBALL = false;
```

   Zmienna `PLAYER_STATUS` powinna być już znana, ponieważ korzystaliśmy z niej w poprzednim projekcie. Będzie ona przyjmowała jedną z pięciu wartości: `STAND` — bałwan nieruchomy, `LEFT` — bałwan porusza się w lewo, `RIGHT` — bałwan porusza się w prawo, `TAKE` — bałwan podnosi śnieżkę, `THROW` — bałwan wyrzuca śnieżkę.

   Zmienna `SNOWBALL` przyjmuje wartość `true`, jeśli bałwan ma śnieżkę (czyli podniósł ją wcześniej), oraz `false`, jeśli śnieżki nie ma. Jest ona potrzebna do prawidłowego określenia wartości zmiennej `PALYER_STATUS` po naciśnięciu przez gracza klawisza *spacji*.

7. Należy teraz stworzyć obiekt nasłuchu, który przypiszemy do klawiatury (reprezentowanej przez obiekt `Key`) oraz zdefiniować jego metody `onKeyDown` oraz `onKeyUp`:

```
//Tworzenie obiektu nasłuchu dla klawiatury
Klawiatura = new Object();

//Metoda wykonywana, gdy gracz wciśnie
//dowolny klawisz
Klawiatura.onKeyDown = function() {
  if (Key.isDown(Key.LEFT))
        PLAYER_STATUS = "LEFT";
  else if (Key.isDown(Key.RIGHT))
        PLAYER_STATUS = "RIGHT";
  else if (Key.isDown(Key.SPACE))
        if (!SNOWBALL)
              PLAYER_STATUS = "TAKE";
        else PLAYER_STATUS = "THROW";
} //Koniec metody onKeyDown
```

Jak widać, wartość zmiennej `PLAYER_STATUS` ściśle zależy do tego, który klawisz wciśniemy. W przypadku spacji sprawdzana jest dodatkowo zmienna `SNOWBALL`, która decyduje, czy bałwan znajduje się w fazie `TAKE` czy też `THROW`. Wartość zmiennej `SNOWBALL` ustalamy bezpośrednio w kodzie symbolu bałwana.

Dalej piszemy:

```
//Metoda wykonywana, gdy gracz zwolni
//wciśnięty wcześniej klawisz
Klawiatura.onKeyUp = function() {
  if (PLAYER_STATUS == "LEFT")
  {
        if (Key.getCode() == 37) //Klawisz: Strzałka w lewo
              PLAYER_STATUS = "STAND";
  }
  else if (PLAYER_STATUS == "RIGHT")
  {
        if (Key.getCode() == 39) //Klawisz: Strzałka w prawo
              PLAYER_STATUS = "STAND";
  }
  else if (PLAYER_STATUS == "TAKE" || PLAYER_STATUS == "THROW")
        if (Key.getCode() == 32) //Klawisz: Spacja
              PLAYER_STATUS = "STAND";
} //Koniec metody onKeyUp
```

Przy zwolnieniu klawisza sprawdzamy wartość zmiennej `PLAYER_STATUS` oraz kod zwolnionego klawisza. Kod klawisza oraz wartość zmiennej muszą się zgadzać, aby bałwan mógł przejść do fazy `STAND`.

Musimy jeszcze przypisać obiekt nasłuchu do obiektu `Key`, który reprezentuje klawiaturę.

```
//Przypisanie obiektu Klawiatura
//jako obiektu nasłuchu do obiektu Key
Key.addListener(Klawiatura);
```

Pora stworzyć funkcję, która zamieni wartość zmiennej `PLAYER_STATUS` na odpowiednią animację oraz ruch bałwana. W przykładzie z rakietą była to funkcja `render`, która przekształcała wartość zmiennej `PALYER_STATUS` bezpośrednio w odpowiedni obraz statku kosmicznego.

Niestety, w tym przypadku nie możemy sobie na to pozwolić, ponieważ poszczególne fazy animacji nie posiadają jednego obrazu. Gdybyśmy zastosowali takie rozwiązanie, przy częstotliwości wywołań funkcji `render` 20 razy na sekundę nie bylibyśmy w stanie wyświetlić więcej niż jedną klatkę animacji każdej z faz.

Musimy więc stworzyć mechanizm, który zabezpieczy nas przed wystąpieniem takiej sytuacji. Dodamy do kodu ActionScript symbolu bałwana nową zmienną o nazwie `AnimStatus`, która będzie informowała kod, w jakiej fazie animacji znajduje się bałwan w danym momencie.

8. Przejdźmy do trybu edycji symbolu bałwana, a następnie wyselekcjonujmy pierwszą klatkę animacji na warstwie *Akcje* i dodajmy do niej skrypt:

```
//Faza animacji to STAND
AnimStatus = "STAND";
```

Wartość `STAND` zmiennej `AnimStatus` oznacza, że bałwan znajduje się w pierwszej fazie animacji.

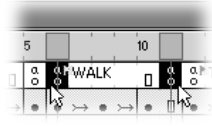
9. Wyselekcjonujmy teraz szóstą klatkę animacji, w której znajduje się ujęcie kluczowe z etykietą `WALK` i dodajmy do niego kod:

```
//Faza animacji to WALK
AnimStatus = "WALK";
```

natomiast do ostatniej klatki animacji tej fazy (klatka 11.) (patrz rysunek 3.7) dodajmy kod:

```
//Zapętlenie animacji w fazie WALK
gotoAndPlay("WALK");
```

Rysunek 3.7.
Każda faza animacji zaczyna się od ujęcia z etykietą, a kończy na następnym ujęciu kluczowym



W pierwszym przypadku ustawiamy wartość zmiennej `AnimStatus` na `WALK`, co oznacza, że bałwan znajduje się w drugiej fazie ruchu, a następnie zapętlamy animację tej fazy ruchu, aby uzyskać płynność.

10. Wyselekcjonujmy teraz klatkę dwunastą, w której mamy ujęcie kluczowe z etykietą `TAKE` oznaczające początek animacji fazy `TAKE`, po czym do tego ujęcia wstawmy kod:

```
//Faza animacji to TAKE
AnimStatus = "SPACE";
```

Jak widać, zmienna `AnimStatus` przyjmuje teraz wartość `SPACE`, która będzie określała trzecią oraz czwartą fazę ruchu bałwana. Rozgraniczenie tych faz ruchu nastąpi w funkcji `render` i nie ma sensu robienie tego teraz.

Wyselekcjonujmy kończące fazę `TAKE` ujęcie kluczowe, które znajduje się w klatce dwudziestej. Dodajmy do niego kod:

```
//Koniec fazy TAKE
//Bałwan ma śnieżkę
AnimStatus = "NULL";
_root.SNOWBALL = true;
stop();
```

W tym ujęciu zmieniamy wartość zmiennej AnimStatus na NULL, aby poinformować funkcję sterującą, że faza ruchu TAKE lub THROW skończyła się, ale nie zaczęła się jeszcze inna faza ruchu. Dzięki wartości zmiennej SNOWBALL po zakończeniu fazy TAKE ustawianej na true wiemy, z którą fazą mamy do czynienia. Na koniec animacja zostaje zatrzymana, aby nie wyświetlać kolejnych klatek.

11. Pozostała nam jeszcze ostatnia faza ruchu, która zaczyna się w klatce dwudziestej pierwszej. Wstawmy do niej kod:

```
//Faza animacji to THROW
AnimStatus = "SPACE";
```

W ostatniej klatce animacji bałwana umieśćmy kod:

```
//Koniec fazy THROW
//Bałwan nie ma śnieżki
AnimStatus = "NULL";
_root.SNOWBALL = false;
stop();
```

Kod ten jest bardzo podobny do umieszczonego w ujęciu kończącym fazę ruchu TAKE, z tą różnicą, że teraz wartość zmiennej SNOWBALL ustawiamy na false, co oznacza, że bałwan nie ma śnieżki.

12. Animacja symbolu bałwana jest już gotowa, przejdźmy więc do edycji animacji głównej. Wyselekcjonujmy pierwszą klatkę na warstwie *Akcje* i otwórzmy panel *Actions*, a następnie rozpocznijmy dopisywanie kodu na dole tego, który znajduje się już w panelu:

```
//Funkcja wyświetlająca odpowiednie animacje bałwana
function render() {
  if (SNOWMAN.AnimStatus != "SPACE")
  {
```

Jak widać, pierwszą czynnością wykonywaną przez kod jest sprawdzanie wartości zmiennej AnimStatus odnośnika SNOWMAN. Jeśli wartość tej zmiennej wynosi SPACE, oznacza to, że w tym momencie bałwan albo podnosi śnieżkę, albo ją wyrzuca. Funkcja nie zrobi więc nic, dopóki animacje tych dwóch faz ruchu nie zostaną zakończone. Dalej piszemy:

```
switch (PLAYER_STATUS)
{
       case "STAND": //Faza STAND
              if (SNOWMAN.AnimStatus != "STAND")
                     SNOWMAN.gotoAndPlay("STAND");
       break;
```

Jeśli wartość zmiennej PLAYER_STATUS wynosi STAND, sprawdzana jest wartość zmiennej AnimStatus odnośnika SNOWMAN. Zapobiega to rozpoczynaniu animacji fazy STAND kilkanaście razy na sekundę, czego skutkiem byłoby jedynie wyświetlanie pierwszej klatki animacji. Jeśli zmienna AnimStatus odnośnika SNOWMAN wynosi STAND, znaczy to, że animacja pierwszej fazy została już rozpoczęta i jest w trakcie odtwarzania.

Dalej piszemy:

```
       case "LEFT": //Faza WALK
              SNOWMAN._x -= 6;
              SNOWMAN._y -= 3;
```

```
                if (SNOWMAN.AnimStatus != "WALK")
                        SNOWMAN.gotoAndPlay("WALK");
                break;
        case "RIGHT": //FAZA WALK
                SNOWMAN._x += 6;
                SNOWMAN._y += 3;
                if (SNOWMAN.AnimStatus != "WALK")
                        SNOWMAN.gotoAndPlay("WALK");
        break;
```

W powyższym fragmencie kodu, jeśli zmienna PLAYER_STATUS wynosi LEFT, przesuwamy odnośnik bałwana w lewo o sześć jednostek oraz do góry o trzy jednostki (wymaga tego pseudoizometryczny rysunek bałwana). Odtwarzana jest również animacja fazy WALK z zabezpieczeniem identycznym jak omówione przed chwilą.

Jeśli wartość zmiennej PLAYER_STATUS wynosi RIGHT, odnośnik SNOWMAN przesuwamy o sześć jednostek w prawo oraz o trzy jednostki w dół obszaru roboczego. Odtwarzamy do tego animację fazy WALK. W powyższym kodzie brakuje zabezpieczenia przed wyjściem bałwana poza obszar roboczy animacji. Jest ono bardzo proste do wykonania. Gdy bałwan porusza się w lewo, powinniśmy jedynie sprawdzić, czy jego zmienna _x nie jest mniejsza od pewnej wartości. W przypadku, w którym bałwan porusza się w prawo, sprawdzamy, czy zmienna _x odnośnika SNOWMAN nie jest większa od pewnej wartości.

Zostały nam jeszcze dwie fazy ruchu:

```
                case "TAKE": //Faza TAKE
                        if (!SNOWBALL)
                        SNOWMAN.gotoAndPlay("TAKE");
                break;
                case "THROW": //Faza THROW
                        if (SNOWBALL)
                        SNOWMAN.gotoAndPlay("THROW");
                break;
        }
    }
  } //Koniec funkcji render
```

W fazie TAKE nie przesuwamy bałwana, ponieważ podnosi on śnieżkę, stojąc w miejscu. Wyświetlamy jedynie animację fazy TAKE i to pod warunkiem że bałwan nie ma śnieżki (zmienna SNOWBALL wynosi false). Zabezpiecza to przed sytuacją, w której bałwan mógłby dwa razy podnieść śnieżkę. Pamiętajmy, że zmienna SNOWBALL jest ustawiana na true, gdy zakończy się animacja fazy TAKE.

W fazie THROW podobnie jak w fazie TAKE nie przesuwamy bałwana. Wyświetlamy animację THROW i to tylko wtedy, gdy bałwan ma śnieżkę (czyli gdy zmienna SNOWBALL posiada wartość true). Przedstawiony mechanizm zabezpiecza bałwana przed próbą kilkukrotnego wyrzucenia śnieżki bez fazy TAKE. Zmienna SNOWBALL jest ustawiana na false, gdy wyświetlona zostanie ostatnia klatka animacji fazy THROW.

Dzięki kodowi kontrolującemu animację fazy TAKE oraz THROW możemy wcisnąć spację i, przytrzymując ją, obserwować jak bałwan kolejno schyla się po śnieżkę, wyrzuca ją, ponownie bierze śnieżkę, wyrzuca ją i tak bez przerwy, dopóki nie zwolnimy spacji.

13. Pozostało jeszcze wywołanie funkcji `setInterval`, która w określonych odstępach czasu będzie wywoływać funkcję `render`. Piszemy więc:

```
//Regularne wywołania funkcji render
setInterval(render, 50);
```

a następnie w oknie *Document Properties* (*Modify / Document*) ustalamy prędkość animacji na 20 klatek na sekundę.

Animacja jest już gotowa i możemy ją opublikować. Gotową animację FLA, która powstanie po wykonaniu powyższego ćwiczenia, możemy znaleźć w pliku *SterowanieBalwanem.fla* w katalogu *Rozdział03/Gotowe*.

Sterowanie postacią za pomocą myszy

W dwóch ćwiczeniach, które wykonaliśmy w tym rozdziale, poznaliśmy sposoby sterowania postacią bezpośrednio za pomocą klawiatury. Często jednak zdarza się, że wygodniej jest sterować myszą. Mówimy tutaj o sytuacji, w której każdy ruch myszy przekłada się bezpośrednio na położenia oraz orientację postaci.



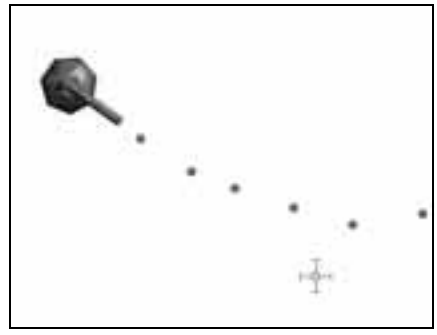
Nie chodzi o sytuację, w której klikamy myszą mapę, a następnie postać idzie do wskazanego punktu. Ten sposób sterowania zakwalifikowalibyśmy do *sterowania postacią złożoną*, z tą różnicą, że postacią nie steruje gracz, lecz komputer.

W tym rozdziale poznamy prosty sposób sterowania postacią za pomocą kursora myszy. Dowiemy się również, jak tworzyć animację pocisków, które są nieodzownym elementem gier-strzelanek.

Ćwiczenie 3.3.

Naszym zadaniem będzie stworzenie animacji, w której działo znajdujące się na środku obszaru roboczego w widoku z góry, będzie się obracało zgodnie z ruchem celownika przypisanego do kursora myszy. Po naciśnięciu lewego przycisku myszy z działa zostanie wystrzelony pocisk lecący w określonym kierunku (patrz rysunek 3.8). Gotową animację SWF, która będzie efektem naszej pracy w tym ćwiczeniu, można znaleźć w pliku Dzialo.swf w katalogu Rozdział03/Gotowe.

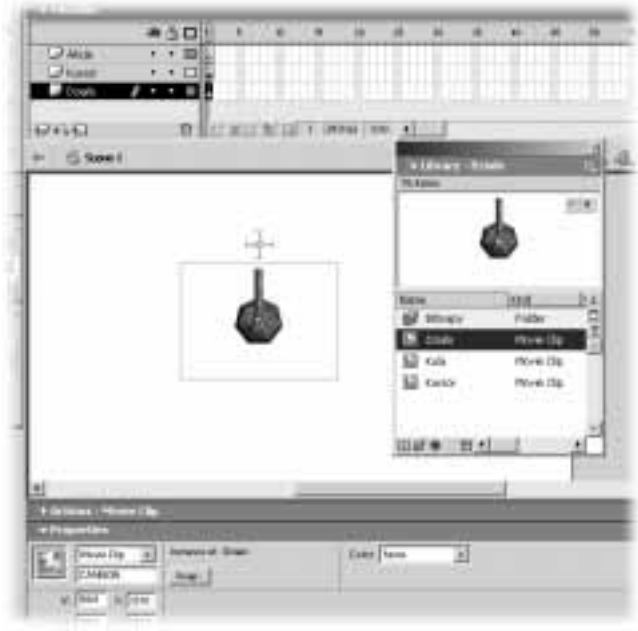
Rysunek 3.8.
Animacja działa obracającego się za kursorem myszy



1. Otwórzmy we Flashu projekt *Dzialo.fla*, który znajduje się w katalogu *Rozdzial03/Cwiczenia*.

   Animacja zawiera trzy warstwy. Na warstwie *Działo* znajduje się odnośnik do symbolu *Dzialo* o nazwie *CANNON*. Na warstwie *Kursor* znajdziemy odnośnik do symbolu *Kursor* o nazwie *Cursor*, który będzie reprezentował kursor myszy w tej animacji. Warstwa *Akcje*, jak łatwo się domyślić, będzie zawierała kod ActionScript kontrolujący animację (patrz rysunek 3.9).

Rysunek 3.9. *Zawartość pliku Dzialo.fla*



   W bibliotece projektu, oprócz dwóch wymienionych wcześniej symboli, znajdziemy jeszcze symbol *Kula*, który reprezentuje pocisk wystrzelony przez działo.

2. Przejdźmy teraz do trybu edycji symbolu *Dzialo*.

   Znajdziemy tam animację klatka po klatce składającą się z 72 ujęć kluczowych, przedstawiającą obrót działa o 360 stopni. W każdej kolejnej klatce działo jest obrócone o 5 stopni (patrz rysunek 3.10).

Rysunek 3.10. *Animacja w symbolu działa składa się z 72 bitmap, z których każda prezentuje działo obrócone o 5 stopni zgodnie z ruchem wskazówek zegara*



Animacja powstała, ponieważ obraz działa został stworzony w programie do modelowania 3D. Obiekt działa był oświetlony źródłem światła, które znajduje się w prawym górnym rogu działa. Aby animacja była w miarę realistyczna, należało stworzyć odpowiednią liczbę obrazków działa dla każdej fazy obrotu przy zachowaniu rozsądnej liczby obrazków.

Gdybyśmy mieli tylko jeden obrazek, który obracalibyśmy dookoła, obracałoby się również źródło światła, co wyglądałoby mało realistycznie.

W kodzie ActionScript, który stworzymy, będziemy musieli uwzględnić tę budowę symbolu *Dzialo*.

3. Powróćmy do animacji głównej, wyselekcjonujmy pierwsze ujęcie kluczowe na warstwie *Akcje*, otwórzmy panel *Actions* i rozpocznijmy wprowadzanie kodu kontrolującego animację działa:

```
//Ukrycie wskaźnika myszki
Mouse.hide();

//Przypisanie celownika do myszy
CURSOR.startDrag(true);
```

Pierwszą rzeczą, jaką zrobimy, będzie stworzenie nowego wizerunku kursora myszy. W grach komputerowych, w których wykorzystywana jest mysz, rzadko kiedy mamy do czynienia z typowym windowsowskim kursorem, czyli strzałką. Z reguły programiści tworzą własny kursor (czasami więcej niż jeden), który jest bardziej odpowiedni w danej sytuacji. Podobnie będzie w naszym przypadku.

Dzięki metodzie `hide` obiektu *Mouse* ukrywamy oryginalny kursor myszy, gdy znajduje się wewnątrz obszaru roboczego animacji. Natomiast dzięki metodzie `startDrag` odnośnika `CURSOR` przypisujemy go do niewidocznego już kursora. Wartość `true` będąca argumentem wywołania metody `startDrag` powoduje, że współrzędne położenia kursora oraz współrzędne położenia odnośnika `CURSOR` są identyczne.

4. Teraz musimy ustalić współrzędne środka obszaru roboczego, ponieważ wszystkie obliczenia wykonywane przez program odnosić się będą do tego punktu. Piszemy więc:

```
//Współrzędne środka obszaru roboczego
SCX = Stage.width / 2;
SCY = Stage.height / 2;
CANNON._x = SCX; CANNON._y = SCY;
```

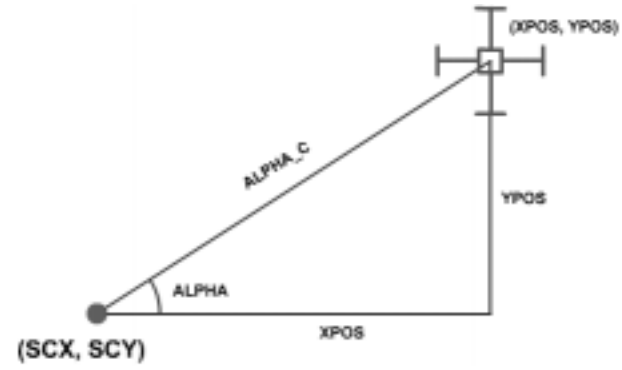
Powyższy kod powinien być jasny. Warto jedynie zaznaczyć, że po ustaleniu współrzędnych środka `SCX` oraz `SCY` przenosimy do niego odnośnik `CANNON`, aby mieć pewność, że nie będzie problemów wynikających z niewłaściwego położenia działa w obszarze roboczym.

5. Teraz przejdziemy do najtrudniejszej części naszego projektu — kodu, który obliczy kąt obrotu działa w zależności od położenia kursora myszy na ekranie. Kod będzie zawarty w metodach reagujących na działania użytkownika wykonywane myszą. Piszemy więc:

```
//Obiekt nasłuchu myszy
Mysza = new Object();

//Metoda wywoływana, gdy poruszymy myszą
Mysza.onMouseMove = function() {
```

Podobnie jak miało to miejsce w przypadku kontroli obiektu za pomocą klawiatury, tworzymy obiekt nasłuchu, który zostanie przypisany do obiektu *Mouse* reprezentującego mysz. Metoda `onMouseMove` obiektu `Mysza` zostanie wywołana, gdy tylko poruszymy myszą, podczas gdy jej wskaźnik będzie znajdował się w obszarze roboczym animacji SWF.

6. W kodzie metody `onMouseMove` musimy obliczyć na podstawie położenia kursora myszy kąt *alfa*, o jaki będziemy chcieli obrócić działo. Do obliczeń wykorzystamy właściwości trójkąta prostokątnego oraz funkcji trygonometrycznych obiektu *Math* (patrz rysunek 3.11):



Rysunek 3.11. *Aby obliczyć kąt alfa, o jaki powinno obrócić się działo, posłużymy się zasadami trygonometrii*

```
//Pozycja X oraz Y wskaźnika myszy
XPOS = _root._xmouse - SCX;   YPOS = _root._ymouse - SCY;

 //Rzeczywista odległość kursora od punktu (SCX, SCY)
ALPHA_C = Math.sqrt((XPOS*XPOS)+(YPOS*YPOS));

 //Kosinus oraz sinus kąta ALPHA
ALPHA_COS = XPOS/ALPHA_C; ALPHA_SIN = YPOS/ALPHA_C;

 //Wartość kąta ALPHA w stopniach
ALPHA_REG = Math.acos(ALPHA_COS)*180/Math.PI;
if (ALPHA_SIN < 0) ALPHA_REG = -ALPHA_REG;
```

Zgodnie z rysunkiem 3.11 do obliczenia kąta musimy znać wartości położenia kursora myszy względem środka obszaru roboczego, czyli wartość `XPOS` oraz `YPOS`, a następnie długość przeciwprostokątnej trójkąta prostokątnego `ALPHA_C`, która jest rzeczywistą odległością (w linii prostej) pomiędzy punktami `(SCX, SCY)` oraz (`XPOS, YPOS`).

Gdy znamy te wartości, obliczamy sinus oraz kosinus kąta *alfa*, po czym korzystając z metody `acos` (*arc cosine*) obiektu *Math*, obliczamy wartość kąta *alfa* w radianach z zakresu od 0 do π (czyli 180 stopni). Pomnożenie otrzymanej wartości przez $180/\pi$ zamieni ją z radianów na stopnie, a my potrzebujemy wartości kąta *alfa* w stopniach.

Dzięki metodzie `acos` obiektu *Math* dowiemy się jedynie, że kąt *alfa* to wartość z przedziału od 0 do 180 bądź też 0 do –180. Aby dokładnie określić przedział, posługujemy się obliczoną wcześniej wartością sinusa kąta *alfa*. Jeśli sinus jest dodatni, to kąt *alfa* pochodzi z przedziału 0 do 180. Jeśli sinus jest ujemny, to *alfa* pochodzi z przedziału –180 do 0. Wszystko jasne. Znamy wartość kąta *alfa*, o jaki powinno obrócić się działo.

7. Jeśli spojrzymy na odnośnik działa, to zobaczymy, że zerowe położenie działa (pierwsze ujęcie kluczowe) odpowiada kątowi –90 stopni. Dzieje się tak dlatego, że kątowi 0° odpowiada punkt położony na osi *X*. Wartości dodatnie kąta obrotu rosną wraz z kierunkiem obrotu wskazówek zegara, ponieważ oś *Y* jest skierowana do dołu (wyższe wartości na osi *Y* ekranu znajdują się niżej niż wartości niższe, punkt 0 osi *Y* to górna krawędź obszaru roboczego).

   Dlatego musimy zmodyfikować obliczoną wartość kąta *alfa*, uwzględniając zerowe położenie działa. Równanie jest bardzo proste i wygląda następująco:

```
//Modyfikacja kąta ALPHA ze względu na animację działa
ALPHA_FINAL = ALPHA_REG + 90;
```

8. Gdybyśmy zamiast animacji zawierającej 72 ujęcia kluczowe, mieli tylko jeden rysunek działa, który obracalibyśmy dookoła, to sam kod animacji byłby trywialny i wyglądałby następująco:

```
CANNON._rotation = ALPHA_FINAL;
```

   Niestety, nie jest to takie proste, ponieważ naszym zadaniem nie będzie obracanie odnośnika, ale zadecydowanie, które ujęcie animacji działa wyświetlić. Musimy przy tym pamiętać, że kolejne ujęcia animacji działa pokazują obiekt obrócony o 5 stopni, a zatem działo będzie skierowane na kursor myszy z dokładnością do 5 stopni. Piszemy więc:

```
//Rendering
if (ALPHA_SIN > 0)
{
      //Ćwiartka 1 oraz 2
      for (iter = 19; iter < 55; iter++)
      if (iter*5 > ALPHA_FINAL)
      {
            CANNON.gotoAndStop(iter);
            break;
      }
```

   Powyższy kod dotyczy pierwszej oraz drugiej ćwiartki układu współrzędnych o środku w punkcie `(SCX, SCY)`. Kod sprawdza po prostu, w którym z 5-stopniowych przedziałów obrotu znajduje się wartość kąta *alfa* i wyświetla ujęcie w odnośniku `CANNON` o odpowiednim numerze. Wartość zmiennej `iter` bezpośrednio określająca numer ujęcia do wyświetlenia została dobrana w ten sposób, ponieważ tylko te ujęcia wyświetlają działo, którego kąt obrotu znajduje się w przedziale 0 – 180 stopni.

   Pora na ćwiartkę trzecią i czwartą:

```
} else {
      for (iter = 1; iter < 19; iter++)
      if (ALPHA_FINAL > 0)
      {
            //Ćwiartka 4
            if (iter*5 > ALPHA_FINAL)
            {
                  CANNON.gotoAndStop(iter);
                  break;
            }
```

```
        } else {
            //Ćwiartka 3
            if(-iter*5 < ALPHA_FINAL)
            {
                CANNON.gotoAndStop(72-iter);
                break;
            }
        }
    }
} //Koniec metody onMouseMove
```

W przypadku ćwiartki czwartej kod wygląda bardzo podobnie do omówionego poprzednio, natomiast ćwiartkę trzecią traktujemy jak lustrzane odbicie ćwiartki czwartej. Funkcja obracająca działo w kierunku kursora myszy jest już gotowa.

9. Aby zobaczyć, jak działa stworzony kod, powinniśmy obiekt nasłuchu `Mysza` przyłączyć do obiektu `Mouse` reprezentującego mysz:

```
//Przypisanie obiektu nasłuchu do myszy
Mouse.addListener(Mysza);
```

Gotowe. Możemy teraz opublikować animację, aby przekonać się, czy działa poprawnie. Przesuwając kursor myszy po ekranie, będziemy mogli zobaczyć, że działo rzeczywiście obraca się za nim. Jeśli będziemy kursor przesuwać bardzo powoli, zobaczymy również, że działo nie obraca się dokładnie, lecz ze skokiem 5-stopniowym.

W pliku *Dzialo_FazaI.fla* znajdującym się w katalogu *Rozdzial03/Gotowe* znajdziemy animację źródłową Flasha MX, która powinna powstać po wykonaniu powyższego ćwiczenia.

Ćwiczenie 3.4.

Rozpocznijmy teraz drugą część projektu, czyli umożliwienie użytkownikowi strzelania z działa po naciśnięciu lewego przycisku myszy. Koncepcja jest następująca: jedno naciśnięcie lewego przycisku myszy powoduje wystrzelenie pocisku lecącego w kierunku punktu, w którym znajduje się kursor myszy. Gdy pocisk opuszcza widoczną część obszaru roboczego, zostaje usunięty.

1. Kontynuujemy pracę z projektem, który wykonaliśmy w poprzednim ćwiczeniu. Wyselekcjonujmy więc pierwsze ujęcie kluczowe na warstwie *Akcje* i otwórzmy panel *Actions*.

   Najpierw musimy stworzyć kilka zmiennych kontrolujących liczbę pocisków, które możemy wystrzelić z działa, oraz ich zachowanie. Umieśćmy więc kursor tekstowy na samym początku kodu i piszmy:

```
//Zmienna określająca maksymalną liczbę strzałów
SHOOT_MAX = 10;

//Zmienna określająca aktualną liczbę strzałów
SHOOT_ACTUAL = 0;

//Zmienna określająca współczynnik prędkość pocisku
SHOOT_SPEED = 0.2;
```

Zmienna SHOOT_MAX nie określa tego, ile razy działo może wystrzelić, ale ile pocisków jednocześnie może znajdować się na ekranie. Musimy zdawać sobie sprawę z tego, że pociski będziemy tworzyć poleceniem attachMovie. Określamy w nim między innymi warstwę, na której znajdzie się stworzony odnośnik. Numer tej warstwy nie może kolidować z warstwami innych obiektów stworzonych za pomocą ActionScript.

Teoretycznie moglibyśmy każdy nowo utworzonym pocisk umieszczać na nowej warstwie, ale jeśli projekt ma być grą, która zawierać będzie oprócz działa i pocisków także inne elementy graficzne, to szybko mogłoby dojść do konfliktu, gdyby pocisk usunął obiekt stworzony wcześniej. Z drugiej strony, prędkość, z jaką lecą pociski, wyrażona współczynnikiem SHOOT_SPEED, powoduje, że na ekranie będzie znajdować się jednocześnie z góry określona liczba pocisków, więc możemy spokojnie ją ograniczyć. Gdy jakiś pocisk zniknie z ekranu i zostanie usunięty, jego warstwa jest znów wolna i można ją wykorzystać dla nowych pocisków.

Aktualną warstwę, na której powstanie nowy pocisk, określa zmienna SHOOT_ACTUAL.

2. Teraz umieśćmy kursor tekstowy poniżej kodu metody onMouseMove obiektu Mysza, ale przed kodem: Mouse.addListener(Mysza); i zacznijmy pisać:

```
//Kod metody wywoływany, gdy wciśniemy lewy przycisk myszy
Mysza.onMouseDown = function () {
```

Oczywiście, jak łatwo się domyślić, cały kod generujący i kontrolujący pociski zostanie umieszczony wewnątrz metody onMouseDown, która jest wywoływana, gdy tylko wciśniemy lewy przycisk myszy.

```
//Stworzenie nowego odnośnika z symbolu o ID Bullet
_root.attachMovie("Bullet", "Shoot"+SHOOT_ACTUAL,
                    10+SHOOT_ACTUAL);
```

Powyższy wiersz kodu tworzy nowy odnośnik na bazie symbolu *Movie Clip*, którego identyfikator eksportu został ustalony na *Bullet* (o tym identyfikatorze opowiemy dokładniej za chwilę). Nazwą odnośnika, który zostanie stworzony, będzie *Shoot-NumerWarstwy*, czyli np. *Shoot0*, *Shoot1*, *Shoot2* itd., a warstwę, na której znajdzie się kula, określamy prostym równaniem 10+SHOOT_ACTUAL. Wartość stała w tym równaniu zapewnia, że nie będzie problemów, jeśli na warstwach Flasha MX w procesie edycji animacji umieścimy więcej obiektów niż w naszym projekcie.

Nowy odnośnik musimy odpowiednio ułożyć w obszarze roboczym:

```
//Ustalenie położenia nowego pocisku
eval("Shoot"+SHOOT_ACTUAL)._x = SCX+ALPHA_COS*65;
eval("Shoot"+SHOOT_ACTUAL)._y = SCY+ALPHA_SIN*65;
```

Będzie się on znajdował na okręgu o środku w punkcie (SCX, SCY) oraz o promieniu 65 jednostek. O dokładnym położeniu zdecydują wartość ALPHA_COS oraz ALPHA_SIN, które, jak pamiętamy, obliczamy przy każdym wywołaniu metody onMouseMove.

Pozostało jeszcze stworzenie zmiennych, które określą prędkość pionową oraz poziomą, z jaką będzie poruszał się pocisk po wystrzeleniu:

```
//Ustalenie prędkości poziomej oraz pionowej nowego pocisku
eval("Shoot"+SHOOT_ACTUAL).XSPEED = SHOOT_SPEED*ALPHA_COS*65;
eval("Shoot"+SHOOT_ACTUAL).YSPEED = SHOOT_SPEED*ALPHA_SIN*65;
```

3. Wszystkie dane, których potrzebuje pocisk do poprawnej animacji, zostały już obliczone. Musimy teraz stworzyć mechanizm, dzięki któremu pocisk będzie przesuwał sam siebie i sam będzie wiedział, kiedy się zniszczyć.

   Nic prostszego, jeśli będziemy pamiętali, że pocisk to odnośnik do symbolu *Movie Clip*, a więc może posiadać własne metody. Skorzystamy tutaj z metody `onEnterFrame`, której zaletą jest to, że jej kod jest wykonywany za każdym razem, gdy Flash Player wyświetli nową klatkę animacji. Piszemy więc:

```
//Metoda animująca odnośnik wywoływana w każdej
//klatce animacji
eval("Shoot"+SHOOT_ACTUAL).onEnterFrame = function() {

      //Przesunięcie odnośnika
      this._x += this.XSPEED; this._y += this.YSPEED;

      //Usunięcie odnośnika, jeśli opuścił widoczny obszar
      if ((this._x < 0) || (this._x > Stage.width) ||
            (this._y < 0) || (this._y > Stage.height))
            this.removeMovieClip();
} //Koniec metody onEnterFrame
```

   Jak widać, kod metody `onEnterFrame` stworzonego wcześniej odnośnika przesuwa go w odpowiednim kierunku, po czym, jeśli pocisk opuścił obszar „dozwolony", usuwa go z animacji.

4. Pozostało jeszcze dodanie kodu kontrolującego wartość zmiennej `SHOOT_ACTUAL`, określającej warstwę, na której powstanie nowy pocisk. Ten kod wygląda następująco:

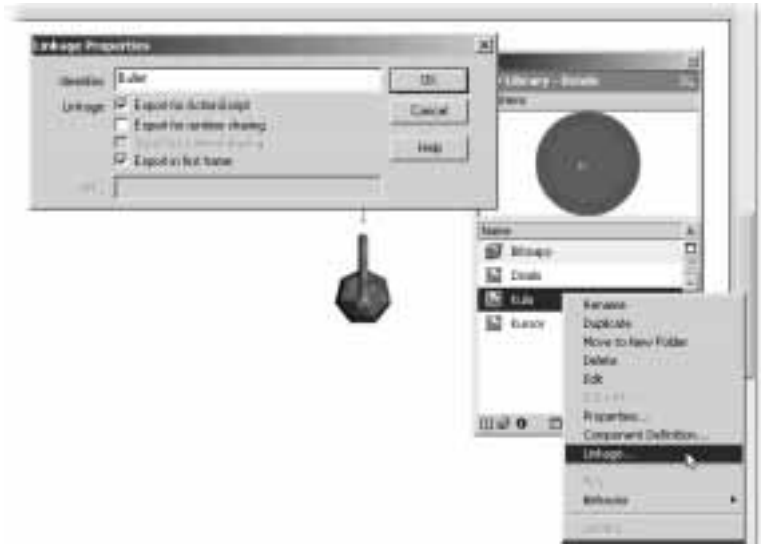
```
//Zwiększenie numeru identyfikującego pocisk
SHOOT_ACTUAL++;

//Sprawdzenie, czy nie wystrzelono określonej liczby pocisków
//Jeśli tak, to numery identyfikujące pociski
//zaczynamy od zera
if (SHOOT_ACTUAL > SHOOT_MAX) SHOOT_ACTUAL = 0;

} //Koniec metody onMouseDown
```

   Kod ActionScript jest już gotowy. Musimy jeszcze określić identyfikator symbolu *Kula*, który jest pociskiem. Jest to konieczne, jeśli w obszarze roboczym chcemy wstawić odnośnik do symbolu, który nie ma innego odnośnika w obszarze roboczym (gdyby był inny odnośnik, moglibyśmy skorzystać z metody `duplicateMovieClip` obiektu *MovieClip*).

   Otwórzmy bibliotekę naszego projektu, a następnie dla symbolu *Kula* wyświetlmy podręczne menu i wybierzmy z niego polecenie *Linkage* (patrz rysunek 3.12). Gdy pojawi się okno *Linkage Properties,* uaktywnijmy opcję *Export for ActionScript* i w polu tekstowym *Identyfier* wpiszmy: `Bullet` (patrz rysunek 3.12). Ten identyfikator będziemy również nazywać nazwą eksportową symbolu.

Projekt jest już gotowy i po jego opublikowaniu powinniśmy otrzymać animację, która wygląda jak na rysunku 3.8. Gotową animację możemy znaleźć w pliku *Dzialo.fla*, w katalogu *Rozdzial03/Gotowe*.

Rysunek 3.12.
Jeśli w kodzie ActionScript chcemy używać metody attachMovie, musimy określić identyfikator symbolu, którego operacja ma dotyczyć



Podsumowanie

Jak mogliśmy się przekonać w tym rozdziale, sterowanie postaciami może być zadaniem bardzo prostym, jak w przypadku rakiety, lub złożonym, jak w przypadku bałwana, oraz całkiem skomplikowanym jak w przypadku działa (oczywiście dla niektórych osób kolejność bałwan, działo niekoniecznie musi być prawdziwa).

Projekty, które omówiliśmy w tym rozdziale, nie wyczerpują bynajmniej wszystkich możliwości, z jakimi możemy mieć do czynienia. Dobrze oddają jednak naturę problemów, z którymi przyjdzie się nam zmierzyć podczas tworzenia własnych projektów.

Zapraszamy do następnego rozdziału „Tworzenie plansz w grach", w którym poznamy sposoby budowy plansz gier, po których będą poruszać się tworzone przez nas postacie.