
Helion
Paweł Zakrzewski
Adobe Flash
CS6
i ActionScript 3.0
Interaktywne projekty od podstaw

Redaktor prowadzący: Michał Mrowiec

Projekt okładki: Anna Mitka

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

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie?flcs5i
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

ISBN: 978-83-246-3865-9

Printed in Poland.

Spis treści

Rozdział 8.
Budujemy gry i interaktywne zabawki

Adobe Flash to doskonałe środowisko dla tworzenia różnego typu interaktywnych zabawek oraz gier. Łatwość użycia, wielkie możliwości oraz ActionScript 3.0 sprawiają, że z technologii Flash korzystają niemal wszyscy. Zarówno ci najbardziej doświadczeni programiści, jak i osoby, które dopiero rozpoczynają swoją działalność developerską. W rozdziale tym chciałbym przedstawić kilka prostych projektów — zabawek, gier, które pozwolą, szczególnie osobom początkującym, poznać wspaniałe możliwości, jakie oferuje z jednej strony program Flash, a z drugiej ActionScript 3.0. Każdy z przedstawionych tu przykładów wykorzystuje coraz bardziej zaawansowane techniki wykorzystania kodu oraz możliwości graficznych, jakie oferuje Flash. Jak to zwykle bywa, teoretyczny wstęp wnosi niewiele, przejdźmy więc do konkretnych przykładów.

Flashowe rysowanie

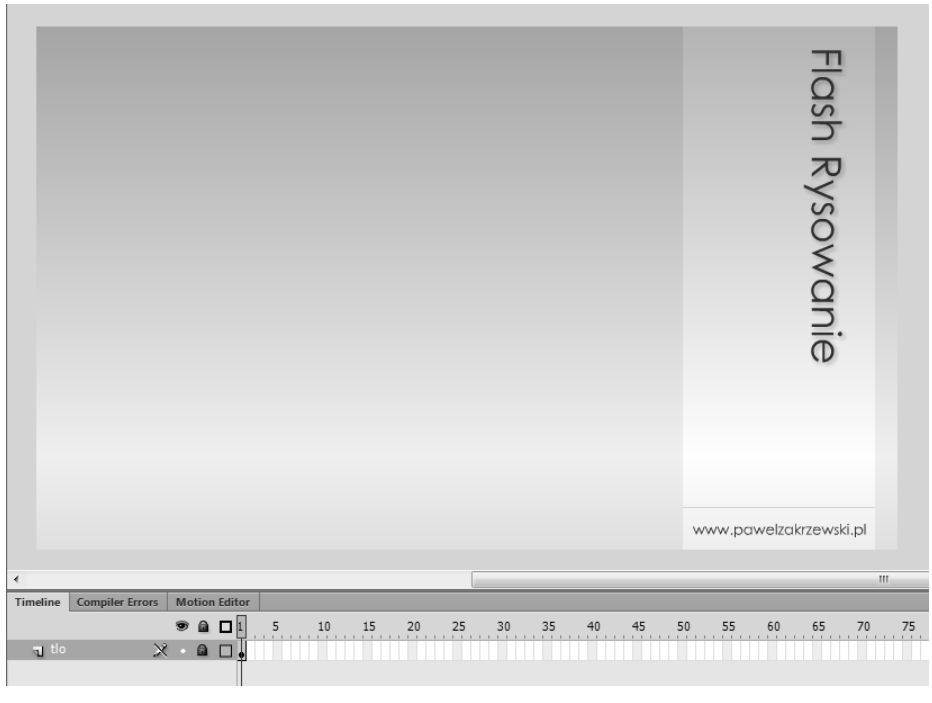
Pierwszym projektem z serii interaktywnych zabawek będzie tym razem prosty program rysunkowy. Przygotujemy tu niemal klasyczny przykład tablicy, na której można będzie rysować dowolne odręczne figury i kształty. Aby całość przypominała nieco programy rysunkowe, dodamy także możliwość zarządzania kolorem i grubością linii oraz oczywiście czyszczenia całości.

W naszej pracy wykorzystamy dwa poznane wcześniej obiekty: `Sprite` (czyli taki `MovieClip` bez osi czasu) oraz `Graphics`, czyli środowisko tworzenia elementów wektorowych języka ActionScript. Do zarządzania kolorem oraz grubością linii użyjemy gotowych komponentów. A zatem do dzieła!

Tworzymy interfejs graficzny

Najogólniej rzecz ujmując, cały wygląd naszej pracy nie ma tu wielkiego znaczenia. Budujemy nowy dokument o dowolnych proporcjach oraz wielkości i ewentualnie ustalamy kolor sceny. Jeśli jednak zależy nam na atrakcyjnej formie, która pozwoli przyciągnąć i zatrzymać choć na chwilkę młodego użytkownika, warto zadbać o przygotowanie prostego tła, dodanie niezbędnych logotypów, elementów graficznych czy choćby nazwy naszej zabawki.

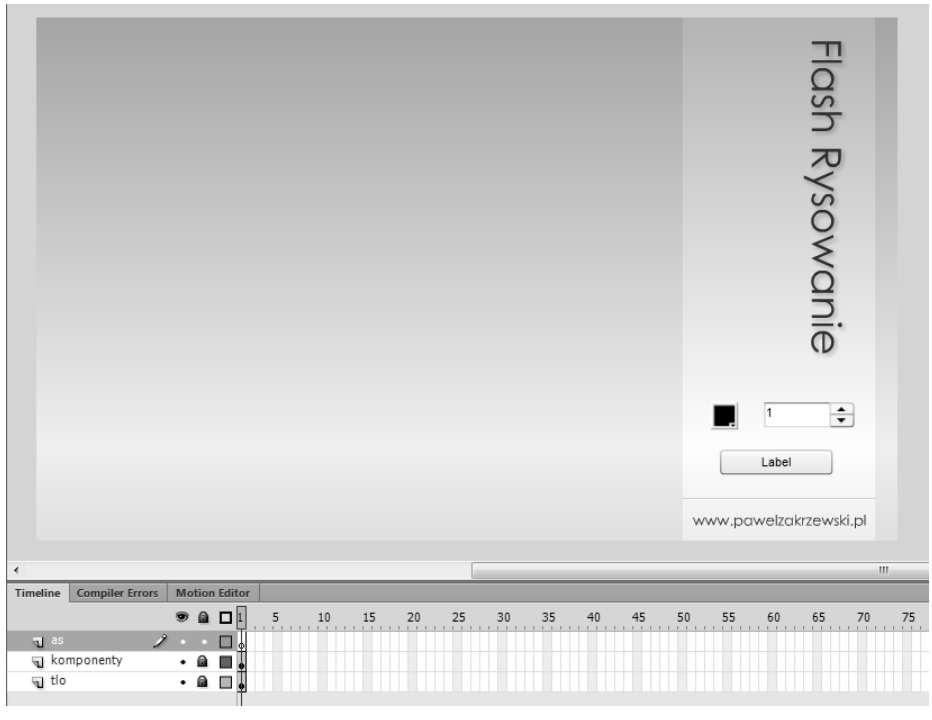
Wszystkie niezbędne elementy możemy przygotować, korzystając z klasycznych technik programu Flash. Na osobnych warstwach rozmieszczamy obiekty o podobnym charakterze i gotową całość blokujemy (rysunek 8.1). Na osobnej warstwie dodamy elementy sterujące, czyli komponenty.



Rysunek 8.1. *Wszystkie niezbędne elementy możemy przygotować, korzystając z klasycznych technik programu Flash. Na osobnych warstwach rozmieszczamy obiekty o podobnym charakterze i gotową całość blokujemy*

W tym celu przechodzimy do palety *Components* (*Ctrl+F7*) i przeciągamy na osobną warstwę instancję komponentów `Button`, `ColorPicker` oraz `NumericStepper`. Pierwszy z nich, coś a' la gumka, pozwoli na wyczyszczenie całej przygotowanej ilustracji, drugi na zmianę koloru rysowania, zaś ostatni — `NumericStepper` na zarządzanie grubością linii.

Aby możliwe było użycie komponentów za pomocą kodu, konieczne są nazwy ich instancji. Korzystając z przyjętego wcześniej schematu, zaznaczamy instancję komponentu `ColorPicker` i nadajemy jej nazwę `CPKolor`. W przypadku komponentu `NumericStepper` wykorzystamy nazwę `NSGrubosc`, zaś `Button` nazywamy po prostu `BReset`. Warstwę, na której znajdują się komponenty, blokujemy, dodajemy także kolejną, nadając jej nazwę *as*. Tu znajdzie się cały kod sterujący. Wbrew pozorom nie będzie go zbyt dużo (rysunek 8.2).



Rysunek 8.2. *Warstwę, na której znajdują się komponenty, blokujemy, dodajemy także kolejną, nadając jej nazwę „as"*

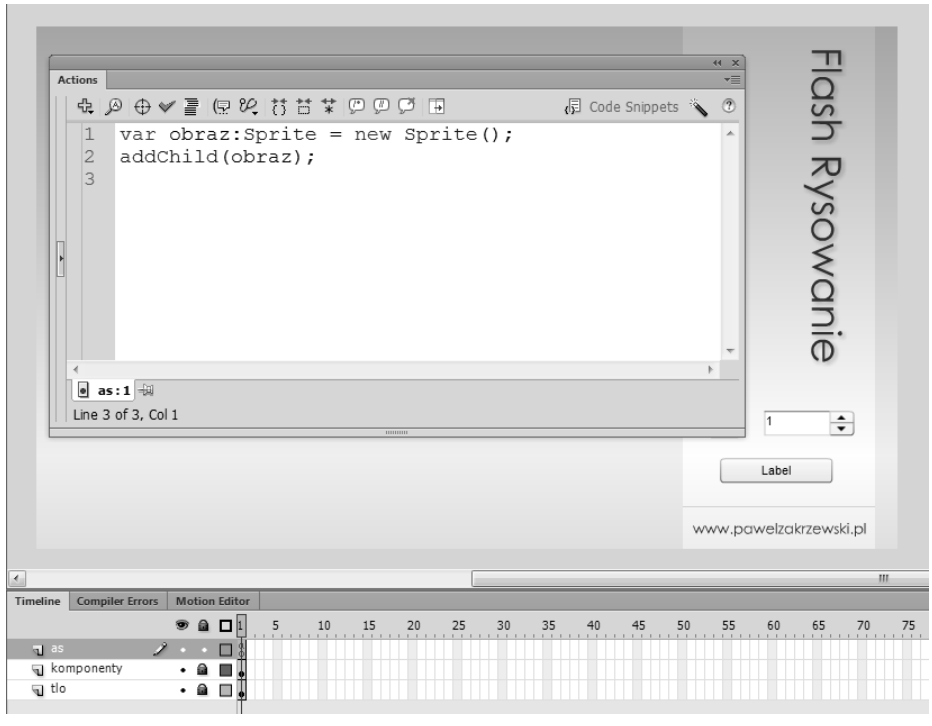
Rysowanie za pomocą kodu ActionScript

Podstawowym elementem odpowiedzialnym za rysowanie obiektów wektorowych jest klasa zwana `Graphics`. To jej metody typu `drawRect()`, `drawCircle()` czy też `curveTo()` lub `lineTo()` pozwalają nam tworzyć proste kształty wektorowe. Aby jednak możliwe było zarządzanie tak przygotowanym elementem, musi on być zamknięty w nadrzędnym obiekcie typu `Sprite` lub `MovieClip`.

Ujmując to od drugiej strony, aby możliwe było wykorzystanie metod rysunkowych, musimy wcześniej przygotować obiekt, czyli kontener, który będzie zawierał cały przygotowany kształt. Najlepiej sprawdza się w tej roli obiekt typu `Sprite`.

Oferuje on wielką funkcjonalność zbliżoną do tej, jaką daje nam `MovieClip`, jednak nie posiada wewnętrznej, niezależnej osi czasu. Nie jest to żaden problem, ponieważ podczas tworzenia naszej rysowanki nie będziemy korzystali z żadnych metod typu `nextFrame()` czy `gotoAndPlay()`. Potrzebny nam jest jedynie kontener, który pozwoli narysować proste kształty wektorowe. `Sprite` nadaje się do tego doskonale.

Klasa `Sprite` nie posiada jednak interfejsu graficznego. Oznacza to, że wszelkie instancje musimy budować, korzystając jedynie z kodu ActionScript. Na szczęście nie jest to trudne. Wprowadzając dwie linie kodu, mamy już instancję klasy `Sprite` na scenie. Nadaliśmy jej nazwę — `obraz` (rysunek 8.3).



Rysunek 8.3. *Klasa Sprite nie posiada jednak interfejsu graficznego. Oznacza to, że wszelkie instancje musimy budować, korzystając jedynie z kodu ActionScript*

```
var obraz:Sprite = new Sprite();
addChild(obraz);
```

Oczywiście, jeśli teraz zdecydujemy się na podgląd efektów naszej pracy, zobaczymy… No właśnie — nic nie zobaczymy. Utworzyliśmy bowiem pusty obiekt typu `Sprite`, który w tej chwili nie zawiera żadnej zawartości.

Aby na wybranej powierzchni naszej pracy możliwe było rysowanie, musimy określić jej rozmiar i kształt. W tym celu najwygodniej będzie narysować prostokąt wektorowy i wypełnić go dowolnym kolorem — tłem, czyli tak jakby kartką do

rysowania. Zwykle większość tego typu prac pozwala na rysowanie różnych kształtów na białym tle, toteż i my pokusimy się o przygotowanie takiego właśnie obszaru. Nie będzie jednak problemem, jeśli chcielibyśmy malować na innym tle. Wprowadzając jako parametr metody `beginFill()` numer innego koloru, możemy dowolnie określić kolorystykę naszego tła roboczego. W naszym przykładzie narysowałem kwadratowy, biały obiekt o wielkości 400 na 400 pikseli usytuowany w lewym górnym narożniku sceny (rysunek 8.4). Posłuży on jako nowa kartka do tworzenia naszej ilustracji.

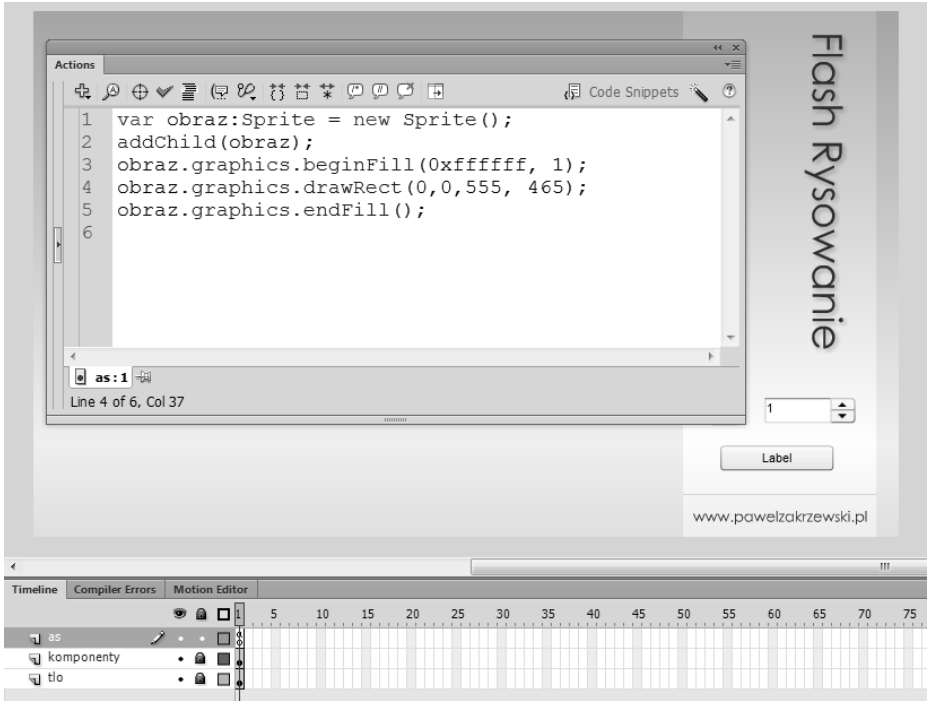
Rysunek 8.4. *Wprowadzając jako parametr metody beginFill() numer innego koloru, możemy dowolnie określić kolorystykę naszego tła roboczego*

```
var obraz:Sprite = new Sprite();
addChild(obraz);
obraz.graphics.beginFill(0xffffff, 1);
obraz.graphics.drawRect(0,0,400, 400);
obraz.graphics.endFill();
```

Oczywiście wielkość obszaru roboczego to tylko przykład, a my możemy zmienić jego wymiary tak, aby lepiej pasował do naszej pracy. W moim przykładzie użyłem nieco większego obszaru o powierzchni 555 na 465 pikseli. (rysunek 8.5).

Podstawą do rysowania za pomocą kodu są trzy proste metody klasy `Graphics`:

- `moveTo()` — metoda pozwalająca na określenie punktu, od którego rozpoczynamy rysowanie.
- `lineTo()` — metoda, która pozwala na rysowanie linii prostej z określonej wcześniej lokalizacji do konkretnego punktu na scenie
- `curveTo()` — to metoda, która pozwala na kreślenie krzywych z bieżącej lokalizacji do określonego punktu na scenie.



Rysunek 8.5. *W moim przykładzie użyłem nieco większego obszaru o powierzchni 555 na 465 pikseli*

W naszym przykładzie skorzystamy jedynie z dwóch pierwszych. Metoda `moveTo()` pozwoli określić punkt, w którym rozpoczynamy tworzenie kolejnego obiektu, zaś `lineTo()` ułatwi konkretne rysowanie. Jak wcześniej wspominałem, cała praca tworzona będzie wewnątrz przygotowanego wcześniej Sprite'a o nazwie `obraz`.

Mimo że cały przykład nie jest trudny, wymaga jednak specjalnego przygotowania. Użytkownik, wciskając klawisz myszki, rozpoczyna rysowanie w miejscu, gdzie znajduje się kursor myszki. Jak się łatwo domyślić, wykorzystamy tu metodę `moveTo()`. Następnie korzystając z metody `lineTo()`, będziemy z możliwie dużą częstotliwością rysowali krótkie, proste odcinki, które pozwolą na tworzenie dowolnej ilustracji. Rzecz jasna, w tym przypadku skorzystamy z metody `lineTo()`, przemieszczając punkt docelowy w nowe położenie określone na podstawie współrzędnych kursora myszki.

Mówiąc inaczej, w chwili gdy wciskamy myszkę, cały czas odbywa się rysowanie. Gdy zwalniamy klawisz myszy, program przenosi nas w nowe położenie, tak aby w chwili ponownego wciśnięcia znów rozpocząć rysowanie w nowym punkcie. Kluczową rolę odgrywa tu detekcja wciśnięcia lub zwolnienia klawisza myszki. Kiedy jednak odbywa się samo rysowanie?

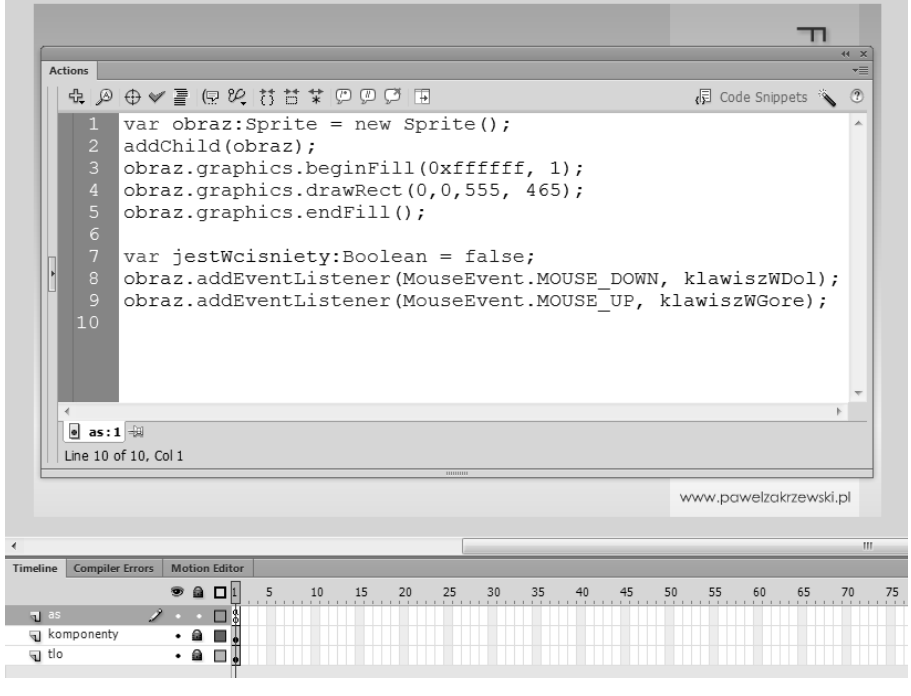
To może nieco zaskakujące, jednak do płynnego rysowania z dużą częstotliwością (duża, bardzo duża częstotliwość powtarzania — jakie zdarzenie generuje

takie działanie?) wykorzystamy tu zdarzenie `ENTER_FRAME` i proste instrukcje warunkowe. Tyle teorii. Spróbujmy teraz przetestować kolejne fragmenty kodu w praktyce. Rozpoczynamy od detekcji wciśnięcia i zwolnienia klawisza myszki. Do przechowywania tego stanu wykorzystamy dodatkową zmienną o charakterze logicznym tzw. `Boolean`. Nadamy jej nazwę `jestWcisniety`.

Podczas użycia zmiennych typu `Boolean` najlepiej jest nadawać im nazwy w formie pytań typu: `czyGram`, `jestWcisniety`, `jestAktywny` itp. Tego typu nazwa jednoznacznie informuje o typie zmiennej i ułatwia późniejsze czytanie oraz edycję kodu.

Ponieważ w chwili uruchomienia naszej zabawki klawisz myszki nie jest od razu wciśnięty, wartość zmiennej przyjmuje stan `false`. W chwili wciśnięcia klawisza myszki zmieni się on na `true`.

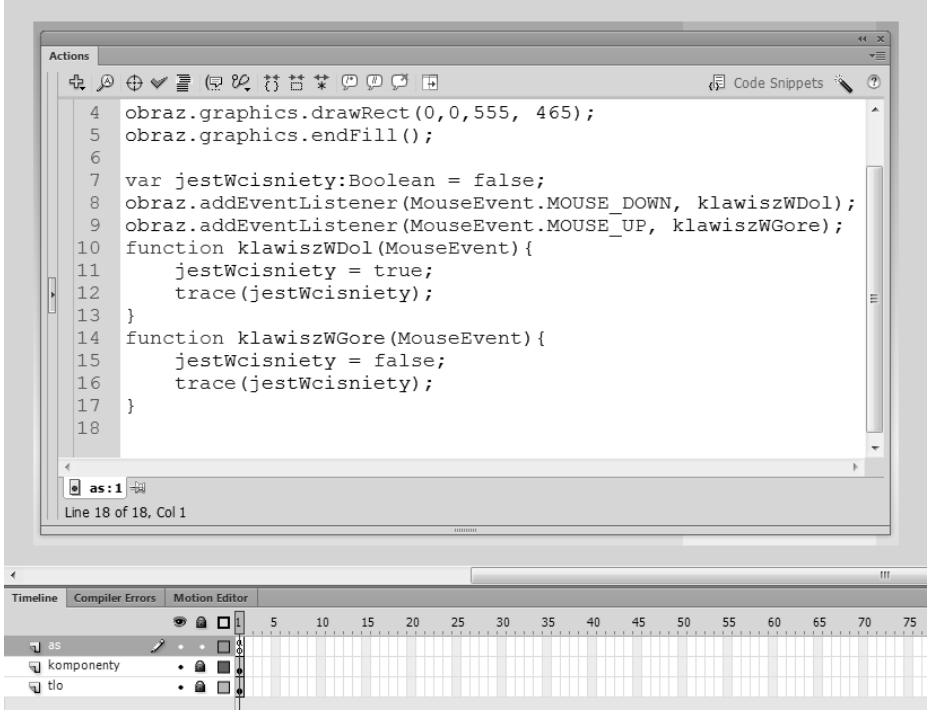
Aby określić, czy kursor myszki znajduje się ponad obszarem do malowania i czy został właśnie wciśnięty, użyjemy znanego zdarzenia `MOUSE_DOWN`, przypisując nasłuchiwanie do przygotowanego wcześniej Sprite'a o nazwie `obraz`. W podobny sposób, korzystając tym razem ze zdarzenia `MOUSE_UP`, spróbujemy sprawdzić, kiedy klawisz myszki został zwolniony. Wykorzystując tu obiekt `obraz`, ograniczymy automatycznie możliwość malowania tylko do obszaru Sprite'a (rysunek 8.6).



Rysunek 8.6. *Aby określić, czy kursor myszki znajduje się ponad obszarem do malowania i czy został właśnie wciśnięty, użyjemy znanego zdarzenia MOUSE_DOWN, przypisując nasłuchiwanie do przygotowanego wcześniej Sprite'a o nazwie obraz. W podobny sposób, korzystając tym razem ze zdarzenia MOUSE_UP, spróbujemy sprawdzić, kiedy klawisz myszki został zwolniony*

```
var jestWcisniety:Boolean = false;
obraz.addEventListener(MouseEvent.MOUSE_DOWN, klawiszWDol);
obraz.addEventListener(MouseEvent.MOUSE_UP, klawiszWGore);
```

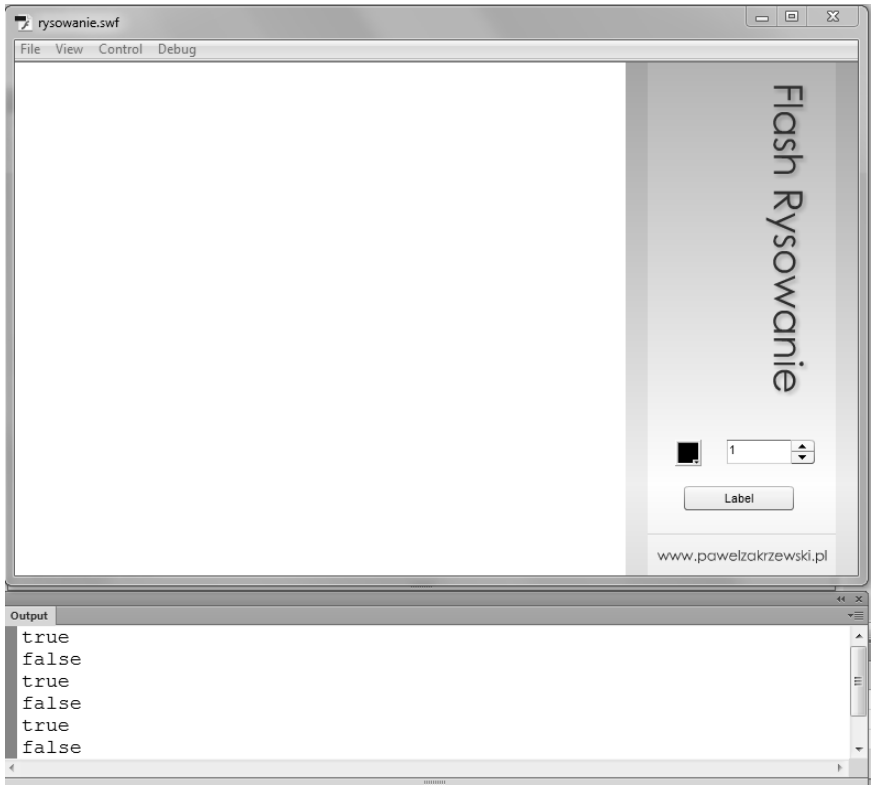
Naturalnie, aby taki kod mógł zostać użyty, konieczne jest przygotowanie dwóch funkcji do obsługi zdarzeń. W obu przypadkach skupimy się tylko na określaniu stanu wartości zmiennej `jestWcisniety` (rysunek 8.7).



Rysunek 8.7. *Naturalnie, aby taki kod mógł zostać użyty, konieczne jest przygotowanie dwóch funkcji do obsługi zdarzeń. W obu przypadkach skupimy się tylko na określaniu stanu wartości zmiennej „jestWcisniety"*

```
obraz.addEventListener(MouseEvent.MOUSE_DOWN, klawiszWDol);
obraz.addEventListener(MouseEvent.MOUSE_UP, klawiszWGore);
function klawiszWDol(MouseEvent){
  jestWcisniety = true;
  trace(jestWcisniety);
}
function klawiszWGore(MouseEvent){
  jestWcisniety = false;
  trace(jestWcisniety);
}
```

Aby sprawdzić, czy wprowadzony kod działa, możemy wykorzystać tymczasowo metodę `trace()`. W rezultacie działania przygotowanego tu kodu w oknie *Output* w chwili wciśnięcia klawisza myszki powinien ukazać się komunikat w postaci `true` , zaś w momencie jej zwolnienia — komunikat `false` (rysunek 8.8).

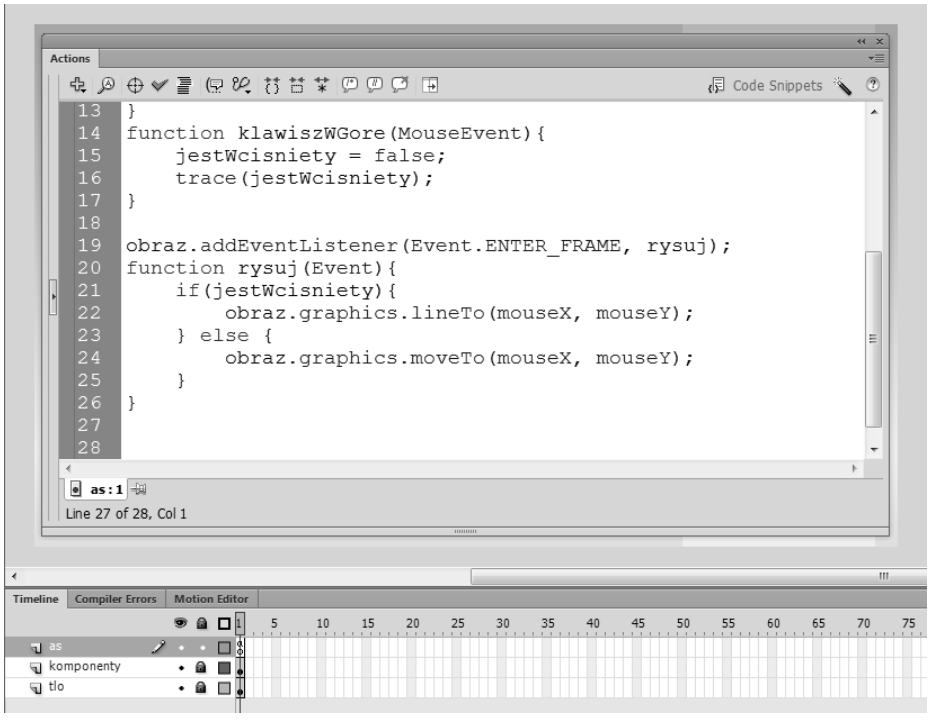


Rysunek 8.8. *W rezultacie działania przygotowanego tu kodu w oknie Output w chwili wciśnięcia klawisza myszki powinien ukazać się komunikat w postaci true, zaś w momencie jej zwolnienia — komunikat false*

Jak wspominałem, w tym miejscu skupiamy się wyłącznie na detekcji wciśnięcia klawisza myszki. Za rysowanie odpowiadać będzie nowa funkcja wywoływana na podstawie zdarzenia `ENTER_FRAME`.

W chwili gdy użytkownik wciska klawisz myszki, ma rozpocząć się rysowanie. Użyjemy tu oczywiście metody `lineTo()`, która pozwoli kreślić dowolny kształt na bazie położenia kursora myszki. W momencie gdy zwolniony zostanie klawisz myszki, wówczas będzie wywoływana metoda `moveTo()`, która pozwoli przenieść punkt początkowy w nowe położenie. Warto zwrócić uwagę na wyrażenie: „Gdy wciskamy, to… W przeciwnym razie to…”. Czy nie jest to klasyczny przykład instrukcji warunkowej `if/else`? Naturalnie! A jak wygląda to w naszym kodzie (rysunek 8.9)?

```
obraz.addEventListener(Event.ENTER_FRAME, rysuj);
function rysuj(Event){
  if(jestWcisniety){
    obraz.graphics.lineTo(mouseX, mouseY);
  } else {
    obraz.graphics.moveTo(mouseX, mouseY);
  }
}
```



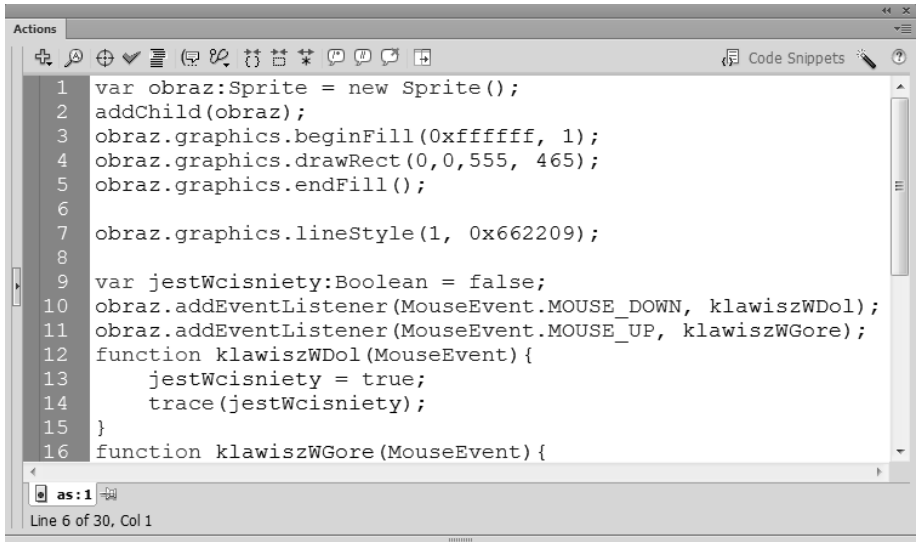
Rysunek 8.9. *Warto zwrócić uwagę na wyrażenie: „Gdy wciskamy, to… W przeciwnym razie to…". Czy nie jest to klasyczny przykład instrukcji warunkowej if/else? Naturalnie! A jak wygląda to w naszym kodzie?*

Dodając powyższy kod, tworzymy prostą strukturę: „Na każde wejście do klatki (można to określić w każdej chwili), gdy jest wciśnięty klawisz myszki, rysuj linię do punktu określonego współrzędnymi kursora myszki na scenie. Gdy klawisz myszki jest zwolniony, także dopasuj punkt początkowy nowego rysowania do współrzędnych kursora myszki". Niby wszystko jest przygotowane poprawnie, jednak cały przykład nie działa. Co może być tu problemem? Sprawa jest całkiem prosta. Jeśli chcielibyśmy w naszym domu pomalować ściany, najpierw rozważamy możliwość wykorzystania konkretnego koloru, a dopiero później wprowadzamy nasz plan w życie. Podobnie jest w programie Flash. Aby narysować linię, musimy określić jej cechy — kolor, grubość czy też stopień krycia. Tego brakuje w naszym przykładzie.

Docelowe elementy określające atrybuty linii określać będziemy za pomocą przygotowanych komponentów, jednak w pierwszej chwili sprawdźmy, czy cały przykład po prostu działa. Później dodamy obsługę komponentów.

Rozpoczynamy od określenia stylu linii, korzystając z metody `lineStyle()`. W zasadzie w dowolnym miejscu, jednak już po utworzeniu instancji klasy `Sprite`, należy dodać pojedynczą instrukcję (rysunek 8.10).

```
obraz.graphics.lineStyle(1, 0x662209);
```



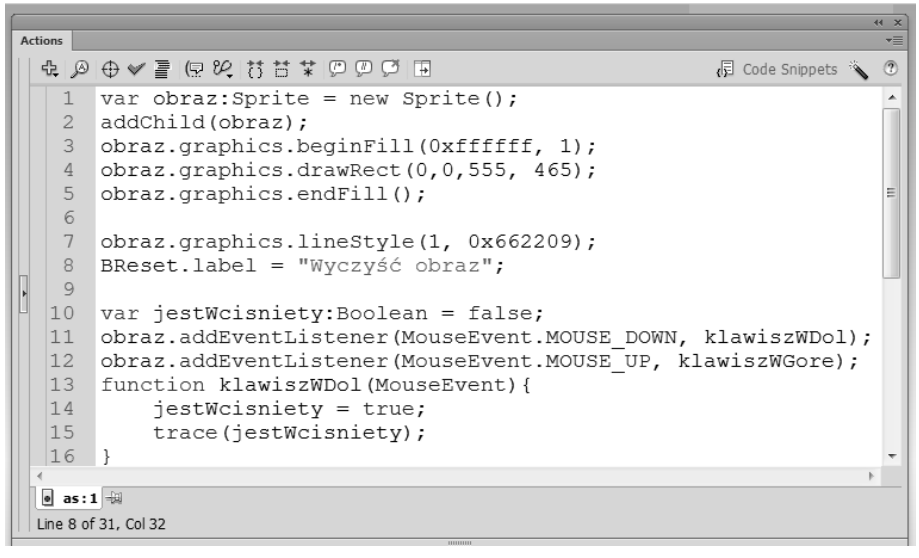
Rysunek 8.10. *Rozpoczynamy od określenia stylu linii, korzystając z metody lineStyle(). W zasadzie w dowolnym miejscu, jednak już po utworzeniu instancji klasy Sprite, należy dodać pojedynczą instrukcję*

W tej chwili cały przykład działa poprawnie, a my możemy testować różne grubości i kolory naszej linii (rysunek 8.11). Pierwszy parametr metody `lineStyle()` odpowiada za grubość, drugi za kolor, zaś kolejny — trzeci (posiada wartość domyślną `1`) za poziom krycia linii. Zmieniając dowolnie owe parametry, możemy całkiem łatwo sterować atrybutami rysowanej kreski.

Rysunek 8.11. *W tej chwili cały przykład działa poprawnie, a my możemy testować różne grubości i kolory naszej linii*

Wykorzystujemy komponenty

Aby z powodzeniem wykorzystać nasze komponenty do określenia parametrów linii, konieczne jest odpowiednie skonfigurowanie ich właściwości. Rozpoczynamy od określenia etykiety przycisku. Korzystając z jego nazwy instancji, możemy wprowadzić prosty fragment (rysunek 8.12).



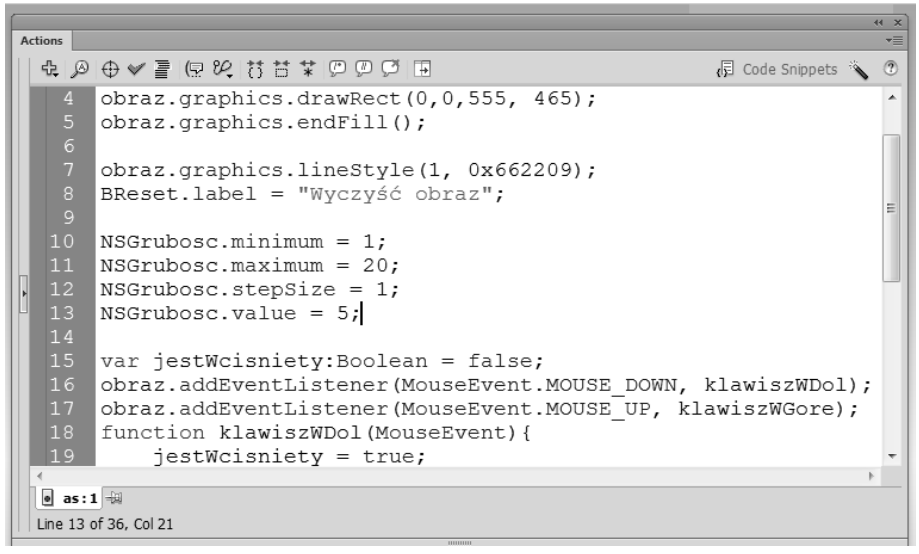
Rysunek 8.12. *Aby z powodzeniem wykorzystać nasze komponenty do określenia parametrów linii, konieczne jest odpowiednie skonfigurowanie ich właściwości*

```
BReset.label = "Wyczyść obraz";
```

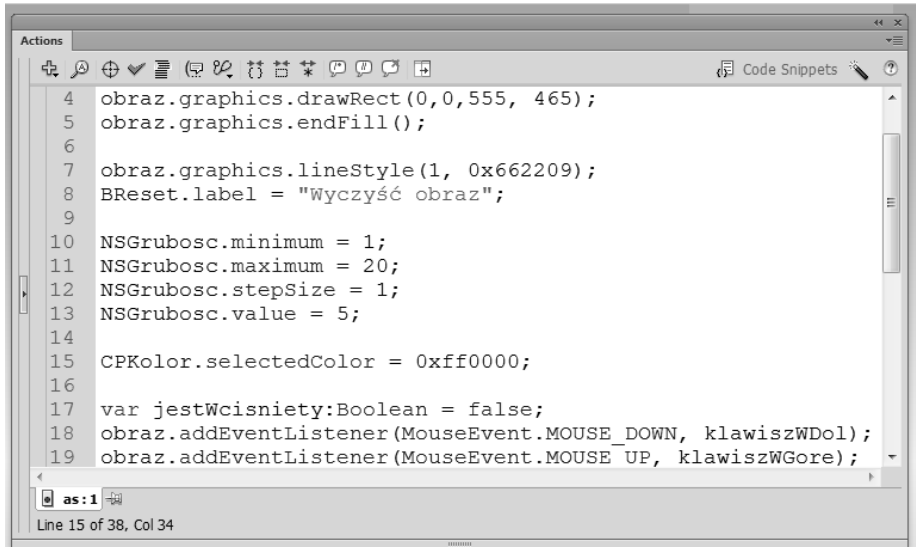
Więcej zachodu wymaga użycie komponentów `NumericStepper`. W tym przypadku powinniśmy określić ich wartość minimalną, maksymalną oraz domyślną, a także skok. W przypadku komponentu określającego grubość linii wykorzystujemy wartości wyrażone w pikselach. Przyjmijmy, że najcieńsza możliwa linia będzie miała grubość 1 piksela, zaś najgrubsza 20 pikseli. Domyślnie rozpoczniemy rysowanie z grubością 5 pikseli. Korzystając z nazwy instancji `NSGrubosc`, możemy szybko określić wymagane parametry (rysunek 8.13).

```
NSGrubosc.minimum = 1;
NSGrubosc.maximum = 2O;
NSGrubosc.stepSize = 1;
NSGrubosc.value = 5;
```

W przypadku komponentu `ColorPicker`, o ile wykorzystamy domyślnie kolor czarny, nie ma konieczności użycia żadnych dodatkowych parametrów. Jeśli jednak od początku chcielibyśmy malować innym kolorem, wystarczy wykorzystać właściwość `selectedColor`, aby określić domyślny kolor (rysunek 8.14).



Rysunek 8.13. *Przyjmijmy, że najcieńsza możliwa linia będzie miała grubość 1 piksela, zaś najgrubsza 20 pikseli. Domyślnie rozpoczniemy rysowanie z grubością 5 pikseli. Korzystając z nazwy instancji „NSGrubosc", możemy szybko określić wymagane parametry*

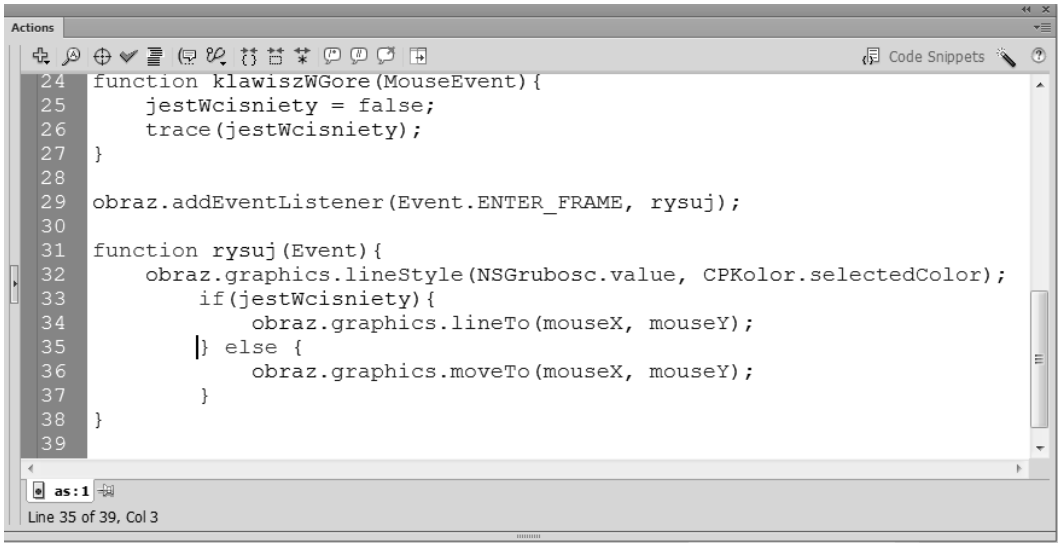


Rysunek 8.14. *W przypadku komponentu ColorPicker, o ile wykorzystamy domyślnie kolor czarny, nie ma konieczności użycia żadnych dodatkowych parametrów. Jeśli jednak od początku chcielibyśmy malować innym kolorem, wystarczy wykorzystać właściwość selectedColor, aby określić domyślny kolor*

```
CPKolor.selectedColor = 0xff0000;
```

W naszym przykładzie jako wartości domyślnej użyłem koloru czerwonego. Jeśli nie wprowadzimy więc żadnej zmiany, rozpoczynam rysowanie kolorem czerwonym o grubości linii 5 pikseli.

Ostatni krok to dodanie atrybutów linii wewnątrz przygotowanej wcześniej funkcji `rysuj()`. Wykorzystamy tu metodę `lineStyle()`, jednak w tym przypadku zamiast podawać konkretne wartości, dynamicznie odczytamy je z naszych komponentów. Wprowadzoną wcześniej linię w postaci `obraz.graphics.lineStyle(1, 0x662209);` możemy usunąć, a nową umieścić wewnątrz funkcji `rysuj()` (rysunek 8.15).

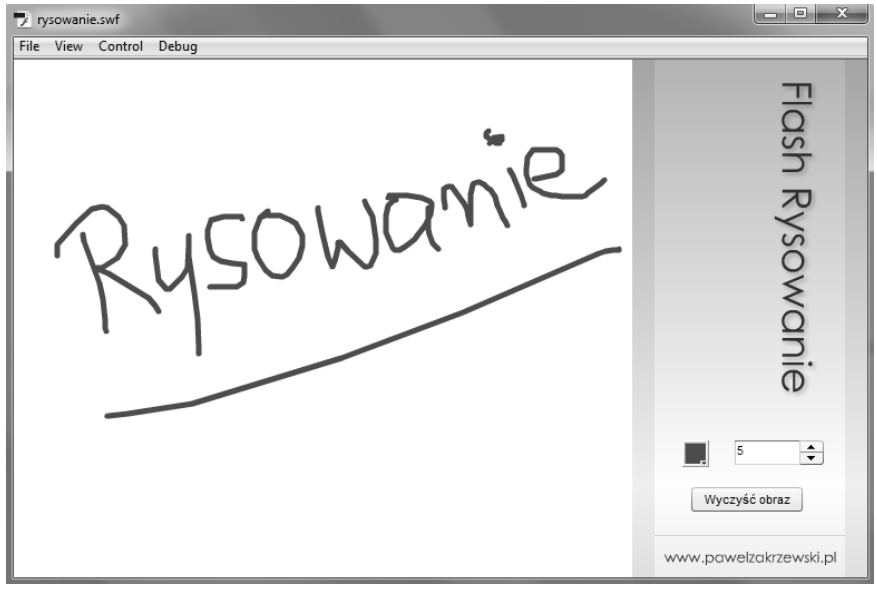


Rysunek 8.15. *Ostatni krok to dodanie atrybutów linii wewnątrz przygotowanej wcześniej funkcji rysuj(). Wykorzystamy tu metodę lineStyle(), jednak w tym przypadku zamiast podawać konkretne wartości, dynamicznie odczytamy je z naszych komponentów*

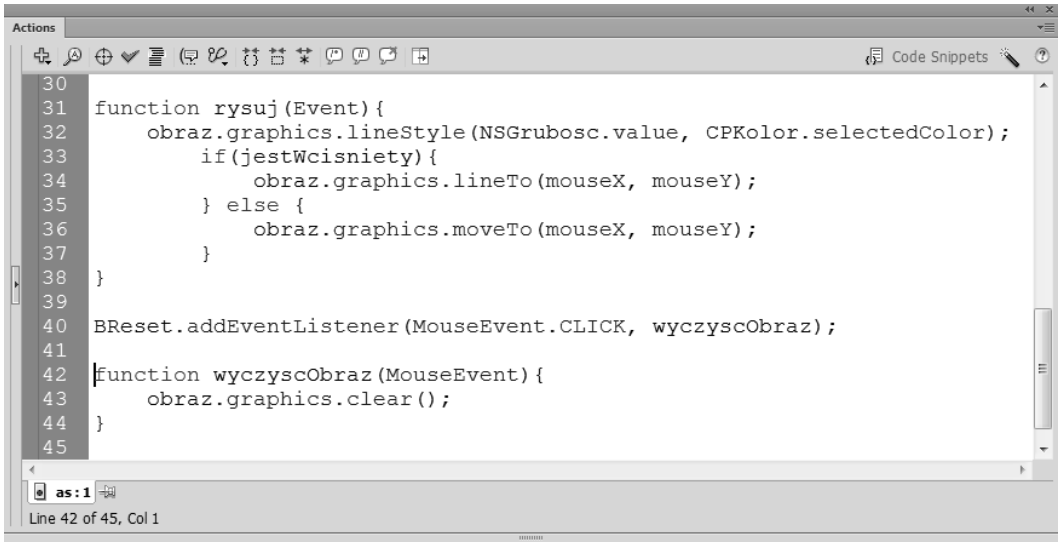
```
obraz.addEventListener(Event.ENTER_FRAME, rysuj);
function rysuj(Event){
  obraz.graphics.lineStyle(NSGrubosc.value, CPKolor.selectedColor);
  if(jestWcisniety){
    obraz.graphics.lineTo(mouseX, mouseY);
  } else {
    obraz.graphics.moveTo(mouseX, mouseY);
  }
}
```

Do określenia stylu linii wykorzystujemy w tym miejscu wartości wybrane za pomocą komponentów. Jeśli przed rozpoczęciem rysowania nie zmienimy żadnych ustawień, będziemy rysować linią o domyślnej konfiguracji (5 pikseli grubości i kolor czerwony) (rysunek 8.16).

Ostatni krok to dodanie możliwości czyszczenia całej pracy za pomocą przycisku *Wyczyść obraz*. Wykorzystamy tu metodę klasy `Graphics` o całkiem prostej nazwie — `clear()`. Całość wymaga jednak przygotowania nasłuchiwania, wciśnięcia przycisku oraz zdefiniowania odpowiedniej funkcji (rysunek 8.17).



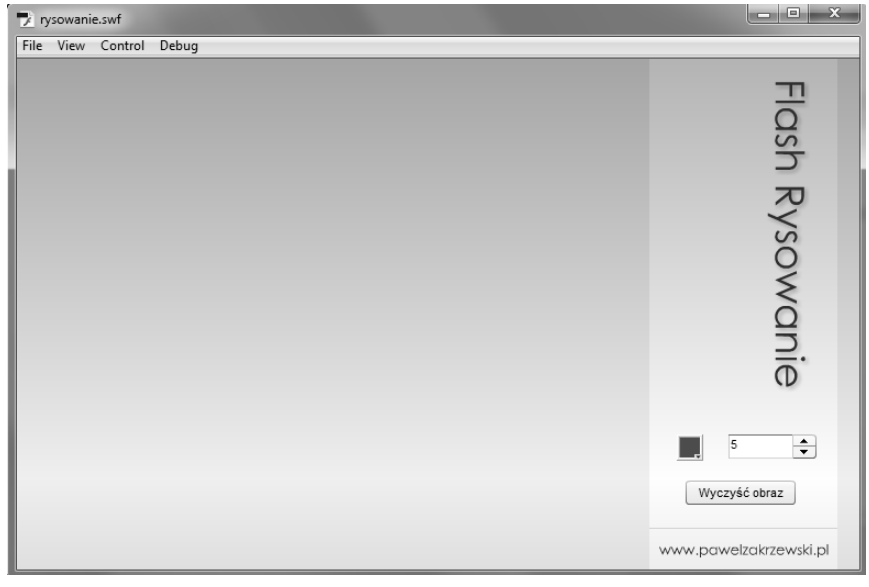
Rysunek 8.16. *Do określenia stylu linii wykorzystujemy w tym miejscu wartości wybrane za pomocą komponentów. Jeśli przed rozpoczęciem rysowania nie zmienimy żadnych ustawień, będziemy rysować linią o domyślnej konfiguracji (5 pikseli grubości i kolor czerwony)*



Rysunek 8.17. *Ostatni krok to dodanie możliwości czyszczenia całej pracy za pomocą przycisku Wyczyść obraz. Wykorzystamy tu metodę klasy Graphics o całkiem prostej nazwie — clear()*

```
BReset.addEventListener(MouseEvent.CLICK, wyczyscObraz);
function wyczyscObraz(MouseEvent){
  obraz.graphics.clear();
}
```

Niestety przy testowaniu tego zapisu narysowane elementy znikają wprawdzie ze sceny, znika jednak także cały obszar przeznaczony do rysowania. Usuwamy bowiem przy okazji całą zawartość wektorową naszego Sprite'a. Oznacza to, że po wyczyszczeniu całej pracy należy ponownie narysować obszar do rysowania (rysunek 8.18).



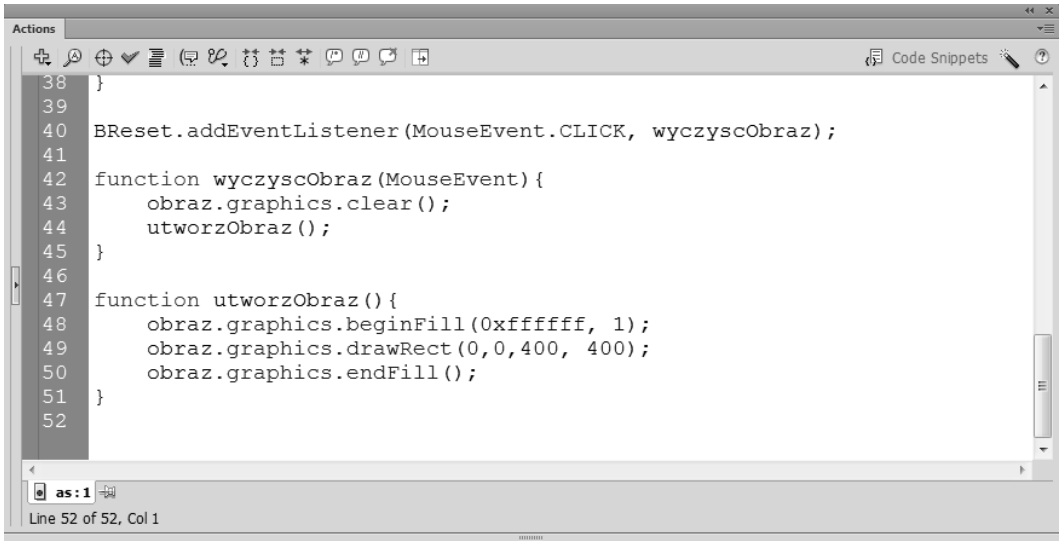
Rysunek 8.18. *Niestety przy testowaniu tego zapisu narysowane elementy znikają wprawdzie ze sceny, znika jednak także cały obszar przeznaczony do rysowania. Usuwamy bowiem przy okazji całą zawartość wektorową naszego Sprite'a*

Oczywiście można wykorzystać tu fragment, który wcześniej użyty został do utworzenia obrazu przeznaczonego do malowania, jednak taki kod nie będzie optymalny.

```
BReset.addEventListener(MouseEvent.CLICK, wyczyscObraz);
function wyczyscObraz(MouseEvent){
  obraz.graphics.clear();
  obraz.graphics.beginFill(0xffffff, 1);
  obraz.graphics.drawRect(0,0,400, 400);
  obraz.graphics.endFill();
}
```

Wszystkie metody odpowiedzialne za utworzenie Sprite'a możemy zamknąć w pojedynczej funkcji i wywołać ją zarówno w chwili uruchomienia dokumentu, jak i podczas czyszczenia naszej ilustracji (rysunek 8.19).

```
BReset.addEventListener(MouseEvent.CLICK, wyczyscObraz);
function wyczyscObraz(MouseEvent){
  obraz.graphics.clear();
  utworzObraz();
}
function utworzObraz(){
  obraz.graphics.beginFill(0xffffff, 1);
  obraz.graphics.drawRect(0,0,400, 400);
  obraz.graphics.endFill();
}
```



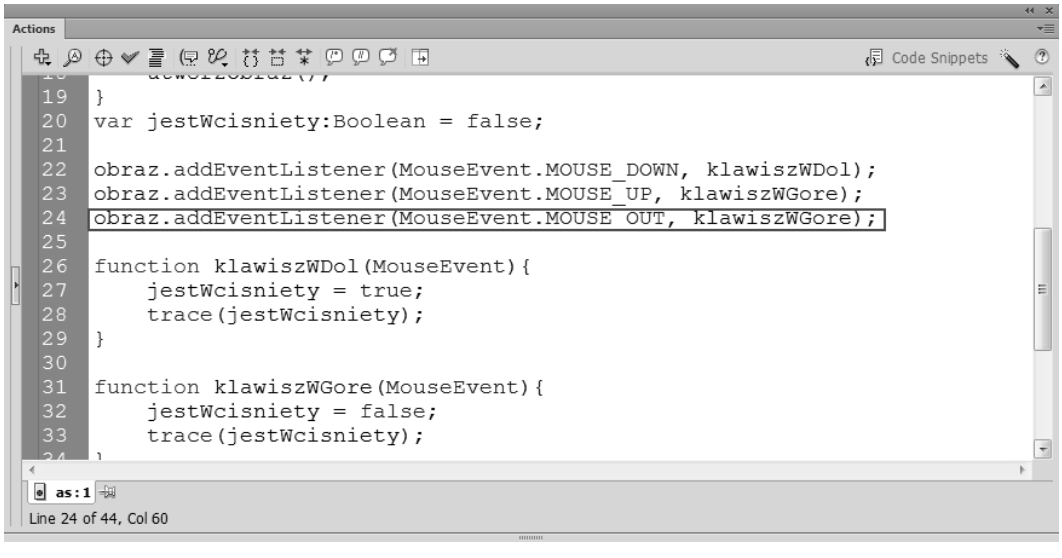
Rysunek 8.19. *Wszystkie metody odpowiedzialne za utworzenie Sprite'a możemy zamknąć w pojedynczej funkcji i wywołać ją zarówno w chwili uruchomienia dokumentu, jak i podczas czyszczenia naszej ilustracji*

Aby nie dublować treści zawartej wewnątrz dodanej właśnie funkcji `utworzObraz()`, możemy usunąć powtórzone elementy z wcześniejszej części kodu, a w ich miejsce dodać jedynie wywołanie tej właśnie funkcji. Całość mogłaby wyglądać w ten oto sposób:

```
BReset.label = "wyczyść obraz";
NSGrubosc.minimum = 1;
NSGrubosc.maximum = 20;
NSGrubosc.stepSize = 1;
NSGrubosc.value = 5;
CPKolor.selectedColor =  0xff0000;
var obraz:Sprite = new Sprite();
addChild(obraz);
function utworzObraz(){
  obraz.graphics.beginFill(0xffffff, 1);
  obraz.graphics.drawRect(0,0,400, 400);
  obraz.graphics.endFill();
}
utworzObraz();
BReset.addEventListener(MouseEvent.CLICK, wyczyscObraz);
function wyczyscObraz(MouseEvent){
  obraz.graphics.clear();
  utworzObraz();
}
var jestWcisniety:Boolean = false;
obraz.addEventListener(MouseEvent.MOUSE_DOWN, klawiszWDol);
obraz.addEventListener(MouseEvent.MOUSE_UP, klawiszWGore);
obraz.addEventListener(MouseEvent.MOUSE_OUT, klawiszWGore);
function klawiszWDol(MouseEvent){
  jestWcisniety = true;
  trace(jestWcisniety);
}
```

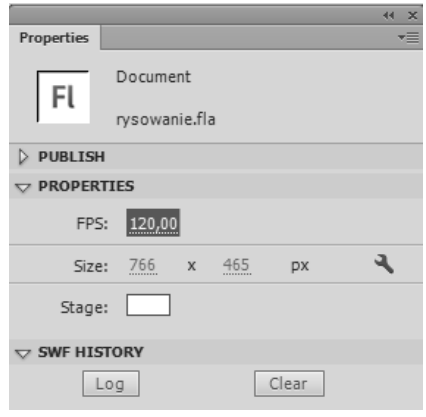
```
function klawiszWGore(MouseEvent){
  jestWcisniety = false;
  trace(jestWcisniety);
}
obraz.addEventListener(Event.ENTER_FRAME, rysuj);
function rysuj(Event){
  obraz.graphics.lineStyle(NSGrubosc.value, CPKolor.selectedColor);
  if(jestWcisniety){
    obraz.graphics.lineTo(mouseX, mouseY);
  } else {
    obraz.graphics.moveTo(mouseX, mouseY);
  }
}
```

W przedstawionym powyżej kodzie pojawiła się jeszcze jedna dodatkowa linia. Dodałem bowiem obsługę zdarzenia `MOUSE_OUT` (rysunek 8.20). W ten sposób w chwili, gdy podczas rysowania wyjedziemy myszką poza obszar przygotowanego obszaru do malowania, wywołana zostanie funkcja `klawiszWGore()`, a to powoduje przerwanie rysowania. W rezultacie program pozwala na malowanie wyłącznie na przygotowanym w tym celu obszarze.

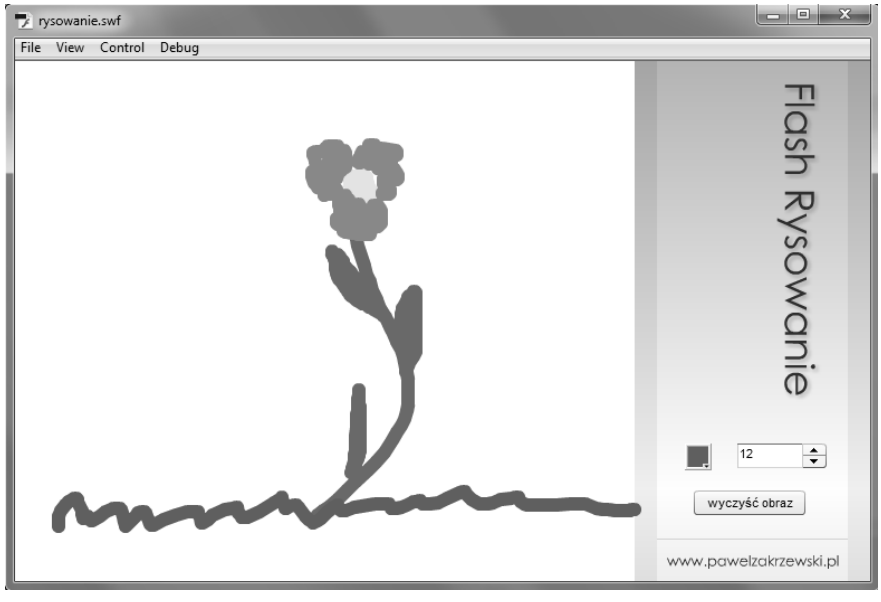


Rysunek 8.20. *W przedstawionym powyżej kodzie pojawiła się jeszcze jedna dodatkowa linia. Dodałem bowiem obsługę zdarzenia MOUSE_OUT*

Aby utworzone podczas rysowania linie były nieco bardziej wygładzone i jednocześnie mniej kanciaste, możemy pokusić się o zwiększenie parametru *Frame Rate*, czyli prędkości odtwarzania naszej pracy. Nadając mu wartość maksymalną na poziomie 120 fps, wymuszamy czyściejsze rysowanie prostych odcinków (rysunek 8.21). W ten sposób rysowane linie stają się bardziej wygładzone i są po prostu ładniejsze (rysunek 8.22).



Rysunek 8.21. *Aby utworzone podczas rysowania linie były nieco bardziej wygładzone i jednocześnie mniej kanciaste, możemy pokusić się o zwiększenie parametru Frame Rate, czyli prędkości odtwarzania naszej pracy*



Rysunek 8.22. *Nadając mu wartość maksymalną na poziomie 120 fps, wymuszamy czyściejsze rysowanie prostych odcinków. W ten sposób rysowane linie stają się bardziej wygładzone i są po prostu ładniejsze*

A może zdrapka?

Innym całkiem prostym przykładem użycia w praktyce kodu ActionScript może być popularna zdrapka. Jej działanie do złudzenia przypomina oryginał znany z różnorodnych loterii i losów z życia codziennego.

Zdrapka z użyciem wypełnienia bitmapowego

Co niezwykle ciekawe, przygotowanie zdrapki z użyciem wypełnienia bitmapowego do złudzenia przypomina omówiony właśnie przykład rysowanki. Różnica tkwi przede wszystkim w ustawieniach „pędzla", który wykorzystany zostanie do zdrapywania/malowania obrazu. W tym przypadku będzie on tworzył wypełnienie bitmapowe na bazie grafiki bitmapowej.

Aby cały przykład nieco uatrakcyjnić i jednocześnie poznać kilka ciekawych technik pracy, do zdrapki wykorzystamy plik graficzny dynamicznie doładowany do naszej pracy.

Rozpoczynamy jednak od przygotowania nowego dokumentu oraz odpowiedniej grafiki — zdjęcia. Zakładamy, że wielkość zdjęcia dopasujemy do naszej sceny. Korzystając zatem z dowolnego programu graficznego, przygotowujemy odpowiedni plik graficzny, ustalamy jego wymiary i w tej postaci zapisujemy na dysku. Budujemy nowy dokument o wielkości sceny zgodnej z rozmiarem przygotowanego zdjęcia. Aby możliwe było dynamiczne ładowanie obrazu do naszej pracy, zarówno zdjęcia, jak i bieżący dokument zapisujemy w tej samej lokalizacji na dysku komputera.

Ładujemy zewnętrzną fotografię za pomocą klasy Loader

Ładowanie zewnętrznych elementów graficznych takich jak pliki PNG, JPEG, GIF, a także SWF wymaga użycia klasy `Loader` lub omówionych wcześniej komponentów. O ile praca z komponentami nie sprawia w zasadzie żadnych problemów, o tyle wykorzystanie klasy `Loader` i samodzielne tworzenie kodu ActionScript wymaga nieco więcej zaangażowania. Oferuje jednak wiele dodatkowych korzyści.

Obiekt ładowany za pomocą klasy `Loader` możemy dowolnie wykorzystać do wszelkich działań, a dodatkowo tak przygotowany projekt jest zwykle lżejszy o 20 KB. Wadą jest tu jednak konieczność samodzielnego opracowania stosownego kodu, który nie tylko załaduje, ale także obsłuży konkretny plik graficzny. Nie narzekajmy jednak zawczasu i spróbujmy przejść do konkretów.

Ładowanie zdjęcia z pliku zewnętrznego

Nową instancję klasy `Loader` o nazwie `loader` tworzymy w klasyczny sposób:

```
var loader:Loader = new Loader();
```

Aby możliwe było załadowanie zewnętrznego pliku, konieczne jest określenie jego adresu URL. Jak pamiętamy, w języku ActionScript istnieje specjalna klasa odpowiedzialna za dostarczanie adresów URL — nazywa się `URLRequest`. Podobnie jak wiele innych klas, `URLRequest` do użycia w naszej pracy wymaga utworzenia instancji. Korzystając więc z pojedynczej linii kodu, budujemy nową instancję i nadajemy jej nazwę `adres`.

```
var adres:URLRequest = new URLRequest();
```

Oczywiście samo przygotowanie instancji nie wystarcza do tego, aby poprawnie wskazać adres `url` pliku do załadowania. Dzięki właściwości `url` klasy `URLRequest` do utworzonej właśnie instancji możemy łatwo przypisać konkretny adres `url`.

```
adres.url = "foto.jpg";
```

Takie działanie pozwoli nam na rozpoczęcie procesu ładowania zdjęcia. Warto zwrócić uwagę na fakt, że wywołanie metody `load()` oznacza wyłącznie rozpoczęcie ładowania wskazanego pliku graficznego (podobnie wygląda to w przypadku ładowania tekstów czy dokumentów SWF), a nie jego załadowanie. Zanim konkretny plik się załaduje, musi upłynąć nieco czasu. Ten zaś zależny jest od wielu często niezależnych od nas czynników takich jak prędkość łącza, chwilowe obciążenie łącza czy wielkość/waga pliku do załadowania.

```
loader.load(adres);
```

Aby można było w jakikolwiek sposób obsłużyć wybrany plik graficzny, musi się on w całości załadować. Dopiero po pełnym załadowaniu możemy wyświetlić go na scenie lub poddać innym działaniom. Klasycznym sposobem zdobycia informacji o zakończeniu ładowania jest użycie nasłuchiwania zdarzenia `COMPLETE`. Zastosujemy je także i w naszym przykładzie.

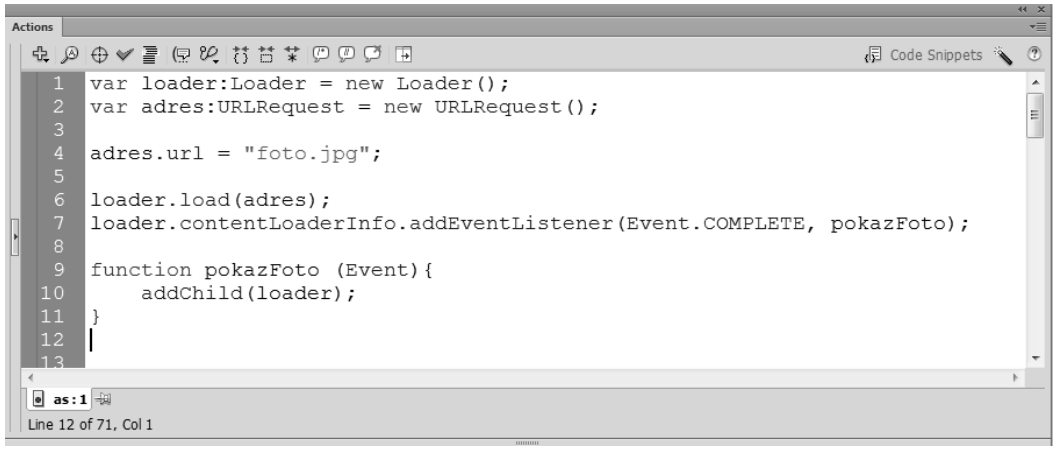
Na pierwszy rzut oka dodanie możliwości nasłuchiwania zdarzenia `COMPLETE` do instancji klasy `Loader` wydaje się całkiem proste, niestety takie nie jest. Obiekt typu `loader` nie reaguje na zdarzenie `COMPLETE`. Dlaczego?

W tym przypadku odpowiedź jest bardzo prosta i, co ważne, logiczna. Po prostu jest on już w pamięci i nie wymaga reakcji na ładowanie. Jest załadowany i gotowy do użycia. Co zatem ładujemy w rezultacie użycia metody `load()`?

W tym przypadku ładujemy **zawartość** *Loadera*. On sam jest już w pamięci, a po wywołaniu metody `load()` rozpoczynamy proces ładowania nowej zawartości. W ten sposób obiektem, który nasłuchuje całkowitego załadowania zdjęcia, powinna być zawartość *Loadera*. I tak jest w istocie. Korzystając z właściwości `content ↪LoaderInfo`, możemy z łatwością nasłuchiwać zdarzenia związanego z ładowaniem zawartości *Loadera*.

```
loader.contentLoaderInfo.addEventListener(Event.COMPLETE, pokazFoto);
```

Naturalnie, aby przedstawiony kod miał szansę działać, konieczne jest przygotowanie funkcji `pokazFoto()`. Jej działanie będzie niezwykle proste — ma jedynie wyświetlić załadowany obraz na scenie. Korzystamy tu z metody `addChild()`, podając jako parametr nazwę instancji naszego *Loadera* (rysunek 8.23).



Rysunek 8.23. *Naturalnie, aby przedstawiony kod miał szansę działać, konieczne jest przygotowanie funkcji pokazFoto(). Jej działanie będzie niezwykle proste — ma jedynie wyświetlić załadowany obraz na scenie*

```
var loader:Loader = new Loader();
var adres:URLRequest = new URLRequest();

adres.url = "foto.jpg";

loader.load(adres);
loader.contentLoaderInfo.addEventListener(Event.COMPLETE, pokazFoto);

function pokazFoto (Event){
  addChild(loader);
}
```

Przedstawione działania pozwalają nam ładować i wyświetlać na scenie różnorodne pliki graficzne, a także projekty w formacie SWF. W tym przykładzie nie potrzeba nam nic więcej (rysunek 8.24).

Aby możliwe było wykorzystanie załadowanej właśnie fotografii do budowy naszej zdrapki, konieczne jest użycie klasy `BitmapData`. Pozwoli to na utworzenie nowego obiektu — grafiki bitmapowej, którą wykorzystamy jako źródło do malowania na scenie. Co ważne, budując nową instancję klasy `BitmapData`, wykorzystamy w tym przypadku załadowaną grafikę. To właśnie ona stanie się bitmapą, którą wykorzystamy nieco później do malowania.

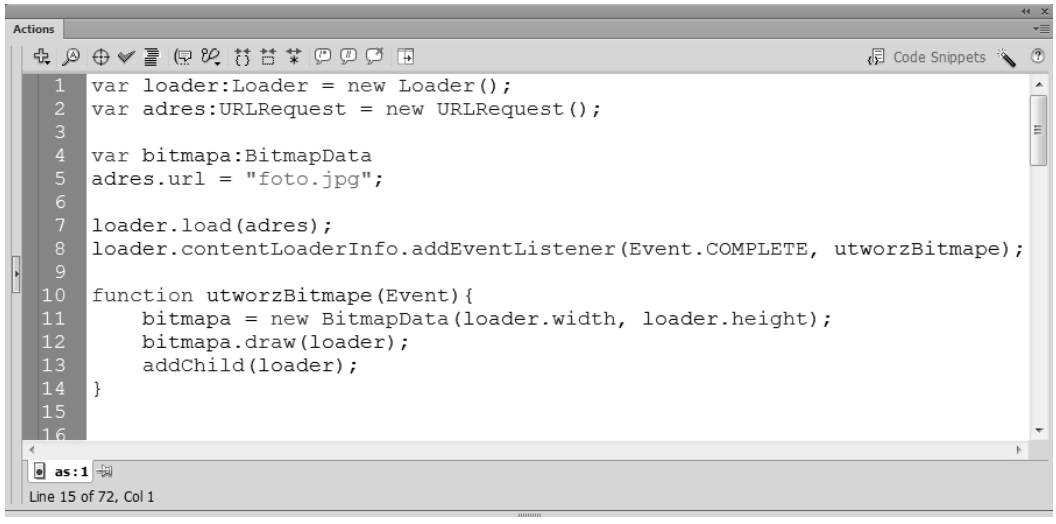
Rysunek 8.24. *Przedstawione działania pozwalają nam ładować i wyświetlać na scenie różnorodne pliki graficzne, a także projekty w formacie SWF. W tym przykładzie nie potrzeba nam nic więcej*

Budujemy nową bitmapę

Utworzenie nowej instancji klasy `BitmapData` wymaga określenia rozmiarów nowej grafiki. W naszym przypadku możemy zastosować wartości relatywne, wykorzystując zarówno wymiary sceny, jak i rozmiar załadowanej fotografii. Chyba najgorszym rozwiązaniem byłoby wprowadzenie sztywnych wartości liczbowych określających wymiar nowego obiektu. Korzystając z właściwości `width` oraz `height`, możemy łatwo odczytać rozmiar załadowanego zdjęcia i wykorzystać te informacje do budowy nowej bitmapy.

Wszystko wydaje się proste, jednak niesie za sobą możliwość popełnienia błędu. W chwili uruchomienia naszej pracy obiekt typu `Loader` rozpoczyna ładowanie wskazanej fotografii. W tym momencie jego rozmiar wynosi 0 x 0 pikseli, a takich wartości nie można wykorzystać do tworzenia nowej bitmapy. Aby precyzyjnie odczytać jego rozmiary, musimy poczekać na załadowanie całego obrazu. Zmodyfikujemy więc nieco nasz kod i poza wyświetleniem zdjęcia na stronie utworzymy z niego nową grafikę bitmapową (rysunek 8.25).

```
var loader:Loader = new Loader();
var adres:URLRequest = new URLRequest();
```



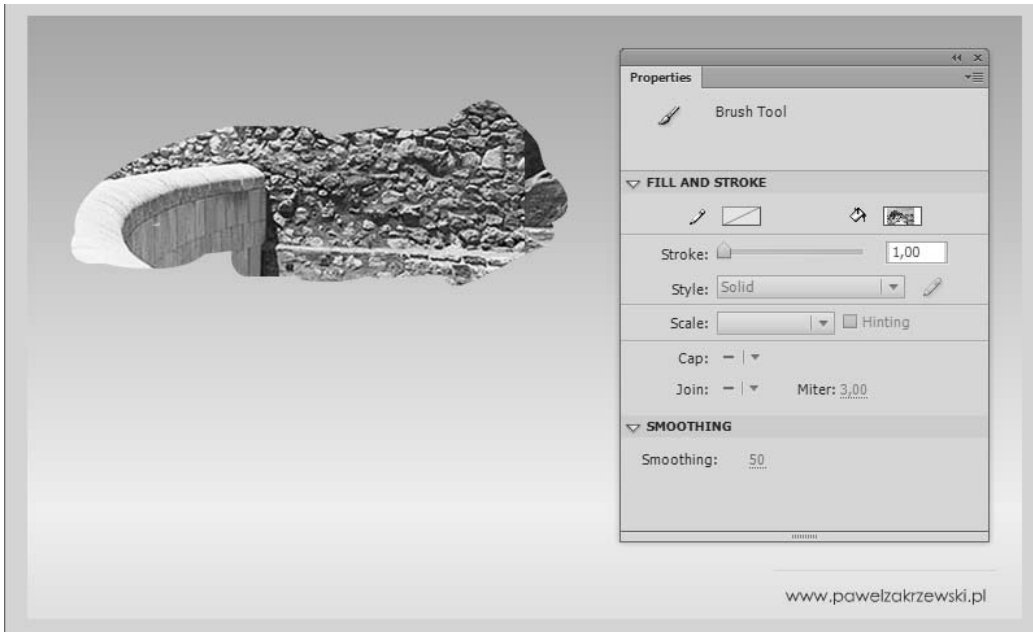
Rysunek 8.25. *Zmodyfikujemy więc nieco nasz kod i poza wyświetleniem zdjęcia na stronie utworzymy z niego nową grafikę bitmapową*

```
var bitmapa:BitmapData
adres.url = "foto.jpg";

loader.load(adres);
loader.contentLoaderInfo.addEventListener(Event.COMPLETE, utworzBitmape);

function utworzBitmape(Event){
  bitmapa = new BitmapData(loader.width, loader.height);
  bitmapa.draw(loader);
  addChild(loader);
}
```

W przedstawionym kodzie warto zwrócić uwagę na metodę `draw()`, która umożliwia wyświetlanie grafiki bitmapowej za pomocą wektorowych algorytmów programu Flash. W ten sposób utworzyliśmy własną grafikę bitmapową, którą nieco później wykorzystamy do malowania na scenie.

Dodajemy malowanie bitmapą

Adobe Flash od (chyba) zawsze umożliwiał nam malowanie wektorowe za pomocą wypełnienia bitmapowego. To ciekawa funkcja, która posłuży nam za schemat przygotowania pierwszej wersji zdrapki. Wykorzystując grafikę bitmapową jako „kolor/źródło" pędzla, możemy z łatwością malować bitmapą (rysunek 8.26). Działanie to do złudzenia przypomina działanie klasycznej zdrapki, a to jest przecież naszym celem. Nie zapeszajmy więc — niewiele pracy nam już pozostało!



Rysunek 8.26. *Adobe Flash od (chyba) zawsze umożliwiał nam malowanie wektorowe za pomocą wypełnienia bitmapowego. To ciekawa funkcja, która posłuży nam za schemat przygotowania pierwszej wersji zdrapki. Wykorzystując grafikę bitmapową jako „kolor/źródło" pędzla, możemy z łatwością malować bitmapą*

Podobnie jak w wielu innych przypadkach związanych z tworzeniem na scenie obiektów wektorowych, pracę rozpoczynamy od przygotowania kontenera, który umożliwi nam malowanie. Wykorzystamy tu obiekt typu `Sprite` z przypisaną nazwą instancji — `obraz`.

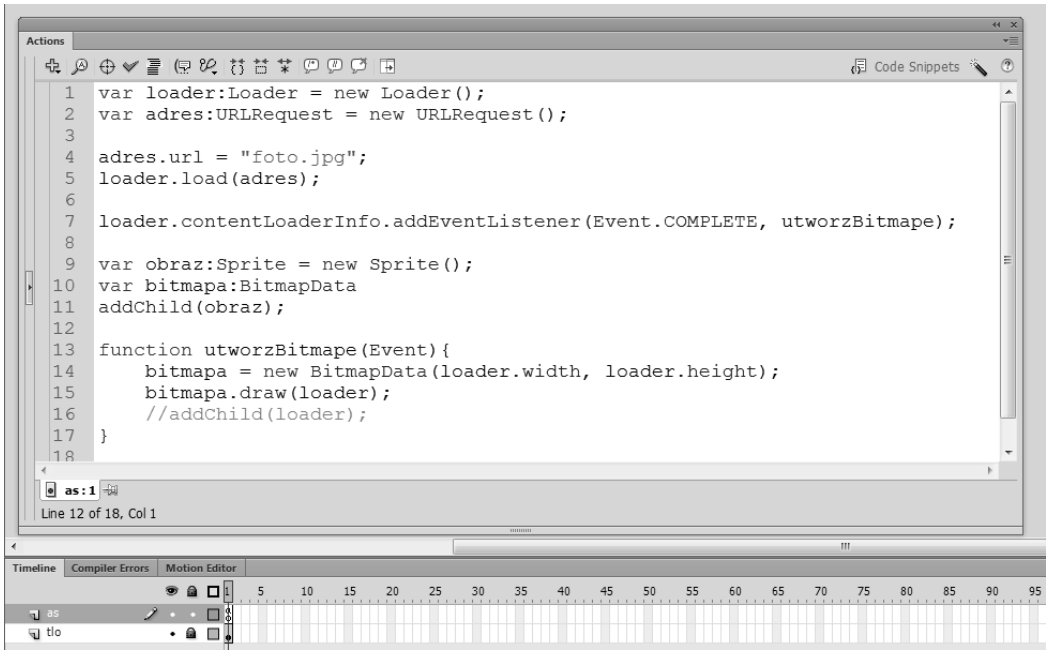
```
var obraz:Sprite = new Sprite();
```

Aby możliwe było użycie do malowania obiektu `obraz`, koniecznie dodajemy go do listy obiektów wyświetlanych na scenie.

```
addChild(obraz);
```

Obie instrukcje możemy wprowadzić niemal w dowolnym miejscu naszego kodu. Najbardziej elegancko będzie tego typu deklaracje umieścić nieopodal podobnych instrukcji (rysunek 8.27).

```
var loader:Loader = new Loader();
var adres:URLRequest = new URLRequest();
adres.url = "foto.jpg";
loader.load(adres);
loader.contentLoaderInfo.addEventListener(Event.COMPLETE, utworzBitmape);
var obraz:Sprite = new Sprite();
var bitmapa:BitmapData
addChild(obraz);
```



Rysunek 8.27. *Obie instrukcje możemy wprowadzić niemal w dowolnym miejscu naszego kodu. Najbardziej elegancko będzie tego typu deklaracje umieścić nieopodal podobnych instrukcji*

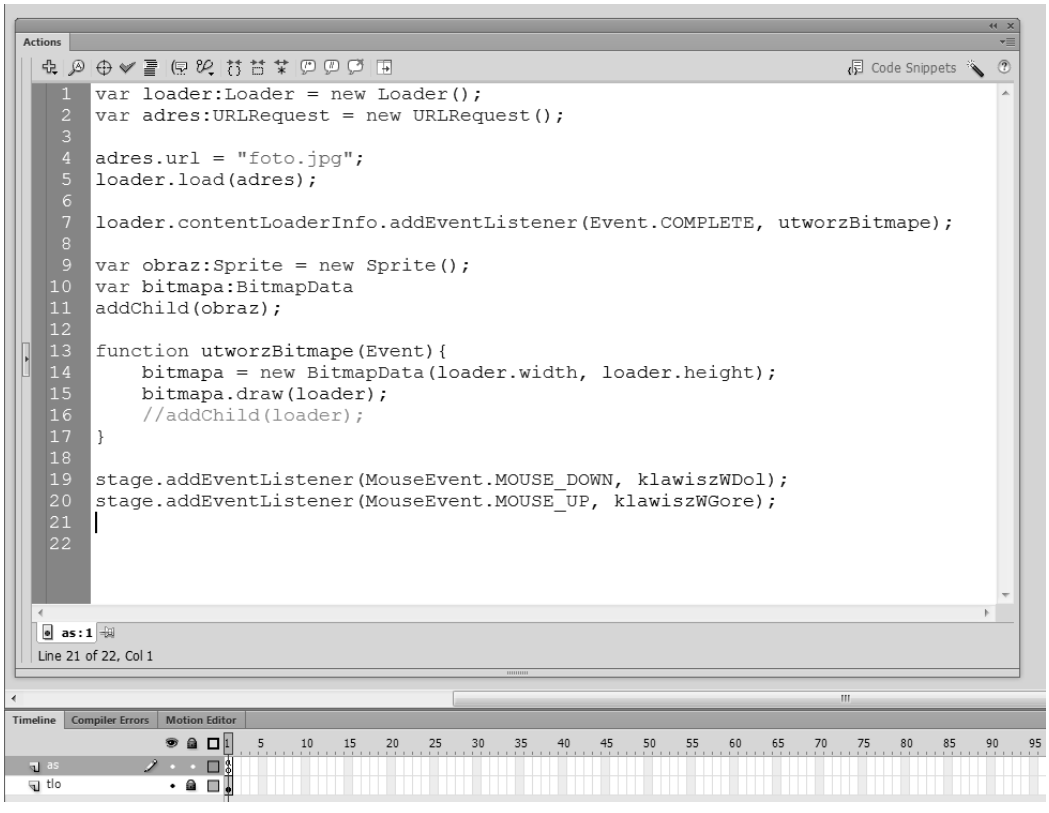
```
function utworzBitmape(Event){
  bitmapa = new BitmapData(loader.width, loader.height);
  bitmapa.draw(loader);
  //addChild(loader);
}
```

W ten sposób mamy już przygotowaną całą strukturę graficzną. Załadowana grafika bitmapowa jest gotowa, aby wykorzystać ją do malowania, na scenie zaś mamy dodany Sprite o nazwie obraz, który umożliwia nam właściwe malowanie.

Sam proces malowania do złudzenia przypomina przygotowaną w poprzednim przykładzie rysowankę. W pierwszej kolejności wprowadzamy nasłuchiwanie zdarzeń `MOUSE_DOWN` oraz `MOUSE_UP`, które decydują o rozpoczęciu i zakończeniu malowania — zdrapywania. Całość zrealizujemy jednak nieco inaczej niż do tej pory.

Kluczowe okaże się tu nasłuchiwanie zdarzeń. W chwili wciśnięcia myszki dodamy możliwość nasłuchiwania zdarzenia `ENTER_FRAME`, które odpowiadać będzie za proces rysowania. W momencie zwolnienia przycisku myszki usuniemy nasłuchiwanie `ENTER_FRAME` i tym samym zakończymy rysowanie. Tak przygotowany przykład (mimo użycia niezbyt wydajnego zdarzenia `ENTER_FRAME`) działa zupełnie dobrze. Reakcja na `ENTER_FRAME` wywoływana jest jedynie w chwili, gdy jest konieczna, czyli w trakcie malowania. W pozostałych sytuacjach zdarzenie to nie jest obsługiwane.

Rozpoczynamy od nasłuchiwania zdarzeń MOUSE_DOWN oraz MOUSE_UP — to one mają podstawowy wpływ na działanie naszej zdrapki. Za nasłuchiwanie tych zdarzeń odpowiedzialna będzie w tym przypadku scena (rysunek 8.28).

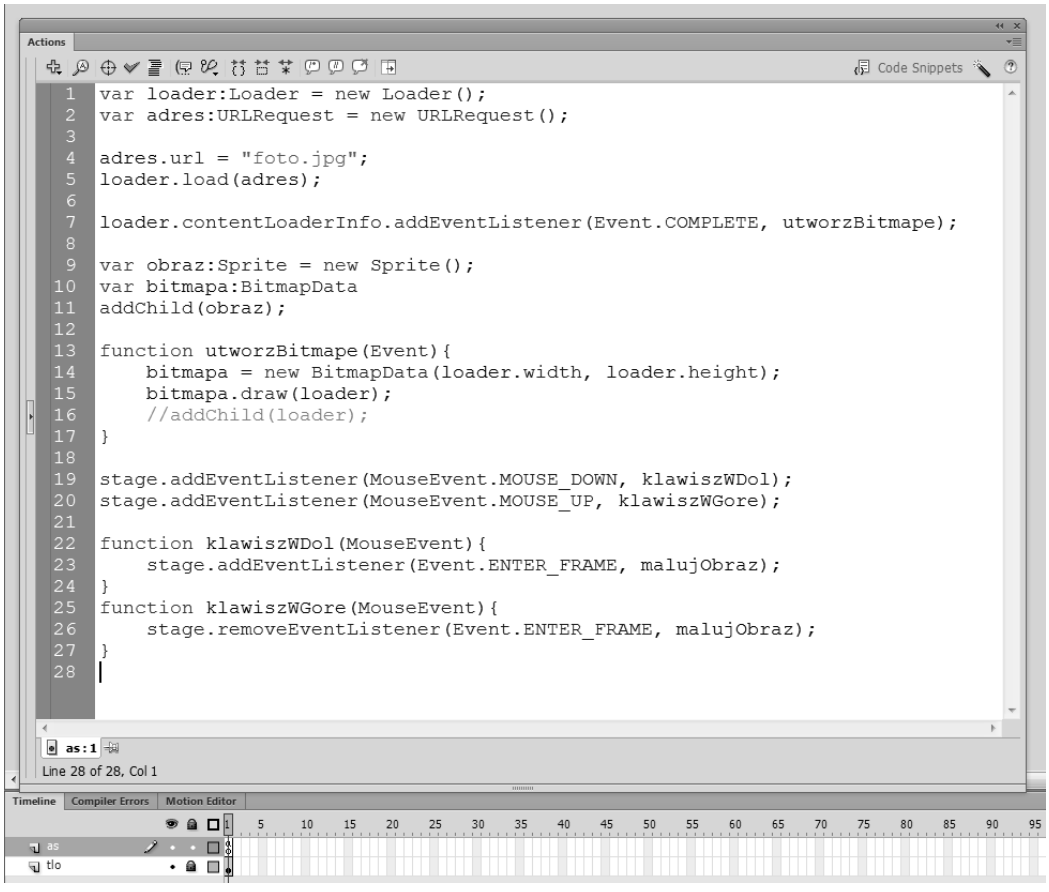


Rysunek 8.28. *Rozpoczynamy od nasłuchiwania zdarzeń MOUSE_DOWN oraz MOUSE_UP — to one mają podstawowy wpływ na działanie naszej zdrapki. Za nasłuchiwanie tych zdarzeń odpowiedzialna będzie w tym przypadku scena*

```
stage.addEventListener(MouseEvent.MOUSE_DOWN, klawiszWDol);
stage.addEventListener(MouseEvent.MOUSE_UP, klawiszWGore);
```

Do obsługi obu zdarzeń musimy przygotować dwie dodatkowe funkcje. To właśnie one odpowiedzialne będą za dodanie lub usunięcie nasłuchiwania zdarzenia ENTER_FRAME realizującego właściwe malowanie (rysunek 8.29).

```
function klawiszWDol(MouseEvent){
  stage.addEventListener(Event.ENTER_FRAME, malujObraz);
}
function klawiszWGore(MouseEvent){
  stage.removeEventListener(Event.ENTER_FRAME, malujObraz);
}
```

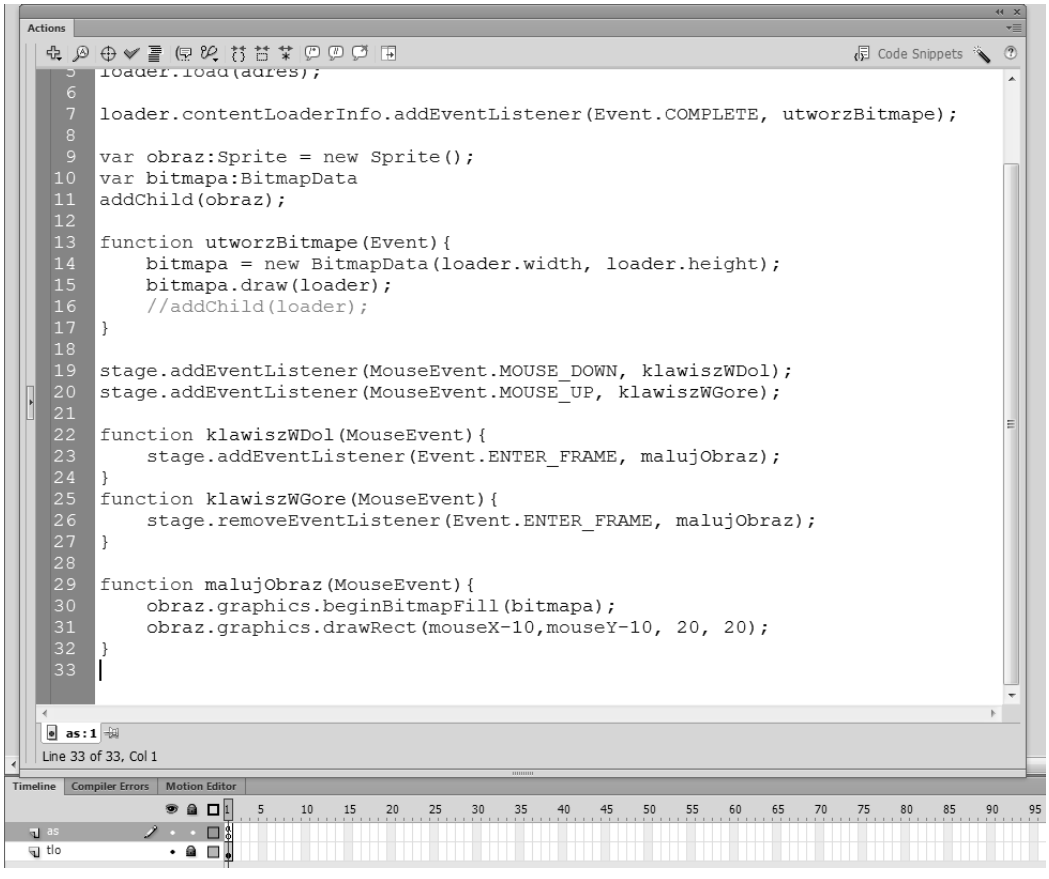


Rysunek 8.29. *Do obsługi obu zdarzeń musimy przygotować dwie dodatkowe funkcje. To właśnie one odpowiedzialne będą za dodanie lub usunięcie nasłuchiwania zdarzenia ENTER_FRAME realizującego właściwe malowanie*

Aby całość mogła zadziałać, musimy jeszcze określić działanie funkcji `malujObraz()`. W tym miejscu określamy parametry pędzla — `beginBitmapFill()` oraz rozpoczynamy rysowanie grafik o dowolnym kształcie wektorowym — `drawRect()` (rysunek 8.30).

```
function malujObraz(MouseEvent){
  obraz.graphics.beginBitmapFill(bitmapa);
  obraz.graphics.drawRect(mouseX-10,mouseY-10, 20, 20);
}
```

Aby uzyskać nieco toporny kształt przypominający klasyczną zdrapkę do malowania, użyłem obiektu o kształcie prostokąta. Można pokusić się jednak o przygotowanie jednocześnie kilku podobnych kształtów, co nada naszej pracy więcej realizmu (rysunek 8.31).



Rysunek 8.30. *Aby całość mogła zadziałać, musimy jeszcze określić działanie funkcji malujObraz(). W tym miejscu określamy parametry pędzla — beginBitmapFill() oraz rozpoczynamy rysowanie grafik o dowolnym kształcie wektorowym — drawRect()*

Rysunek 8.31. *Aby uzyskać nieco toporny kształt przypominający klasyczną zdrapkę do malowania, użyłem obiektu o kształcie prostokąta. Można pokusić się jednak o przygotowanie jednocześnie kilku podobnych kształtów, co nada naszej pracy więcej realizmu*

Kod całej zabawki przedstawia się następująco:

```
var loader:Loader = new Loader();
var adres:URLRequest = new URLRequest();
adres.url = "foto.jpg";
loader.load(adres);
loader.contentLoaderInfo.addEventListener(Event.COMPLETE,
➥utworzBitmape);
var obraz:Sprite = new Sprite();
var bitmapa:BitmapData
addChild(obraz);
function utworzBitmape(Event){
  bitmapa = new BitmapData(loader.width, loader.height);
  bitmapa.draw(loader);
}
stage.addEventListener(MouseEvent.MOUSE_DOWN, klawiszWDol);
stage.addEventListener(MouseEvent.MOUSE_UP, klawiszWGore);
function klawiszWDol(MouseEvent){
  stage.addEventListener(Event.ENTER_FRAME, malujObraz);
}
function klawiszWGore(MouseEvent){
  stage.removeEventListener(Event.ENTER_FRAME, malujObraz);
}
function malujObraz(MouseEvent){
  obraz.graphics.beginBitmapFill(bitmapa);
  obraz.graphics.drawRect(mouseX-10,mouseY-10, 20, 20);
  obraz.graphics.drawRect(mouseX-14,mouseY-14, 3, 5);
  obraz.graphics.drawRect(mouseX+14,mouseY-14, 8, 12);
  obraz.graphics.drawRect(mouseX+18,mouseY+11, 12, 5);
}
```

Dziecięca kolorowanka

Jednym z klasycznych przykładów zabawek dla dzieci wykonanych za pomocą programu Flash może być prosta kolorowanka. Mimo że nie wymaga wprowadzania wielu linii kodu, jest jednak dość wymagająca w sferze przygotowania odpowiedniej grafiki. Wymaga bowiem przygotowania serii obiektów typu MovieClip utworzonych na bazie fragmentów obrazu.

Przygotowanie grafiki do kolorowania

Do budowy kolorowanki najlepiej jest wykorzystać prosty, czarno-biały rysunek wektorowy z niewielką liczbą drobnych szczegółów. Możemy zastosować w tym miejscu zarówno grafikę utworzoną w dowolnym programie wektorowym typu Adobe Illustrator czy Corel Draw, jak i plik bitmapowy przekształcony w wektor w programie Flash.

W naszym przypadku skorzystałem z tej drugiej możliwości. Importujemy więc grafikę bitmapową i za pomocą polecenia *Break Apart* (*Ctrl+B*) przekształcamy ją w wypełnienie bitmapowe (rysunek 8.32). Taka konstrukcja przyda się do tworzenia symboli.

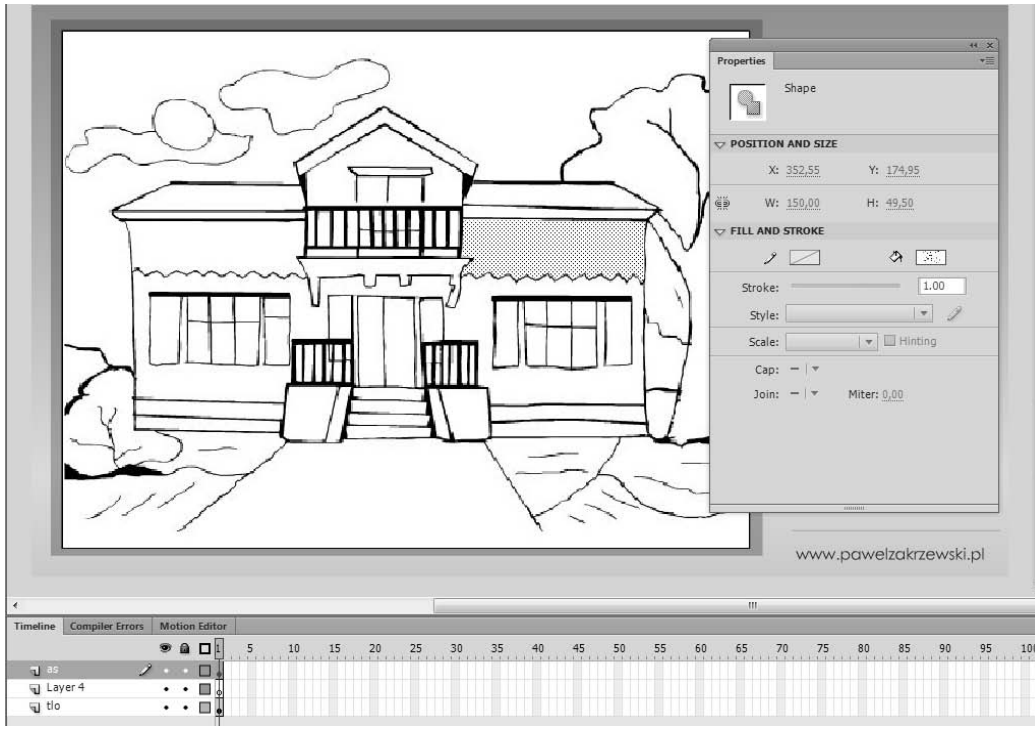
Rysunek 8.32. *Importujemy grafikę bitmapową i za pomocą polecenia* Break Apart *(*Ctrl+B*) przekształcamy ją w wypełnienie bitmapowe*

Wybieramy narzędzie *Lasso* (*L*) i w dolnej części palety zmieniamy tryb jego działania, przełączając się na *magiczną różdżkę* (rysunek 8.33). Ułatwi nam ona zaznaczanie obszarów obrazu zamkniętych pomiędzy czarnymi krawędziami. Aby sprawdzić, czy wszystko działa poprawnie, klikamy dowolny fragment obrazu. W rezultacie powinno pojawić się zaznaczenie o nieregularnym kształcie, które powstało na bazie wybrania jednolitego koloru — bieli pomiędzy czarnymi liniami (rysunek 8.34). Mechanizm ten wykorzystamy do tworzenia docelowych symboli.

Aby odznaczyć zaznaczenie, klikamy dowolny punkt na szarym obszarze wokół sceny i przystępujemy do konkretnej pracy. Będziemy zaznaczali fragmenty obrazu i te, które powinny być kolorowane jednocześnie, po zaznaczeniu przekształcamy w symbol typu `MovieClip`. Zasada działania jest więc następująca.

Rysunek 8.33. *Wybieramy narzędzie Lasso (L) i w dolnej części palety zmieniamy tryb jego działania, przełączając się na magiczną różdżkę*

Korzystając z przedstawionej w naszym przykładzie grafiki, klikamy fragment ściany domu i obserwujemy powstałe zaznaczenie. W moim przykładzie aktywny stał się fragment ściany widoczny po prawej stronie. Prawdopodobnie większość z nas w ten sam sposób pokoloruje także drugi fragment ściany widoczny po lewej stronie. Aby dodać ten kształt do zaznaczenia, należy kliknąć. W rezultacie aktywne są obie ściany budynku, które przekształcimy w pojedynczy symbol. W tym celu, korzystając z polecenia *F8*, wywołujemy polecenie *Convert to symbol* i na bazie przygotowanego zaznaczenia budujemy symbol typu `MovieClip` (rysunek 8.35). Jego nazwa w bibliotece nie ma tu żadnego znaczenia, więc szkoda czasu, aby ją w tym miejscu wprowadzać. Po utworzeniu symbolu klikamy dowolny punkt na szarym tle wokół sceny, aby dezaktywować zaznaczenie, a następnie w podobny sposób zaznaczamy kolejny element lub elementy obrazu. Każdy fragment, który chcemy malować oddzielnie, przekształcamy kolejno w symbol typu `MovieClip` (rysunek 8.36).

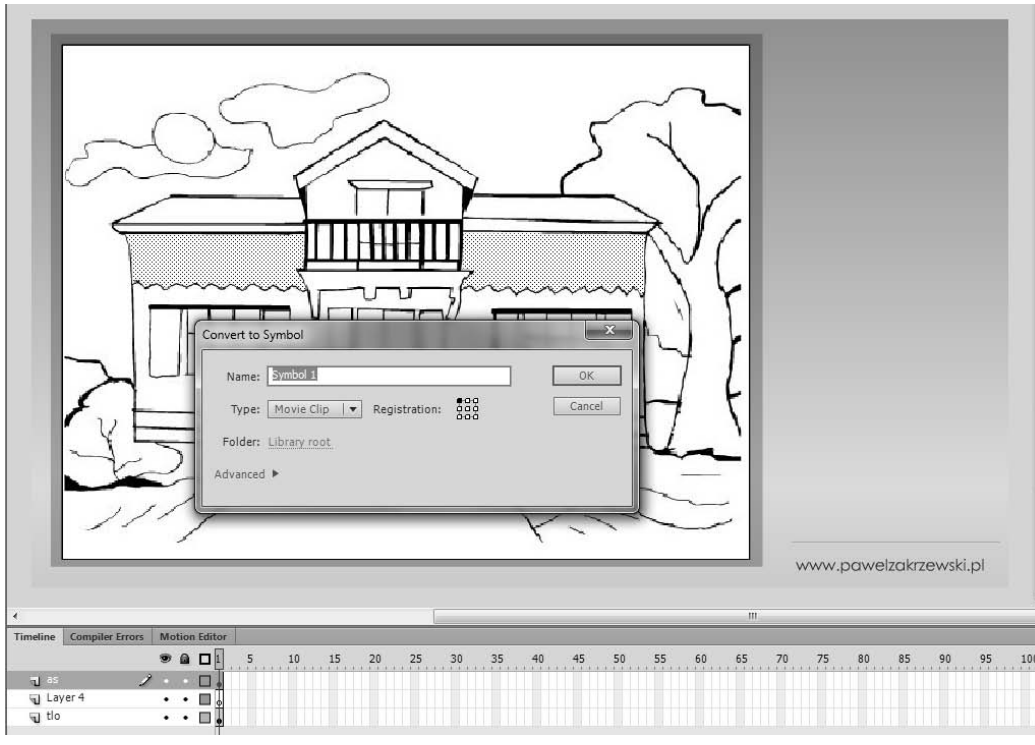


Rysunek 8.34. *Klikamy dowolny fragment obrazu. W rezultacie powinno pojawić się zaznaczenie o nieregularnym kształcie, które powstało na bazie wybrania jednolitego koloru — bieli pomiędzy czarnymi liniami*

Jeśli w niektórych miejscach brakuje precyzji, aby rozdzielić pewne fragmenty obrazu, możemy pokusić się o zmianę tolerancji różdżki. Umożliwi to przycisk *Magic Wand Settings*, dostępny w dolnej części palety *Tools* po wybraniu narzędzia *Lasso* (rysunek 8.37). Większe wartości pozwalają na zmniejszenie czułości i zaznaczanie większych obszarów obrazu. Mniejsze wartości pozwalają zaznaczać mniejsze fragmenty. Zwykle trzeba przeprowadzić kilka prób, aby odnaleźć dobre ustawienie dla konkretnej pracy. W naszym przykładzie korzystałem z tolerancji o wartości 10, co dało mi całkiem zadowalające rezultaty.

Innym sposobem na rozdzielenie konkretnych fragmentów obrazu jest użycie programu graficznego typu Adobe Photoshop i przygotowanie tam osobnych kawałków obrazu na oddzielnych warstwach (rysunek 8.38). Tak importowany plik PSD możemy od razu w chwili importu podzielić na osobne symbole typu `MovieClip` (rysunek 8.39).

Niestety każdy obiekt, który chcemy później malować, powinien być osobnym symbolem z białym wypełnieniem. Nie możemy wykorzystywać tu obiektów budowanych wyłącznie na bazie konturów.

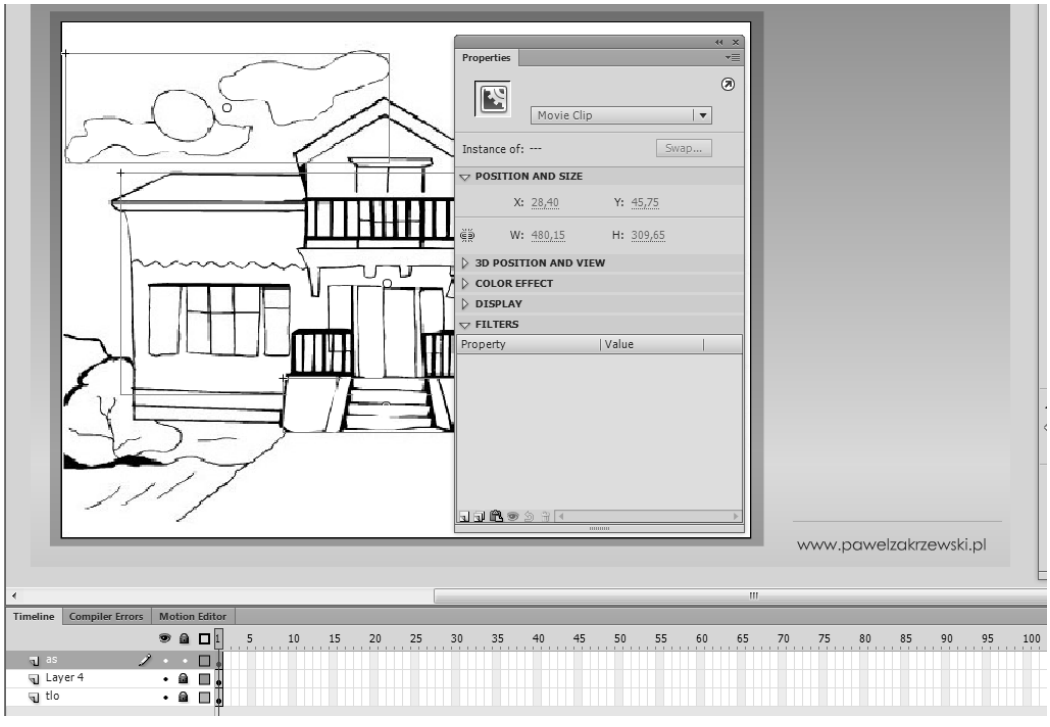


Rysunek 8.35. *W rezultacie aktywne są obie ściany budynku, które przekształcimy w pojedynczy symbol. W tym celu, korzystając z polecenia F8, wywołujemy polecenie Convert to symbol i na bazie przygotowanego zaznaczenia budujemy symbol typu MovieClip*

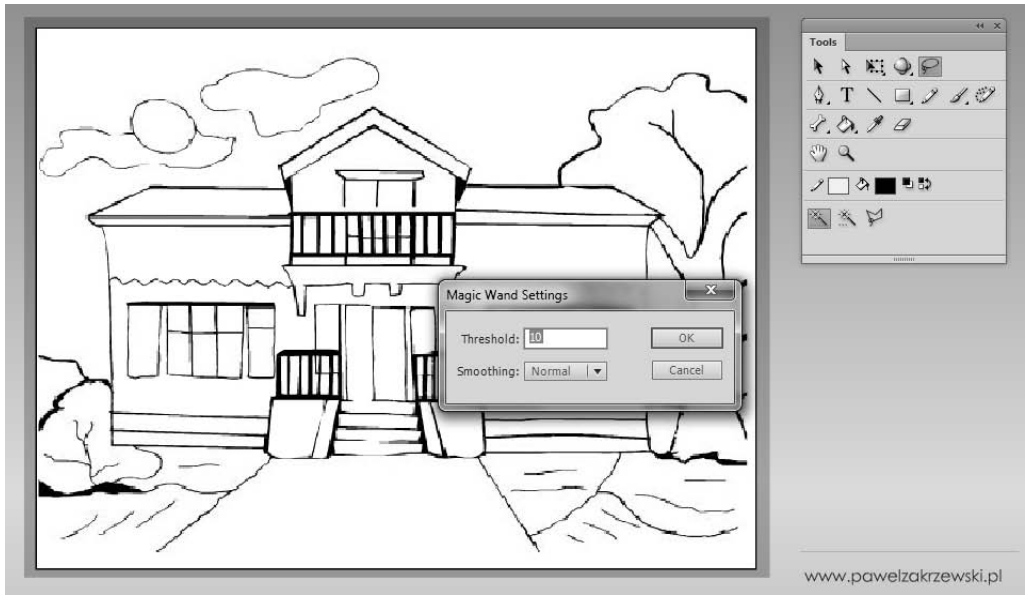
Jest to nieco mozolna praca i wymaga sporej dozy cierpliwości i precyzji. Efekty są jednak niezwykle przyjemne. Dodając bowiem kilka linii kodu, możemy przygotować całkiem przyjemną kolorowankę dla dzieci. Gdyby nie tworzenie tych symboli…

Gotowy rysunek (złożony z serii przygotowanych obiektów typu `MovieClip`) zaznaczamy w całości i przekształcamy w pojedynczy symbol typu `MovieClip`. Nadajemy mu nazwę instancji — `obrazek` (rysunek 8.40).

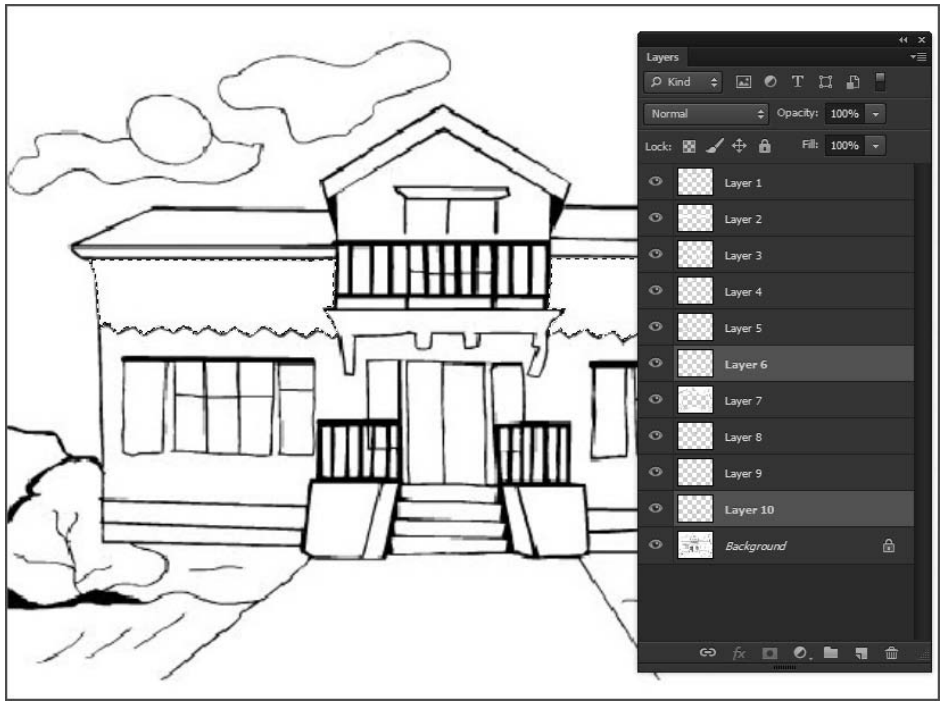
Aby nadać naszej pracy bardziej atrakcyjną postać, możemy pokusić się o dodanie niezbędnych elementów tła, logotypów, tekstów itp. Ostatnim ważnym elementem będzie jeszcze instancja komponentu `ColorPicker`, która pozwoli nam na zmianę koloru wypełnienia. Nadajemy jej nazwę `CPKolor` i blokujemy wszystkie warstwy naszego projektu (rysunek 8.41).



Rysunek 8.36. *Każdy fragment, który chcemy malować oddzielnie, przekształcamy kolejno w symbol typu MovieClip*

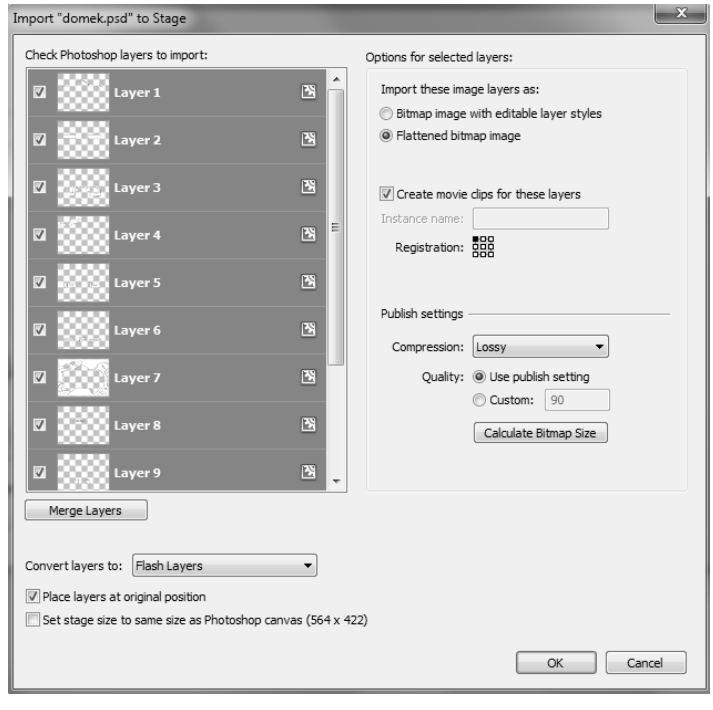


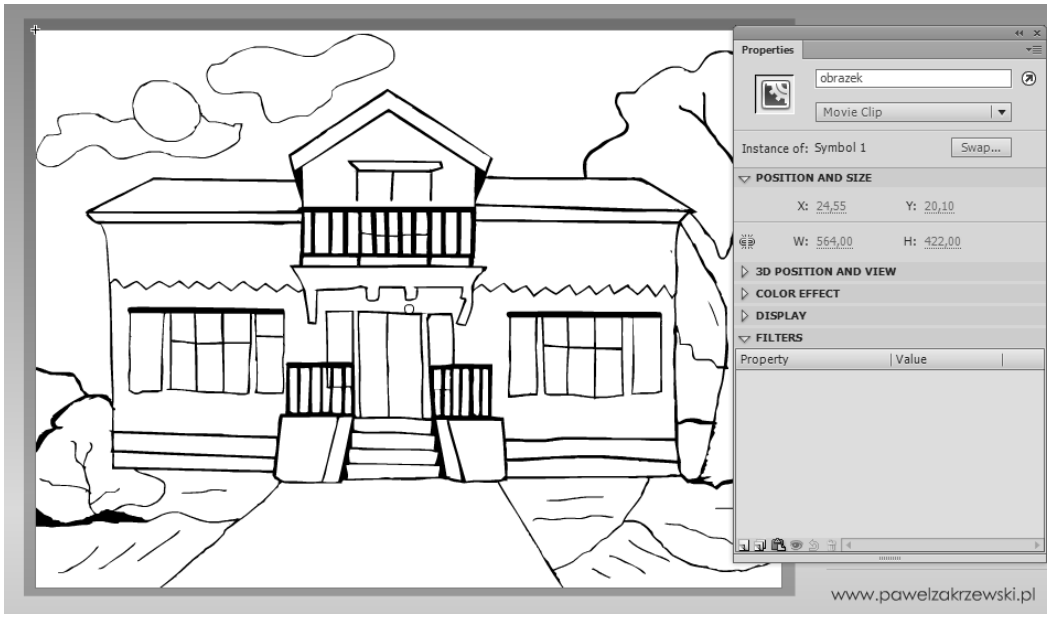
Rysunek 8.37. *Jeśli w niektórych miejscach brakuje precyzji, aby rozdzielić pewne fragmenty obrazu, możemy pokusić się o zmianę tolerancji różdżki. Umożliwi to przycisk Magic Wand Settings, dostępny w dolnej części palety Tools po wybraniu narzędzia Lasso*



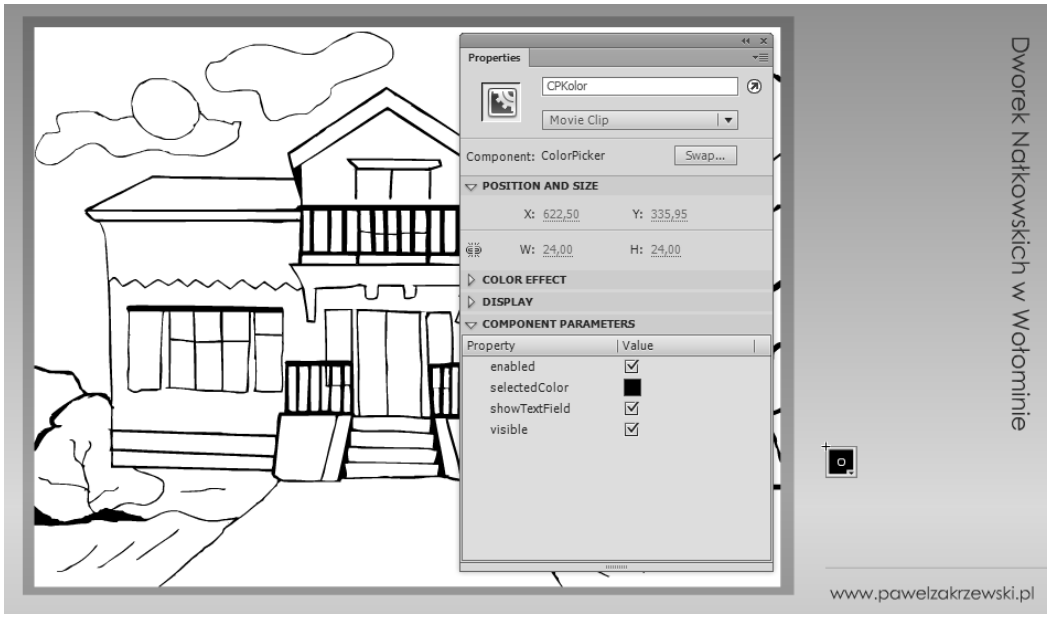
Rysunek 8.38. *Innym sposobem na rozdzielenie konkretnych fragmentów obrazu jest użycie programu graficznego typu Adobe Photoshop i przygotowanie tam osobnych kawałków obrazu na oddzielnych warstwach*

Rysunek 8.39. *Importowany plik PSD możemy od razu w chwili importu podzielić na osobne symbole typu MovieClip*





Rysunek 8.40. *Gotowy rysunek (złożony z serii przygotowanych obiektów typu MovieClip) zaznaczamy w całości i przekształcamy w pojedynczy symbol typu MovieClip. Nadajemy mu nazwę instancji — „obrazek"*



Rysunek 8.41. *Ostatnim ważnym elementem będzie jeszcze instancja komponentu ColorPicker, która pozwoli nam na zmianę koloru wypełnienia. Nadajemy jej nazwę „CPKolor" i blokujemy wszystkie warstwy naszego projektu*

Dodajemy ActionScript do kolorowanki

Ostatni krok tego przykładu to dodanie dosłownie kilku linii kodu odpowiedzialnych za poprawne działanie przykładu. Prostota naszych działań wynagrodzi tu czas spędzony na wcześniejszym tworzeniu niezbędnych symboli.

Rozpoczynamy od dodania nasłuchiwania zdarzenia `CLICK`. Dodajemy je do całego klipa na scenie.

```
obrazek.addEventListener(MouseEvent.CLICK, malujObraz);
```

Aby możliwe było testowanie naszej zabawki, konieczne jest przygotowanie funkcji `malujObraz()` W tym przypadku wykorzystamy ponownie trik z użyciem obiektu, który generuje zdarzenie. Korzystając ze zmiennej `evt` i właściwości target, możemy łatwo wskazać symbol, który chcemy pokolorować bez konieczności użycia jego nazwy. Wykorzystamy tu właśnie kliknięty obiekt, a więc ten, który wygenerował zdarzenie. Stanie się on celem funkcji `setTint()`, odpowiedzialnej za nadanie koloru wskazanego klipa.

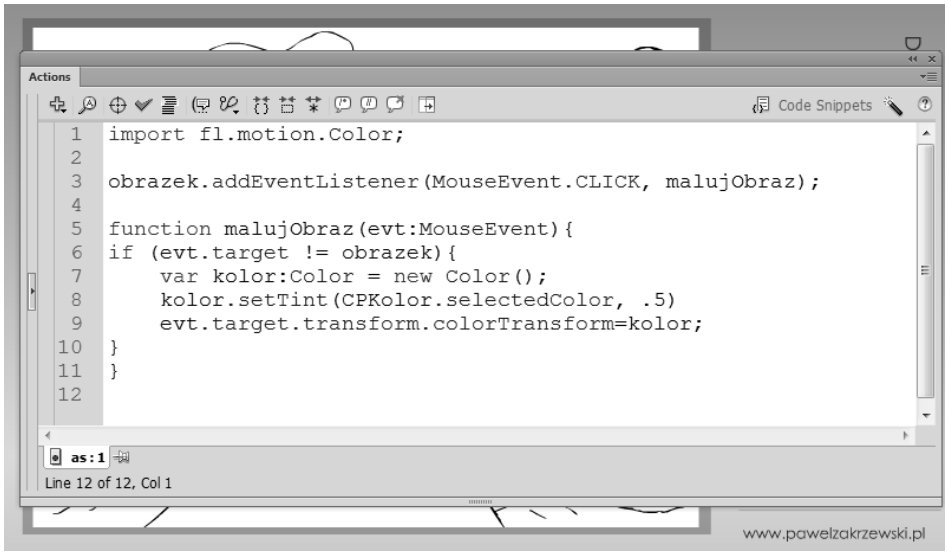
Do pracy z kolorem wykorzystamy (co raczej nie jest zaskoczeniem) klasę `Color` i jej metodę `setTint()`. Ta wymaga określenia dwóch parametrów. Pierwszym jest kolor, jaki chcemy wykorzystać do malowania, a drugim poziom jego krycia — `alpha`. Aby uniknąć zbyt intensywnych kolorów, parametr `alpha` przyjmuje w tym przypadku wartość `.5`, co pozwoli na malowanie symbolu z kryciem na poziomie 50%.

Aby ułatwić wybór właściwego koloru, parametr ten pobierzemy, korzystając z komponentu `ColorPicker` i jego właściwości `selectedColor`.

Do zmiany koloru obiektu wykorzystamy właściwości `transform` oraz `color ➥Transform`, które ostatecznie zmienią barwę klikniętego obiektu (rysunek 8.42).

```
import fl.motion.Color;

obrazek.addEventListener(MouseEvent.CLICK, malujObraz);

function malujObraz(evt:MouseEvent){
if (evt.target != obrazek){
  var kolor:Color = new Color();
  kolor.setTint(CPKolor.selectedColor, .5)
  evt.target.transform.colorTransform=kolor;
}
}
```

I to jest cały kod odpowiedzialny za działanie naszej kolorowanki — naprawdę jest go niezwykle mało!



Rysunek 8.42. *Do zmiany koloru obiektu wykorzystamy właściwości transform oraz colorTransform, które ostatecznie zmienią barwę kliknięte go obiektu*

Aby uniknąć kolorowania pozostałych elementów obrazu, zastosowałem tu dodatkową instrukcję warunkową, która pozwala zmieniać kolor wyłącznie przygotowanym wcześniej klipom, które zawarte są wewnątrz symbolu o nazwie `obrazek` (rysunek 8.43).

Rysunek 8.43. *Aby uniknąć kolorowania pozostałych elementów obrazu, zastosowałem tu dodatkową instrukcję warunkową, która pozwala zmieniać kolor wyłącznie przygotowanym wcześniej klipom zawartym wewnątrz symbolu o nazwie „obrazek". Gotowy projekt wygląda całkiem ciekawie*

Prosta gra typu ping-pong

Wiele, wiele lat temu, gdy pojawiły się pierwsze gry, zwane wtedy telewizyjnymi, ogromną popularnością cieszyła się bardzo prosta gra Ping-Pong. Gra zręcznościowa (niemal bez żadnej grafiki) potrafiła przyciągnąć graczy nawet na wiele długich godzin. Czy dzisiaj też tak może być?

W natłoku różnorodnych kolorowych gier, które atakują nas niemal z każdej strony, powrót do klasyki może być przyjemną rozrywką. W tym przykładzie pokusimy się o przygotowanie prostej gry wzorowanej na popularnym Ping-Pongu. Z założenia gra ta projektowana jest dla jednego użytkownika, jednak po drobnej modyfikacji mogą w nią pograć jednocześnie dwie osoby. Niestety nie mówię tu o przykładzie gry sieciowej, w której mogą rywalizować gracze z różnych miejsc na świecie, a o prostej formie gry dla dwóch graczy korzystających z tego samego komputera.

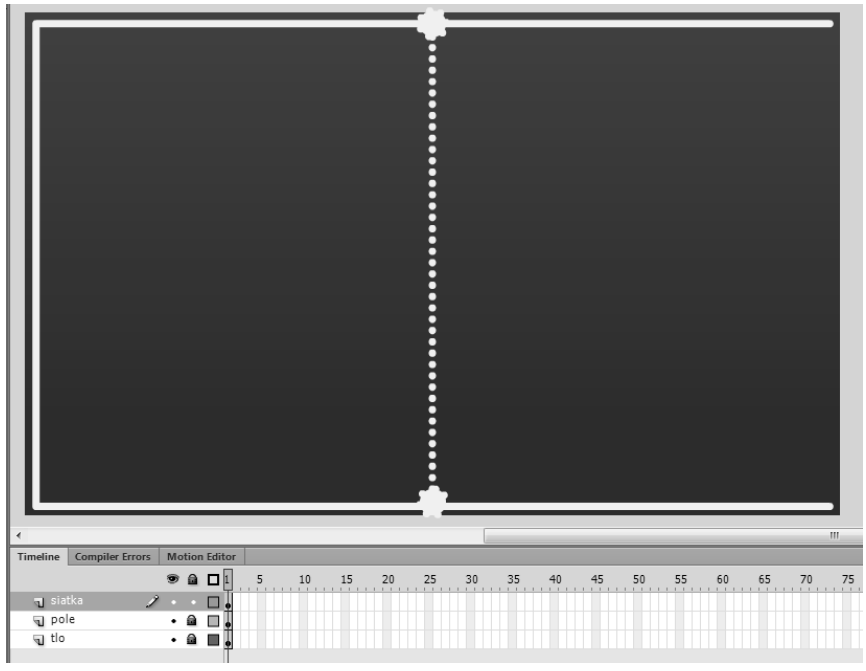
Budujemy ekran do gry

W przypadku naszej gry układ graficzny nie będzie odgrywał wielkiej roli. Możemy wykorzystać tu tło w całości przygotowane w zewnętrznym programie graficznym lub narysować prosty ekran o przyjemnej dla oka kolorystyce.

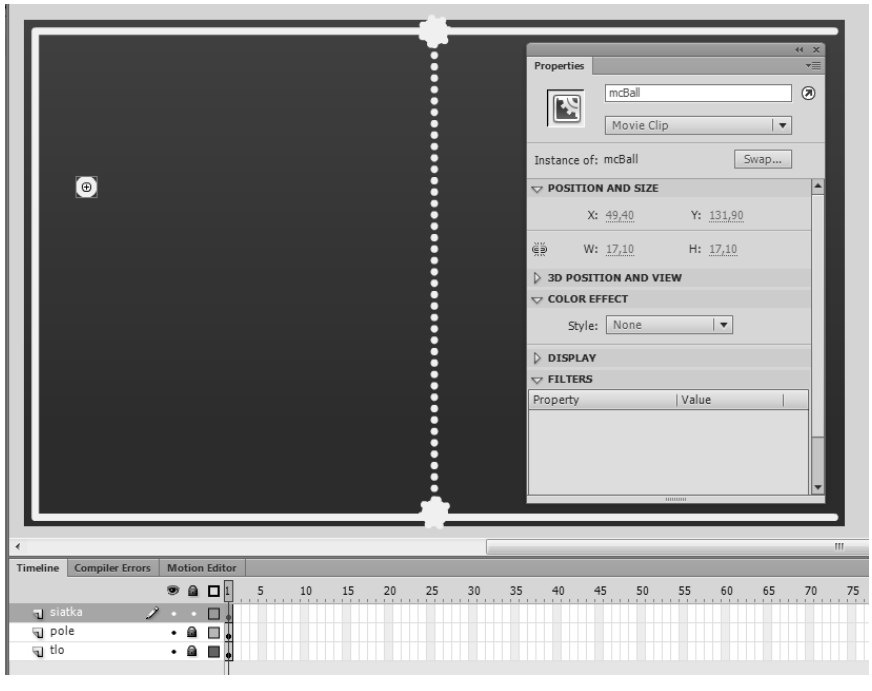
Najważniejszym elementem całej gry będzie tu pole gry. Możemy wykorzystać całą scenę, możemy też ograniczyć ją do wybranego obszaru. Oczywiście możemy pokusić się o dodanie linii zewnętrznych, siatki oraz innych elementów ozdobnych, nie będą one jednak wykorzystywane w samej grze (rysunek 8.44).

W naszym przykładzie pokusiłem się o narysowanie placu gry na bazie prostych kształtów wektorowych, które budowane są bezpośrednio w programie Flash. Nie ma tu żadnych specjalnych elementów, jedynie tło, które posłuży za podkład do właściwej gry. Kluczowym elementem jest za to symbol typu `MovieClip`, czyli piłeczka, którą rozgrywane będą nasze małe zawody. Jej kształt, kolor i wielkość nie mają istotnego znaczenia, liczy się jednak nazwa instancji. Nazwijmy ten symbol `mcBall` (rysunek 8.45).

Zasady naszej gry są raczej proste. Użytkownik steruje rakietką gracza po prawej stronie ekranu i nie może dopuścić do sytuacji, aby piłka przeleciała obok i znalazła się poza sceną po jej prawej stronie. Każda tego typu sytuacja to strata i utracony punkt. Cała gra odgrywana będzie na czas, a najlepszym graczem zostanie ten, kto straci najmniej punktów. Aby uatrakcyjnić nieco samą grę, możemy pokusić się o wprowadzenie kilku poziomów trudności, rakietek różnych rozmiarów czy elementu losowości w ruchu piłeczki. Rozpoczynamy jednak od głównej funkcjonalności.



Rysunek 8.44. *Najważniejszym elementem całej gry będzie pole gry. Możemy wykorzystać całą scenę, możemy też ograniczyć ją do wybranego obszaru*



Rysunek 8.45. *Kluczowym elementem jest za to symbol typu MovieClip, czyli piłeczka, którą rozgrywane będą nasze małe zawody. Jej kształt, kolor i wielkość nie mają istotnego znaczenia, liczy się jednak nazwa instancji. Nazwijmy ten symbol „mcBall"*

Na scenie mamy więc tło zablokowane na swej warstwie, zaś na kolejnej — symbol `MovieClip` o nazwie instancji `mcBall`. Naszym zadaniem będzie ożywić ów obiekt. W tym celu dodajemy nową warstwę i nadajemy jej nazwę instancji *as*. Otwieramy panel *Actions* i rozpoczynamy zabawę z kodem.

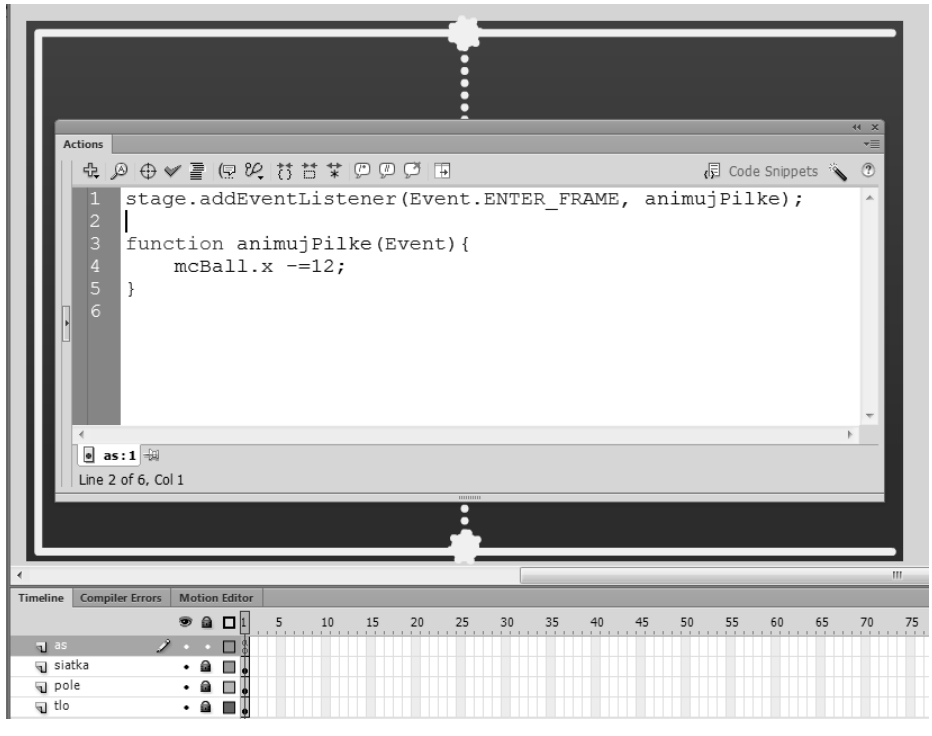
Animujemy piłeczkę

Najważniejszy w naszej zabawie jest animowany symbol, który za pomocą interaktywnej rakietki należy zatrzymać na scenie. W pierwszym kroku postaramy się dodać jeszcze najprostszą funkcjonalność — animację.

Wykorzystamy w tym celu zdarzenie `ENTER_FRAME`, którego nasłuchiwać będzie scena naszego projektu.

```
stage.addEventListener(Event.ENTER_FRAME, animujPilke);
```

Kolejny krok to przygotowanie funkcji `animujPilke()`, która odpowiedzialna jest za animację (rysunek 8.46).

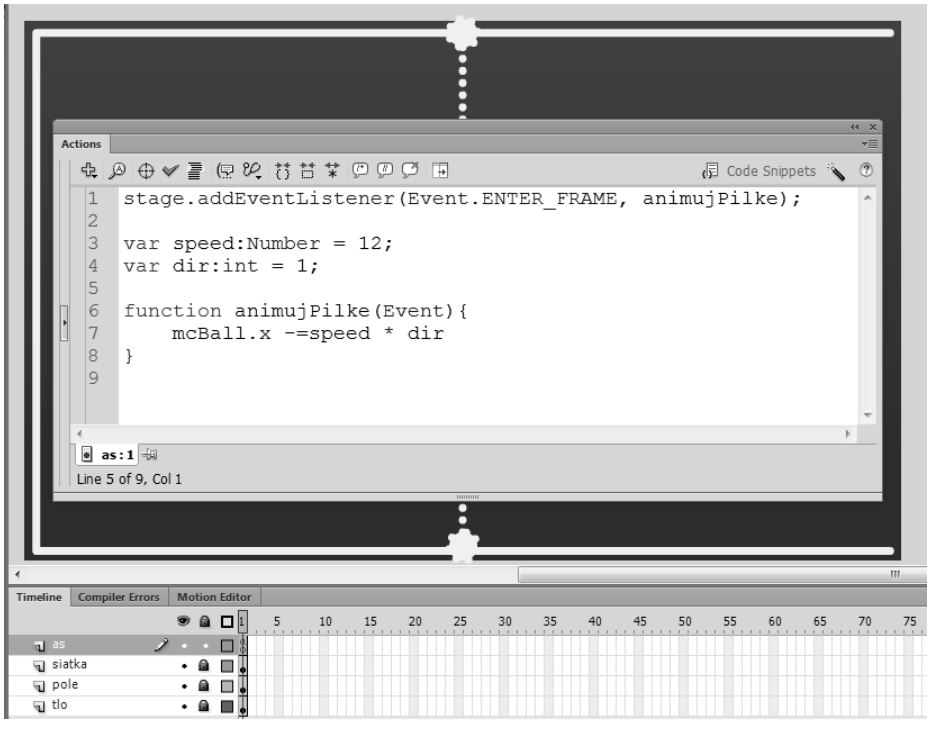


Rysunek 8.46. *Kolejny krok to przygotowanie funkcji animujPilke(), która odpowiedzialna jest za animację*

```
function animujPilke(Event){
  mcBall.x -=12;
}
```

W rezultacie użycia tego prostego kodu symbol `mcBall` porusza się w lewo i znika poza granicą sceny. Aby uniknąć tej sytuacji, musimy przygotować animację, która pozwoli na zmianę kierunku ruchu piłeczki w chwili, gdy ta osiągnie lewą krawędź sceny. Takie przykłady budowaliśmy już kilka rozdziałów wcześniej. Kluczem do tego efektu była tam dodatkowa zmienna `dir`, która przyjmując wartości 1 lub –1, odpowiednio modyfikowała kierunek animacji. Spróbujmy wykorzystać ten sam mechanizm także w tym przykładzie.

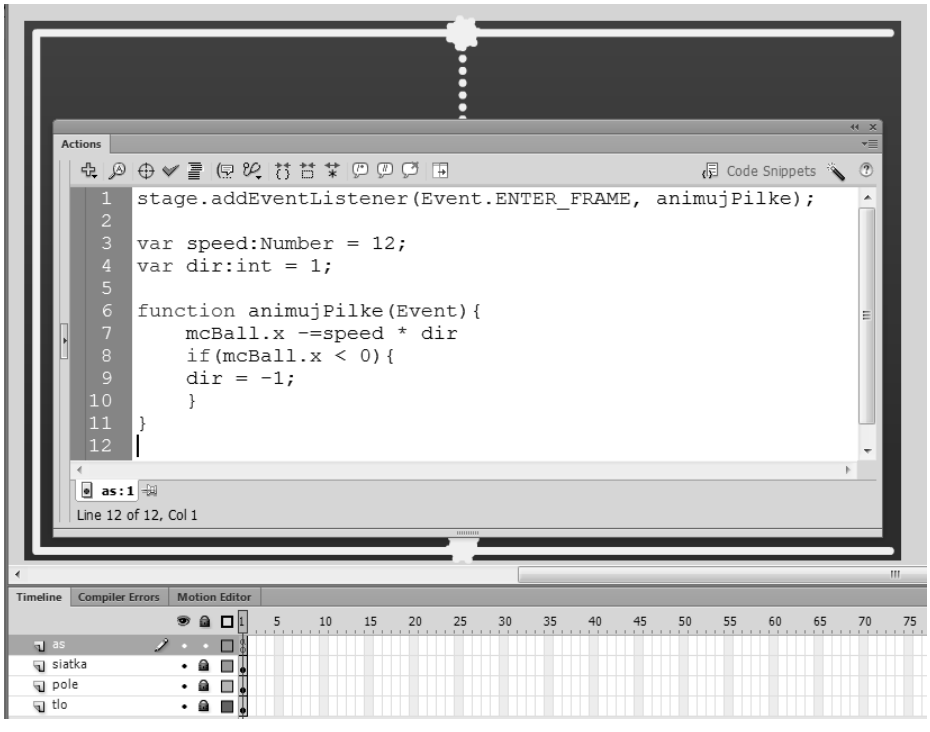
Deklarujemy więc zmienną `dir` i nadajemy jej wartość początkową 1. Aby wprowadzić większą kontrolę nad prędkością animacji, zadeklarujemy także dodatkową zmienną `speed`. Ta pozwoli nam na łatwą zmianę szybkości ruchu symbolu na scenie. Mnożąc prędkość animacji przez wartość zmiennej `dir`, otrzymamy zarówno wartości ujemne, jak i dodatnie — będzie to podstawą zmiany kierunku piłeczki na scenie (rysunek 8.47).



Rysunek 8.47. *Mnożąc prędkość animacji przez wartość zmiennej „dir", otrzymamy zarówno wartości ujemne, jak i dodatnie — będzie to podstawą zmiany kierunku piłeczki na scenie*

```
stage.addEventListener(Event.ENTER_FRAME, animujPilke);
var speed:Number = 12;
var dir:int = 1;
function animujPilke(Event){
  mcBall.x -=speed * dir
}
```

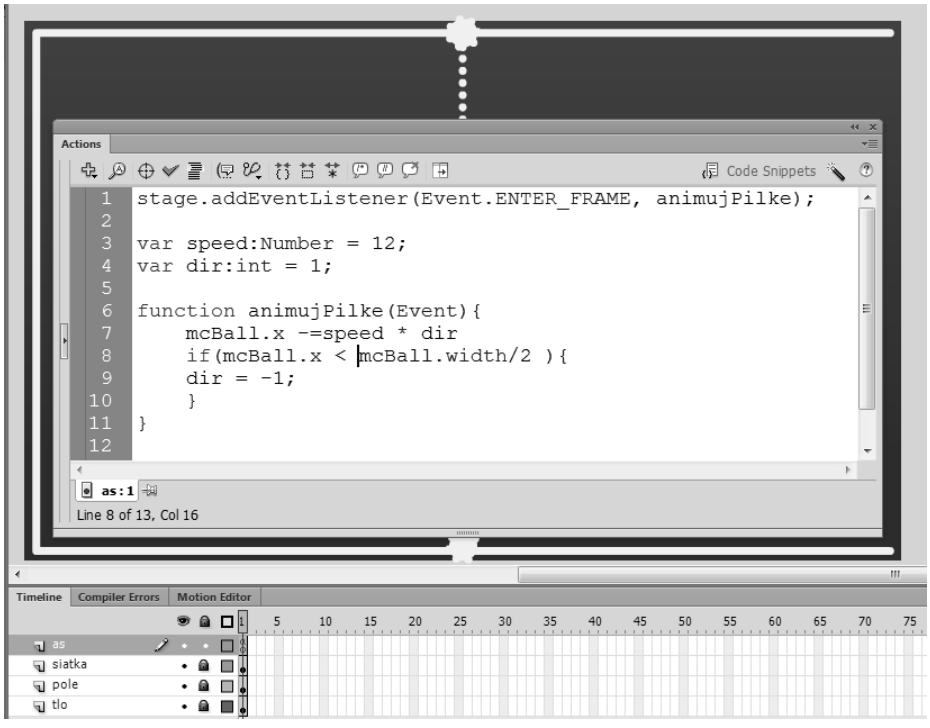
Klasycznym sposobem sprawdzenia, czy symbol wyszedł poza obszar sceny, są proste instrukcje warunkowe. Wykorzystamy je także w naszym przykładzie. Rozpoczynamy od wprowadzenia zmiany kierunku w chwili, gdy piłka spotka się z lewą krawędzią sceny (rysunek 8.48).



Rysunek 8.48. *Rozpoczynamy od wprowadzenia zmiany kierunku w chwili, gdy piłka spotka się z lewą krawędzią sceny*

```
function animujPilke(Event){
  mcBall.x -=speed * dir
  if(mcBall.x < 0){
  dir = -1;
  }
}
```

Już ten prosty fragment kodu umożliwia reakcję piłeczki na lewą krawędź sceny. W chwili gdy współrzędna `x` symbolu `mcBall` będzie mniejsza niż 0 (czyli lewa krawędź sceny), zmienna `dir` zmienia swój znak, co automatycznie wpływa na modyfikację kierunku animacji. W tej prostej instrukcji nie uwzględniamy na razie szerokości piłki, a warto by było. Aby uniknąć sytuacji, że piłka w połowie wychodzi poza obszar sceny, do naszego warunku należy dodać jeszcze połowę jej szerokości. Niewielka modyfikacja kodu daje dużo lepszy efekt (rysunek 8.49).

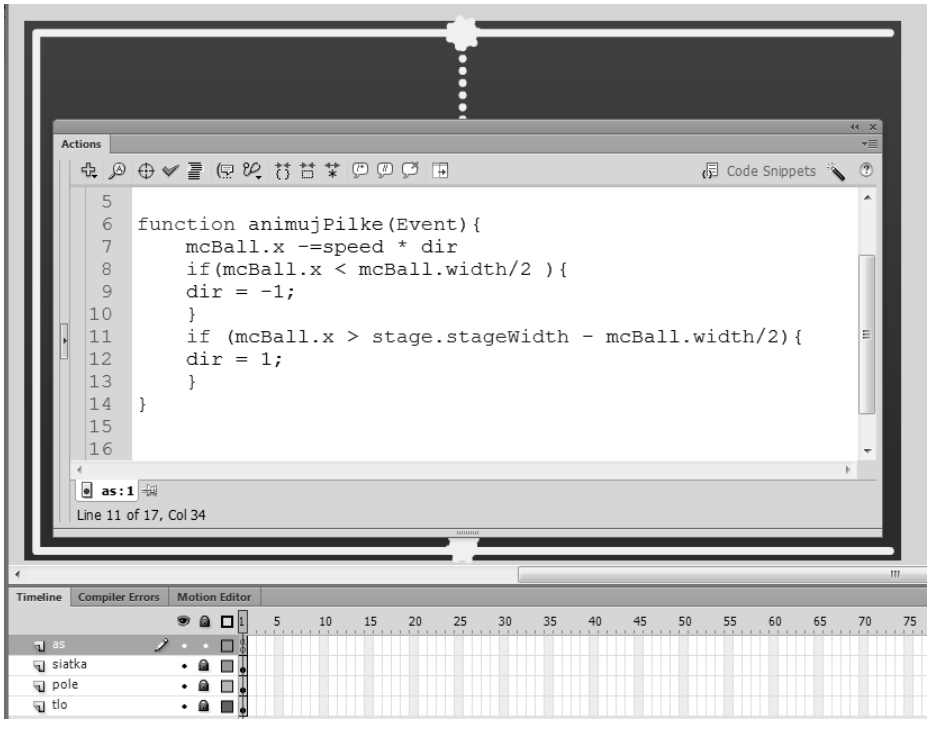


Rysunek 8.49. *W tej prostej instrukcji nie uwzględniamy na razie szerokości piłki, a warto by było. Aby uniknąć sytuacji, że piłka w połowie wychodzi poza obszar sceny, do naszego warunku należy dodać jeszcze połowę jej szerokości. Niewielka modyfikacja kodu daje dużo lepszy efekt*

```
function animujPilke(Event){
  mcBall.x -=speed * dir
  if(mcBall.x < mcBall.width/2 ){
  dir = -1;
  }
}
```

Aby ułatwić sobie testowanie gry, w tej chwili wprowadzimy także podobne ograniczenie na prawej krawędzi sceny. Docelowo w tym miejscu nie będzie żadnych ograniczeń co do animacji. Zmianę jej kierunku wygeneruje gracz, odbijając piłkę we właściwym momencie (rysunek 8.50).

```
function animujPilke(Event){
  mcBall.x -=speed * dir
  if(mcBall.x <mcBall.width/2 ){
  dir = -1;
  }
  if (mcBall.x > stage.stageWidth-mcBall.width/2){
  dir = 1;
  }
}
```

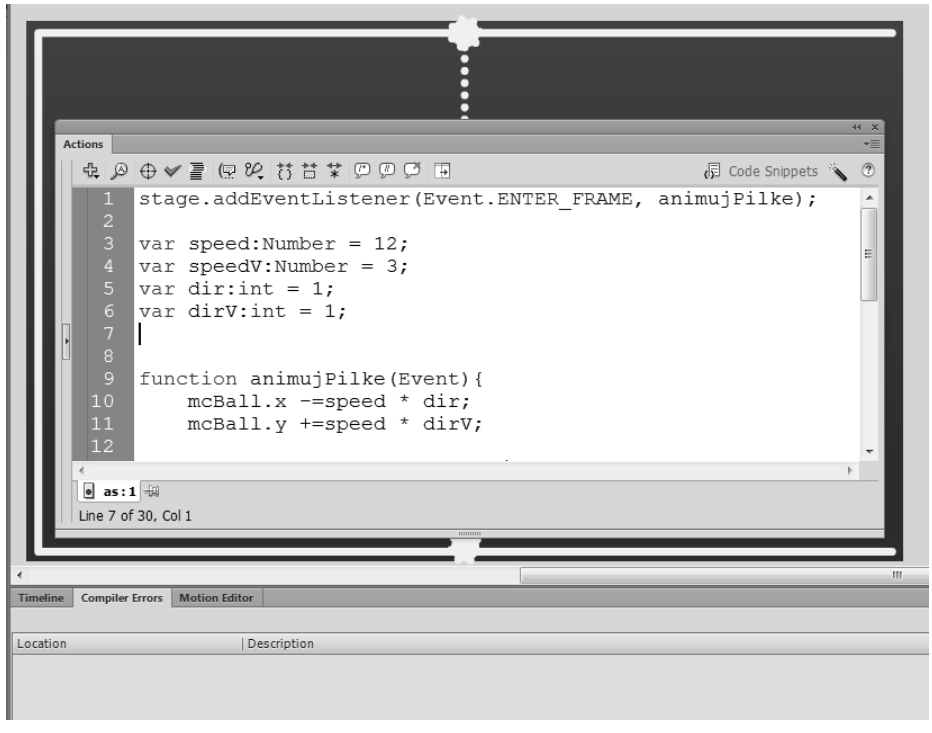


Rysunek 8.50. *Aby ułatwić sobie testowanie gry, w tej chwili wprowadzimy także podobne ograniczenie na prawej krawędzi sceny. Docelowo w tym miejscu nie będzie żadnych ograniczeń co do animacji. Zmianę jej kierunku wygeneruje gracz, odbijając piłkę we właściwym momencie*

Widoczny fragment wprowadza nam niekończącą się animację piłeczki w obu kierunkach na scenie. W tej chwili piłka porusza się stale po tym samym torze (na tej samej wysokości), co zupełnie nie nadaje się do naszej gry. Spróbujmy to zmienić, dodając nieco przypadkowych zmian kierunku animacji i uwzględniając zarówno ruch w pionie, jak i w poziomie. Takie działanie wymusi nam wprowadzenie podobnych ograniczeń w ruchu animacji w pionie. Piłeczka nie powinna wychodzić z obszaru sceny. Wymaga to dodania dodatkowej zmiennej `dirV` oraz wprowadzenia animacji także w pionie.

Aby możliwe było niezależne sterowanie zarówno prędkością, jak i kierunkiem animacji w pionie i w poziomie, wprowadzimy także dodatkową zmienną `speedV` określającą prędkość animacji w pionie. W wyniku wprowadzenia niezbędnych poprawek cały kod jest teraz nieco dłuższy. Pozwala jednak na zapętlenie animacji na scenie, tak że obiekt w chwili zetknięcia się z dowolną krawędzią zmienia swój kierunek ruchu. Niebawem dodamy tu jeszcze element przyspieszania, dzięki czemu animacja stanie się zdecydowanie ciekawsza, a co za tym idzie — trudniejsza.

Najprostszym sposobem na realizację przyspieszania jest wykorzystanie efektu mnożenia bieżącej prędkości (zmienna `speed` oraz `speedV`) przez niewielką liczbę typu `1.01`. Jeśli przyspieszanie realizowane będzie przy każdym odbiciu piłeczki od dowolnej ścianki, przyrost prędkości stanie się coraz bardziej zauważalny. Inaczej mówiąc, im dłużej pogramy, tym poziom trudności będzie z każdą chwilą wyższy (rysunek 8.51).



Rysunek 8.51. *Najprostszym sposobem na realizację przyspieszania jest wykorzystanie efektu mnożenia bieżącej prędkości (zmienna „speed" oraz „speedV") przez niewielką liczbę typu 1.01. Jeśli przyspieszanie realizowane będzie przy każdym odbiciu piłeczki od dowolnej ścianki, przyrost prędkości stanie się coraz bardziej zauważalny*

Przygotowanie takich modyfikacji nie jest trudne. Wystarczy wprowadzić kilka zmian wewnątrz warunków, a nasza gra stanie się dużo bardziej atrakcyjna.

```
var speed:Number = 12;
var speedV:Number = 3;
var dir:int = 1;
var dirV:int = 1;

function animujPilke(Event){
  mcBall.x -=speed * dir;
  mcBall.y +=speed * dirV;

  if(mcBall.x <mcBall.width/2 ){
```

```
    dir = -1;
    speed *=1.01
    }
    if (mcBall.x > stage.stageWidth-mcBall.width/2){
    dir = 1;
    speed *=1.02
    }
    if(mcBall.y < mcBall.width/2){
    dirV = 1;
    speedV *=1.1
    }
    if(mcBall.y > stage.stageHeight - mcBall.height/2){
    dirV = -1;
    speedV *=1.2
    }
  }
```

Dodajemy rakietkę użytkownika

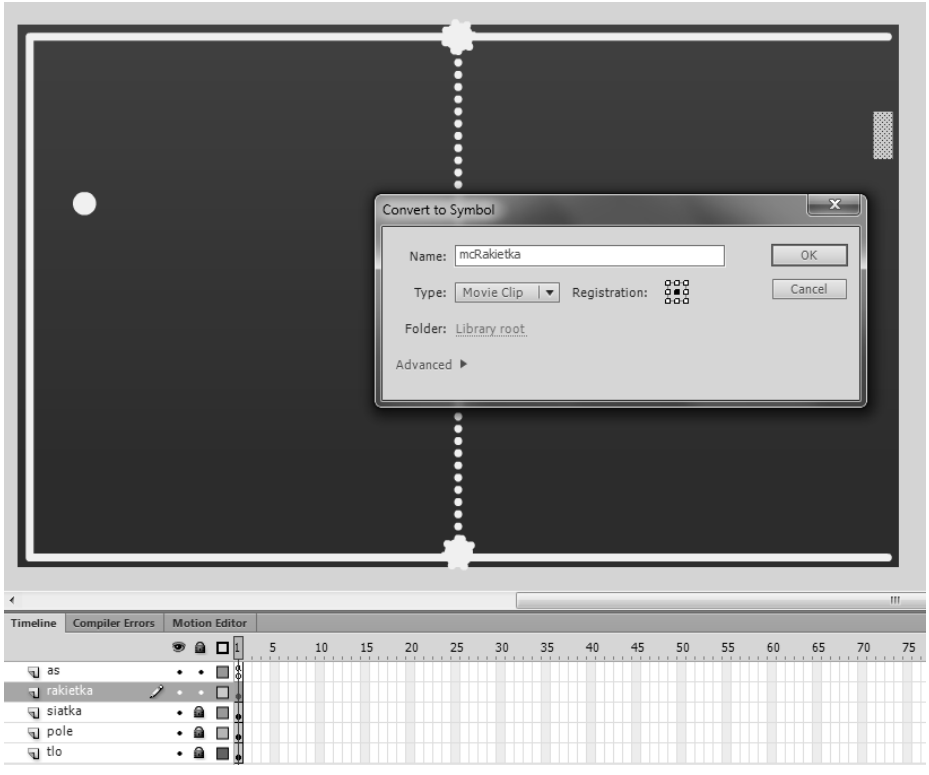
W chwili gdy piłeczka z powodzeniem animuje się na naszym ekranie, pora przejść do pracy nad interaktywną rakietką użytkownika. W tym celu importujemy lub rysujemy niewielki obiekt, który mógłby posłużyć nam do odbijania animowanej piłeczki. Warto zwrócić uwagę, aby obiekt ten nie był zbyt duży, ponieważ znacznie ułatwi to grę.

W naszym przykładzie pokusiłem się o przygotowanie obiektu na bazie kształtu klasycznej gry ping-pong. Będzie to po prostu niewielki prostokąt. Chcąc animować rakietkę za pomocą kodu, muszę przekształcić przygotowaną grafikę w symbol typu `MovieClip`. Najlepszym miejscem zaczepienia punktu *Registration* będzie niewątpliwie środek symbolu (rysunek 8.52).

Aby zarządzanie rakietką za pomocą kodu ActionScript było możliwe, nadajemy jej nazwę instancji — `mcRakietka` i umieszczamy ją nieopodal prawej krawędzi ekranu (rysunek 8.53). W tym miejscu użytkownik, korzystając z klawiszy na klawiaturze, będzie mógł przemieszczać obiekt w górę lub w dół, tak aby skutecznie odbić nadlatującą piłeczkę.

Sterowanie rakietką

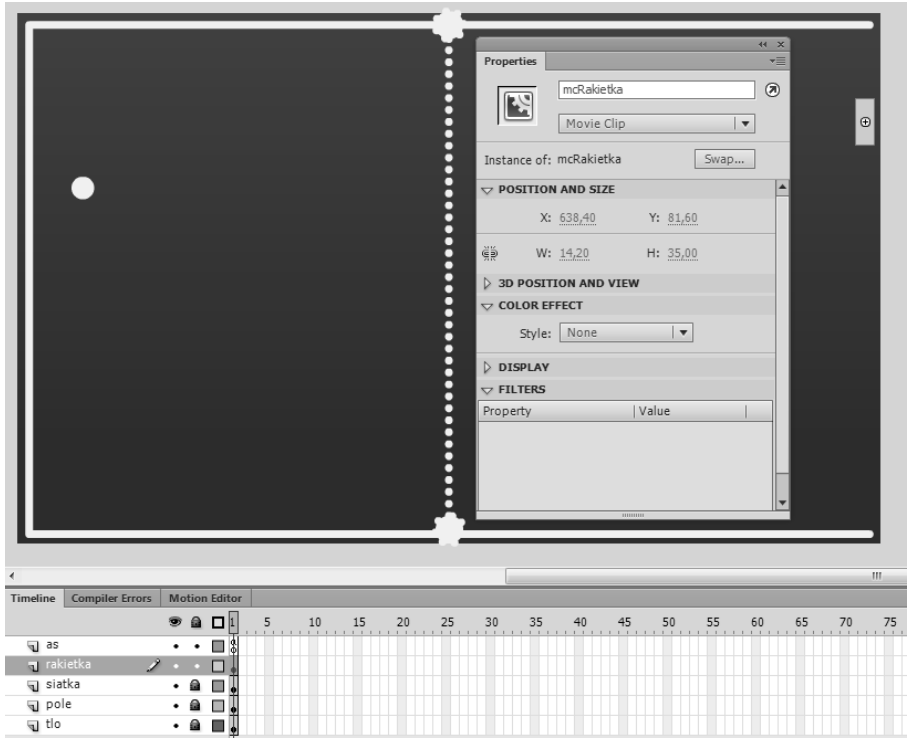
Do sterowania ruchem rakietki wykorzystamy klawisze ze strzałkami. Aby możliwa była ich obsługa, zastosujemy przedstawiony wcześniej wariant z nasłuchiwaniem wciśnięcia lub też zwolnienia danego klawisza. Wciśnięty klawisz pozwoli na szybkie przemieszczanie obiektu w danym kierunku. Zwolnienie klawisza zatrzyma ten ruch.



Rysunek 8.52. *W naszym przykładzie pokusiłem się o przygotowanie obiektu na bazie kształtu klasycznej gry ping-pong. Będzie to po prostu niewielki prostokąt. Chcąc animować rakietkę za pomocą kodu, muszę przekształcić przygotowaną grafikę w symbol typu MovieClip. Najlepszym miejscem zaczepienia punktu Registration będzie niewątpliwie środek symbolu*

Za obsługę klawiszy na naszej klawiaturze odpowiadają zdarzenia z kategorii `KeyboardEvent`. Najpopularniejsze z nich to oczywiście `KEY_DOWN` oraz `KEY_UP`, które odpowiadają odpowiednio za wciśnięcie lub zwolnienie dowolnego klawisza. W tym miejscu nie ma znaczenia, który klawisz został wciśnięty. Zdarzenia `KEY_DOWN` oraz `KEY_UP` reagują na użycie zupełnie dowolnego klawisza. Warto zwrócić uwagę na fakt, że Adobe Flash nie daje nam bezpośrednio zdarzenia, które informuje o wciśnięciu i przytrzymaniu wciśniętego klawisza. ActionScript pozwala na reakcję jedynie na wciśnięcie lub zwolnienie klawisza. Inne przydatne zdarzenia musimy przygotować samodzielnie.

Aby animacja rakietki przebiegała płynnie, ponownie wykorzystamy tu zdarzenie `ENTER_FRAME` i na jego bazie wprowadzimy nasz symbol w ruch. Rozpoczynamy jednak od nasłuchiwania wciśnięcia odpowiednich klawiszy na naszej klawiaturze. Podobnie jak we wcześniejszym przykładzie, najlepiej będzie wykorzystać tu obiekt typu `stage`.

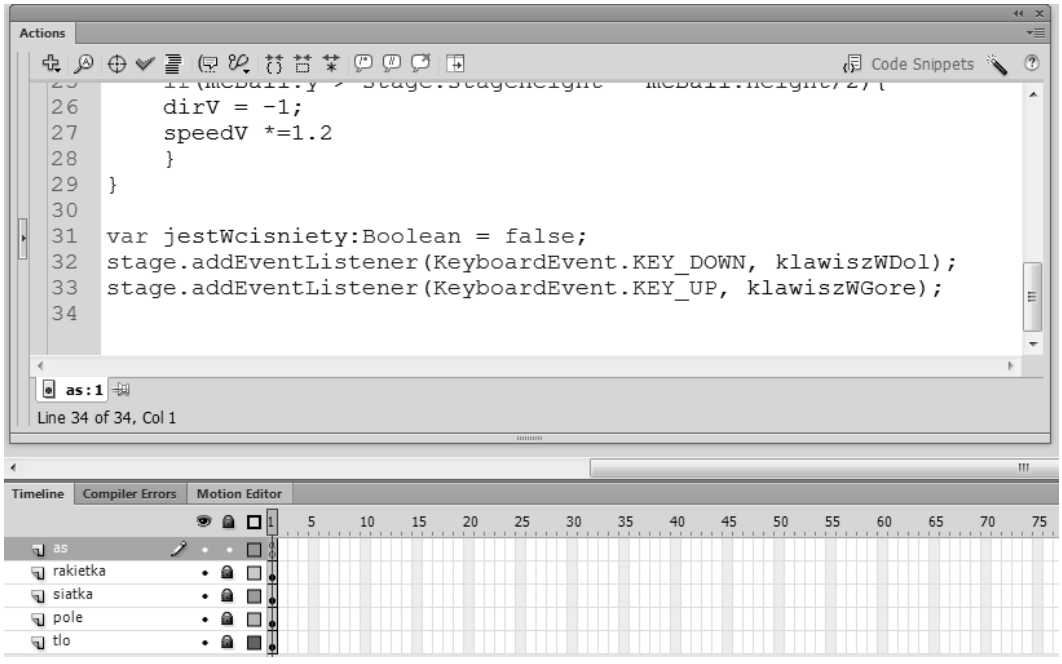


Rysunek 8.53. *Aby zarządzanie rakietką za pomocą kodu ActionScript było możliwe, nadajemy jej nazwę instancji — mcRakietka i umieszczamy ją nieopodal prawej krawędzi ekranu*

Wciśnięcie i zwolnienie właściwego klawisza spowoduje zmianę ruchu naszej rakietki na scenie. W jaki sposób połączymy teraz owe działania? Wykorzystamy tu dodatkową zmienną `jestWcisniety`, która pozwoli na zapis i przechowanie informacji o wciśnięciu lub zwolnieniu klawisza. Na tej podstawie przygotujemy odpowiednie animacje. Aby dodać zmienną o charakterze logicznym (`Boolean`), wprowadzimy jeszcze dodatkową linię kodu. W chwili uruchomienia naszej gry żaden klawisz nie będzie automatycznie wciśnięty, zatem wartość zmiennej `jestWcisniety` przyjmuje początkowo stan `false` (rysunek 8.54).

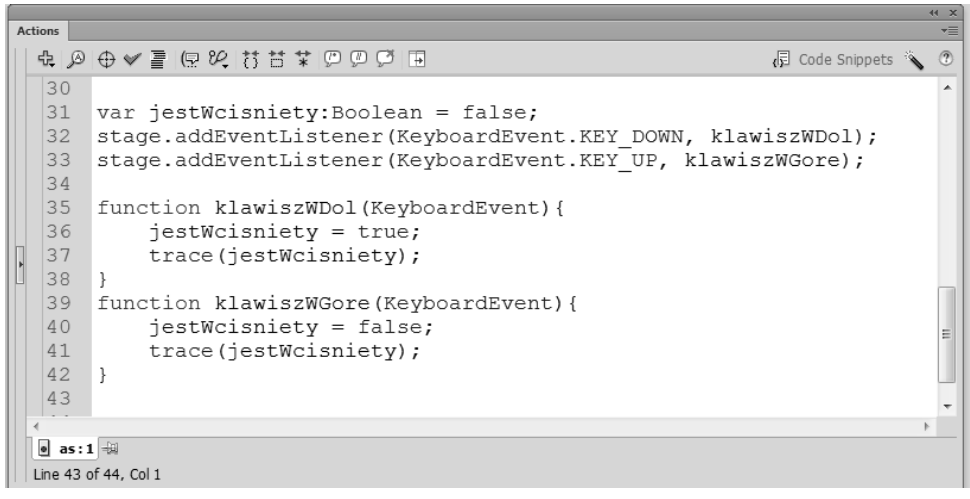
```
var jestWcisniety:Boolean = false;
stage.addEventListener(KeyboardEvent.KEY_DOWN, klawiszWDol);
stage.addEventListener(KeyboardEvent.KEY_UP, klawiszWGore);
```

Aby wprowadzony dodatkowo kod mógł zadziałać, konieczne jest oczywiście dodanie odpowiednich funkcji obsługi nasłuchiwanych zdarzeń. W tym miejscu warto zwrócić uwagę na pewną konsekwencję w konstrukcji nazw, jakie używam w naszych przykładach. Wprowadzając często te same nazwy obiektów, funkcji czy nawet własnych klas, oswajamy się z kodem ActionScript i popełniamy mniej błędów literowych. Nie bez znaczenia jest także fakt, że w wielu miejscach możemy po prostu skopiować i wkleić wykorzystany we wcześniejszej pracy kod bez konieczności wprowadzania niemal żadnych zmian.



Rysunek 8.54. *Aby dodać zmienną o charakterze logicznym (Boolean), wprowadzimy jeszcze dodatkową linię kodu. W chwili uruchomienia naszej gry żaden klawisz nie będzie automatycznie wciśnięty, zatem wartość zmiennej „jestWcisniety" przyjmuje początkowo stan false*

Obie funkcje mają za zadanie odpowiednio zmodyfikować wartość zmiennej. Jeśli wciskamy klawisz na klawiaturze, zmienna przyjmie wartość `true`, po zwolnieniu ponownie powróci do stanu `false` (rysunek 8.55).



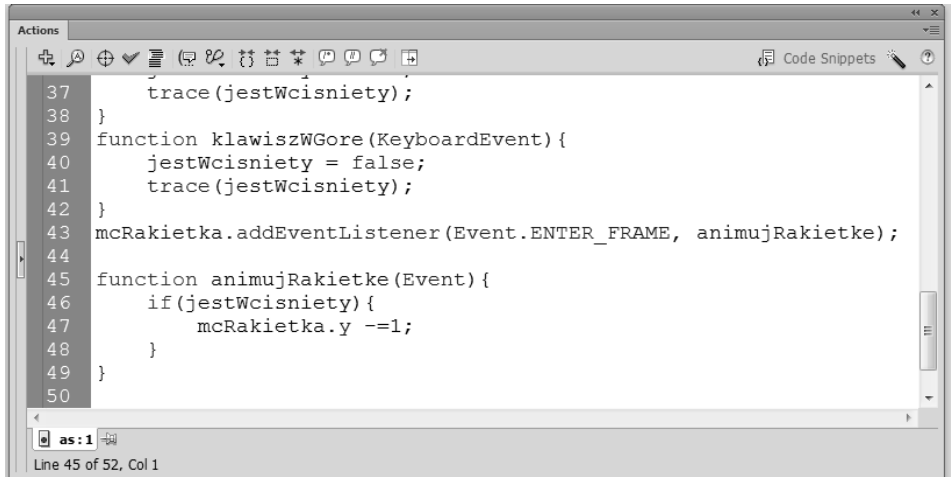
Rysunek 8.55. *Obie funkcje mają za zadanie odpowiednio zmodyfikować wartość zmiennej. Jeśli wciskamy klawisz na klawiaturze, zmienna przyjmie wartość true, po zwolnieniu ponownie powróci do stanu false*

```
function klawiszWDol(KeyboardEvent){
  jestWcisniety = true;
  trace(jestWcisniety);
}
function klawiszWGore(KeyboardEvent){
  jestWcisniety = false;
  trace(jestWcisniety);
}
```

Aby sprawdzić, czy cały kod działa poprawnie, możemy skorzystać z widocznych powyżej metod `trace()`. W chwili wciśnięcia lub zwolnienia klawisza myszki zmienia się wartość zmiennej `jestWcisniety`, a informacje te za pomocą metody `trace()` prezentujemy w oknie *Output*. Oczywiście tylko na chwilkę, aby się upewnić, że wszystko działa dobrze. Jeśli w wyniku wciśnięcia i puszczenia dowolnego klawisza na klawiaturze generowane są informacje typu: `true`, `false`, `true`, `false`, oznacza to, że przykład prawidłowo reaguje na nasze działania. W tej sytuacji instrukcję `trace()` możemy już usunąć z kodu.

Na bazie zmiennej `jestWcisniety` przygotujemy teraz samą animację. Nie będzie to trudne, choć wymaga wprowadzenia całkiem sporej dawki kodu. W pierwszej chwili dodajmy możliwość poruszania rakietką w górę w chwili wciśnięcia dowolnego klawisza. W momencie zwolnienia rakietka powinna się zatrzymać. Reakcję na wciśnięcie odpowiednich klawiszy dodamy niebawem.

Ponieważ animacja rakietki odbywać się będzie na bazie zdarzenia `ENTER_FRAME`, mamy tu dwie możliwości. Możemy wykorzystać albo istniejącą funkcję `animuj` `↪Pilke()` (działa ona przecież na bazie zdarzenia `ENTER_FRAME`), albo napisać od nowa własną. Aby ułatwić sobie czytanie kodu, przygotujemy nową funkcję do animacji rakietki. Wymaga to jednak dodania nasłuchiwania zdarzenia `ENTER_` `↪FRAME`. Wykorzystamy w tym miejscu samą rakietkę (rysunek 8.56).



Rysunek 8.56. *Aby ułatwić sobie czytanie kodu, przygotujemy nową funkcję do animacji rakietki. Wymaga to jednak dodania nasłuchiwania zdarzenia ENTER_FRAME. Wykorzystamy w tym miejscu samą rakietkę*

```
mcRakietka.addEventListener(Event.ENTER_FRAME, animujRakietke);
function animujRakietke(Event){
  if(jestWcisniety){
    mcRakietka.y -=1;
  }
}
```

Aby teraz wprowadzić odrębną reakcję na wciśnięcie klawisza ze strzałką w górę (ruch w górę) i dół (animacja w dół), musimy wprowadzić nasłuchiwanie kodu klawisza, który generował zdarzenia. W wielu wcześniejszych przykładach wykorzystywaliśmy w tym miejscu dodatkową zmienną `evt` oraz właściwość `target`. Tak zrobimy także w tym przypadku.

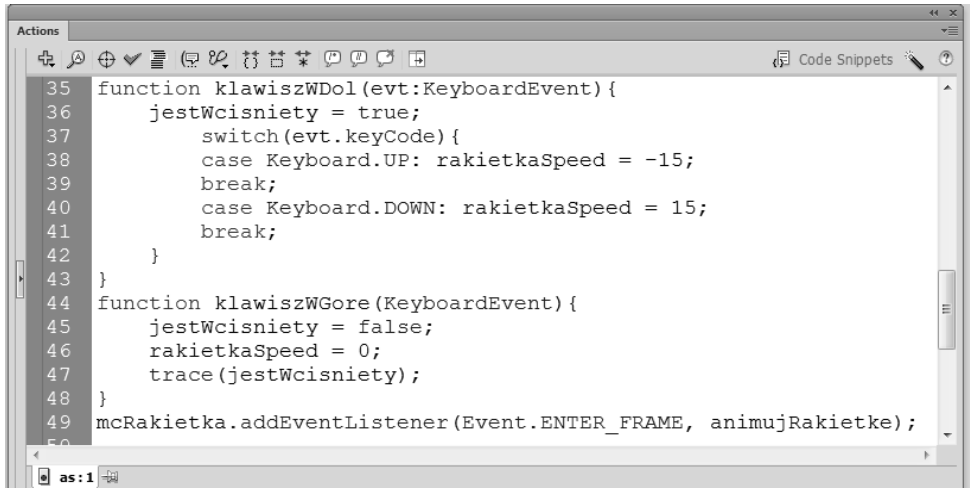
Spróbujmy jednak najpierw się zastanowić, jaki efekt mamy osiągnąć. Jeśli użytkownik wciśnie klawisz ze strzałką w górę, rakietka powinna przesunąć się w górę, jeśli wciśniemy strzałkę w dół, rakietka odpowiednio przesunie się w dół. Czy tego typu konstrukcja coś nam przypomina?

Jeżeli spełniony jest warunek, wykonaj to…

Jest to klasyczna instrukcja warunkowa, możemy więc wykorzystać w tym miejscu tradycyjny zapis z użyciem instrukcji `if` lub zapis z użyciem metody `switch()`. Oba sposoby dają te same efekty. Ponieważ instrukcja `if` nieco częściej pojawia się w naszych przykładach, spróbujmy w tym miejscu przećwiczyć wykorzystanie metody `switch()`. Gdzie jednak dodamy stosowne działania?

Całą logikę wykrywania właściwego klawisza przypiszemy wewnątrz funkcji `klawiszWDol()`. To ona reaguje przecież na wciśnięcie klawisza. W tym miejscu najłatwiej będzie przechwycić kod klawisza, który generował zdarzenie. Co jednak daje nam tego typu informacja? Jeśli wciskamy klawisz ze strzałką w górę, nasza rakietka powinna poruszać się (zmienić wartość współrzędnej `y`) w górę. Wartości jej współrzędnej `y` powinny zatem maleć (punkt 0,0 znajduje się w lewym górnym narożniku sceny). Jeśli wciskamy strzałkę w dół, wartość współrzędnej `y` powinna rosnąć. W obu wypadkach odwołujemy się więc do tej samej właściwości `y`, czyli pionowego położenia rakietki na scenie. Aby nie generować dodatkowego kodu, do reakcji na wciśnięcie odpowiedniego klawisza zastosujemy więc pojedynczą zmienną `rakietkaSpeed`, która w zależności od wybranego klawisza przyjmie wartość ujemną lub dodatnią. Co ważne, funkcja `klawiszWGore()` resetuje wartość zmiennej `rakietkaSpeed`, nadając jej ponownie wartość `0` w chwili, gdy zwolnimy klawisz. Zmienną `rakietkaSpeed` wykorzystamy także do animacji rakietki wewnątrz funkcji `animujRakietke()`. Rozpoczynamy od dodania zmiennej oraz instrukcji warunkowej umieszczonej wewnątrz funkcji `klawiszWDol()` (rysunek 8.57).

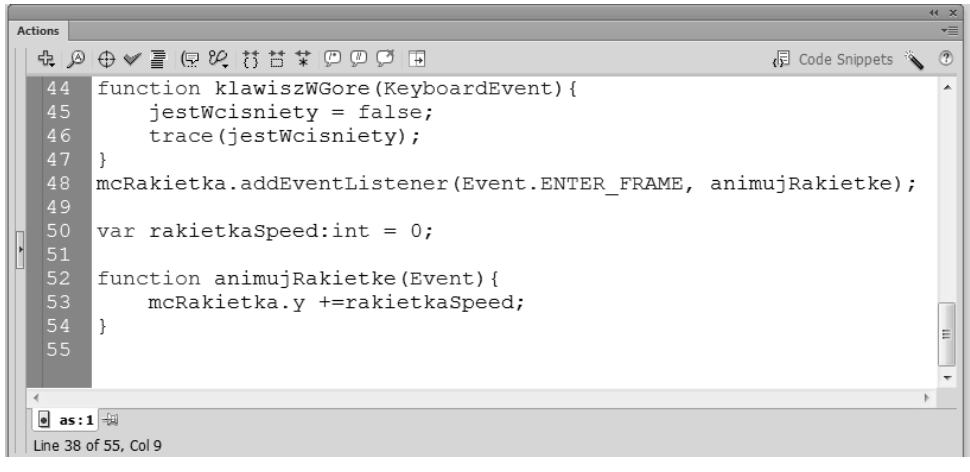
```
function klawiszWDol(evt:KeyboardEvent){
  jestWcisniety = true;
    switch(evt.keyCode){
    case Keyboard.UP: rakietkaSpeed = -15;
```



Rysunek 8.57. *Rozpoczynamy od dodania zmiennej oraz instrukcji warunkowej umieszczonej wewnątrz funkcji klawiszWDol()*

```
    break;
    case Keyboard.DOWN: rakietkaSpeed = 15;
    break;
  }
}
function klawiszWGore(KeyboardEvent){
  jestWcisniety = false;
  rakietkaSpeed = 0;
  trace(jestWcisniety);
}
```

Ostatni krok to użycie zmiennej `rakietkaSpeed` do budowy animacji rakietki na scenie. W tym celu wewnątrz funkcji `animujRakietke()` wprowadzamy stosowną zmianę (rysunek 8.58).

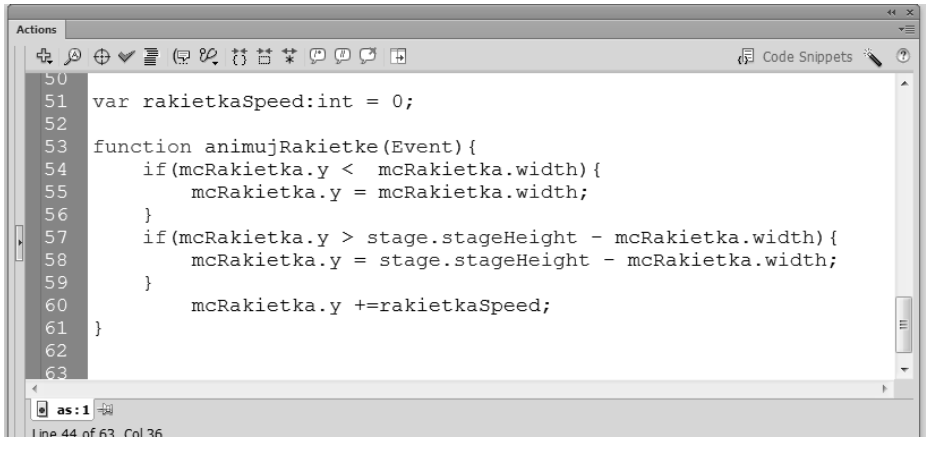


Rysunek 8.58. *Ostatni krok to użycie zmiennej „rakietkaSpeed" do budowy animacji rakietki na scenie. W tym celu wewnątrz funkcji animujRakietke() wprowadzamy stosowną zmianę*

```
function animujRakietke(Event){
  mcRakietka.y +=rakietkaSpeed;
}
```

W tej chwili cały kod naszej gry nieco się rozrósł. Nie jest jednak przesadnie skomplikowany. Pozwala już na animację piłki na scenie oraz sterowanie rakietką za pomocą odpowiednich klawiszy. Jeśli zależy nam, aby zmienić prędkość animacji rakietki, wystarczy zmodyfikować wartości zmiennej `rakietkaSpeed`. Wyższe wartości bezwzględne pozwalają na szybsze poruszanie symbolem na scenie i ułatwiają nam grę. Mniejsze wartości skutecznie utrudnią zabawę, szczególnie gdy ta nieco przeciągnie się w czasie i prędkość piłeczki wyraźnie wzrośnie.

Testując naszą zabawkę, warto zwrócić uwagę na fakt, że wszystko działa poprawnie, jednak w ferworze gry możemy doprowadzić do sytuacji, w której rakietka całkiem wyjedzie poza scenę. Nie jest to chyba zamierzony efekt. Aby uniknąć takiej sytuacji, należy wprowadzić ograniczenia brzegowe nieco podobne do tych, jakich używamy do utrzymania piłeczki w obrębie sceny (rysunek 8.59).



Rysunek 8.59. *Aby uniknąć takiej sytuacji, należy wprowadzić ograniczenia brzegowe nieco podobne do tych, jakich używamy do utrzymania piłeczki w obrębie sceny*

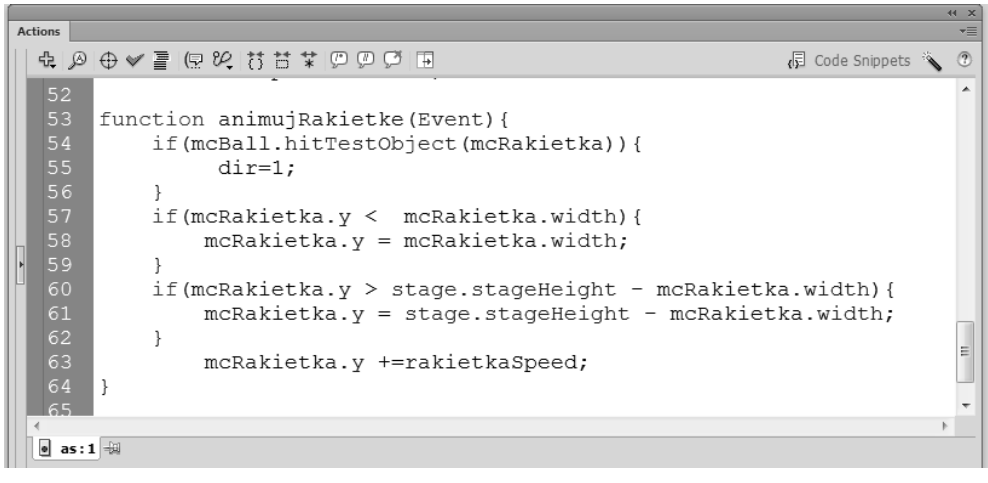
```
function animujRakietke(Event){
  if(mcRakietka.y <  mcRakietka.width){
    mcRakietka.y = mcRakietka.width;
  }
  if(mcRakietka.y > stage.stageHeight - mcRakietka.width){
    mcRakietka.y = stage.stageHeight - mcRakietka.width;
  }
    mcRakietka.y +=rakietkaSpeed;
}
```

Jeśli więc rakietka znalazła się poza sceną, automatycznie umieszczamy ją ponownie na scenie, umożliwiając dalszą animację. Warto zwrócić uwagę na fakt, że animacja obiektu przebiega niezależnie od warunków sprawdzających położenie symbolu na scenie.

Odbijanie piłeczki

Cały sens naszej gry sprowadza się do aktywnego odbijania piłeczki, gdy ta znajdzie się nieopodal prawej krawędzi strony. Gracz nie powinien dopuścić do sytuacji, aby piłka przeleciała poza scenę i nie była odbita rakietką.

Aby wprowadzić tę funkcjonalność, musimy zapoznać się z niezwykle popularną w grach metodą `hitTestObject()`. Jest to metoda, która pozwala na detekcję zderzenia się dwóch obiektów. W naszym przykładzie dzięki użyciu `hitTestObject()` możemy wprowadzić zmianę kierunku ruchu piłeczki w chwili zetknięcia z rakietką. Podobnie jak podczas sprawdzania położenia piłeczki czy też rakietki na scenie, także w tym przypadku wykorzystamy instrukcję warunkową `if`. Jeśli piłeczka dotyka rakietki, zmieniamy wartość zmiennej `dir`. Wówczas piłka zmienia kierunek ruchu i gra toczy się dalej. Całość dodajemy wewnątrz funkcji `animujRakietke()` (rysunek 8.60).

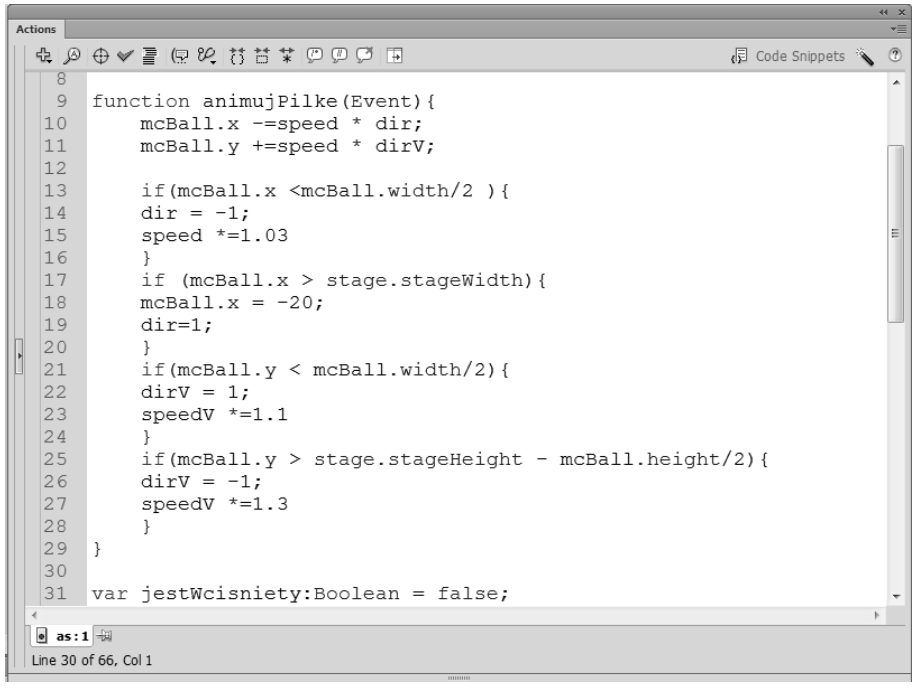


Rysunek 8.60. *Jeśli piłeczka dotyka rakietki, zmieniamy wartość zmiennej dir. Wówczas piłka zmienia kierunek ruchu i gra toczy się dalej. Całość dodajemy wewnątrz funkcji animujRakietke()*

```
function animujRakietke(Event){
  if(mcBall.hitTestObject(mcRakietka)){
     dir=1;
     }
  if(mcRakietka.y < 0){
    mcRakietka.y = 0;
  }
  if(mcRakietka.y >stage.stageHeight){
    mcRakietka.y = stage.stageHeight;
  }
    mcRakietka.y +=rakietkaSpeed;
}
```

W tym momencie podstawowe założenia naszej gry działają poprawnie. Piłeczka porusza się automatycznie pomiędzy krawędziami sceny. Użytkownik za pomocą wybranych klawiszy może sterować położeniem rakietki, a gdy zetknie się ona z nadlatującą piłką, zmienia kierunek animacji. Niestety to jeszcze nie koniec. W tej chwili, gdy nie uda nam się odbić piłki, ta automatycznie odbije się od krawędzi i gra toczy się dalej. Spróbujmy to zmienić, dodając nieco realizmu.

W chwili gdy piłeczka nie zostanie odbita i przeleci poza krawędź ekranu, postaramy się umieścić ją na scenie po lewej stronie. W ten sposób przypominać to będzie klasyczny serwis, a gra potoczy się dalej. Aby wszystko zadziałało należycie, musimy wprowadzić drobne zmiany w jednym z warunków wewnątrz funkcji `animujPilke()`. W chwili gdy piłka wychodzi ze sceny, umieszczamy ją tuż przed sceną z lewej strony i kontynuujemy animację (rysunek 8.61).



Rysunek 8.61. *W chwili gdy piłka wychodzi ze sceny, umieszczamy ją tuż przed sceną z lewej strony i kontynuujemy animację*

```
function animujPilke(Event){
  mcBall.x -=speed * dir;
  mcBall.y +=speed * dirV;

  if(mcBall.x <mcBall.width/2 ){
  dir = -1;
  speed *=1.03
  }
```

```
    if (mcBall.x > stage.stageWidth){
    mcBall.x = -20;
    dir=1;
    }
    if(mcBall.y < mcBall.width/2){
    dirV = 1;
    speedV *=1.1
    }
    if(mcBall.y > stage.stageHeight - mcBall.height/2){
    dirV = -1;
    speedV *=1.3
    }
}
```

W rezultacie nasza gra coraz bardziej zaczyna przypominać oryginalny pierwowzór. Brakuje nam jeszcze zliczania punktów i reakcji na nieodbitą piłkę. Spróbujmy zrobić to tak. W chwili gdy piłka wyleci ze sceny, użytkownik traci jedną piłkę, czyli szansę. Za każde odbicie zaliczony zostanie dodatkowy punkt. Gramy do chwili, gdy użytkownik zmarnuje swoją trzecią szansę. W ten sposób po utracie trzech piłek nasza gra automatycznie się zakończy. Wygrywa ten, kto zdobędzie najwięcej punktów, czyli najwięcej razy odbije piłeczkę.

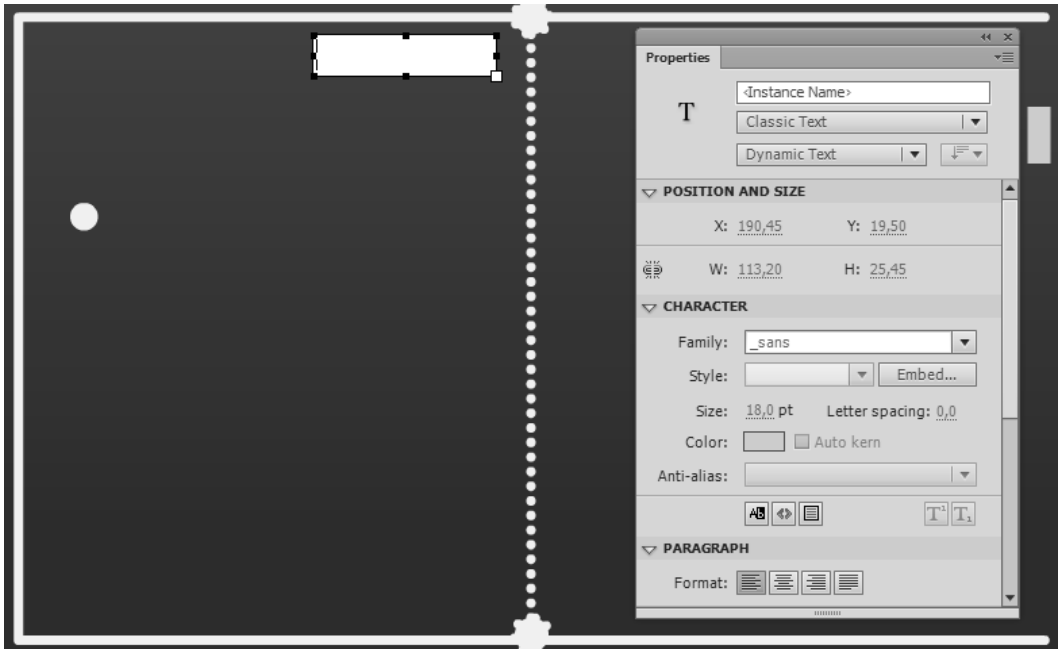
Dodajemy śledzenie wyniku i zakończenie gry

Do prezentacji wyniku powinniśmy przygotować na scenie dynamiczne pole tekstowe. Przechodzimy więc na scenę, dodajemy nową warstwę i rysujemy dynamiczne pole tekstowe o dowolnym rozmiarze. Korzystając z opcji formatowania tekstu, nadajemy mu odpowiedni kolor, wielkość oraz krój pisma. Jeśli nie zależy nam na szczególnej czcionce, warto wybrać w tym miejscu krój pisma *_sans* (rysunek 8.62), co oznacza domyślny krój bezszeryfowy komputera, na którym odtwarzana jest nasza gra. W większości przypadków będzie to *Arial* lub bardzo podobna czcionka. Nadajemy nazwę instancji naszego pola — `txtWynik` (rysunek 8.63).

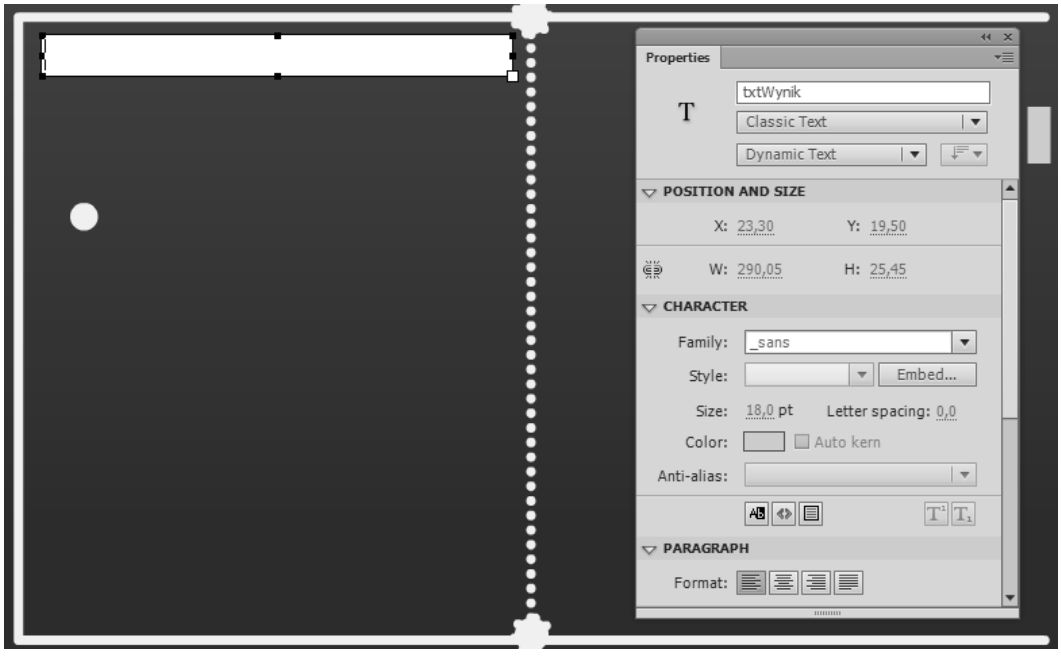
Upewnijmy się, że pole tekstowe jest wystarczająco duże (szczególnie szerokie), aby prezentować właściwy tekst. Jeśli podczas dalszych prób nie widać końcowego fragmentu tekstu, oznacza to, że pole jest po prostu zbyt małe. Całkowicie niewidoczny tekst może sugerować użycie pisma o kolorze tła.

Do prezentacji i zliczania punktacji wykorzystamy dodatkową zmienną — `wynik`. Dodajemy ją na początku kodu nieopodal deklaracji pozostałych zmiennych (rysunek 8.64).

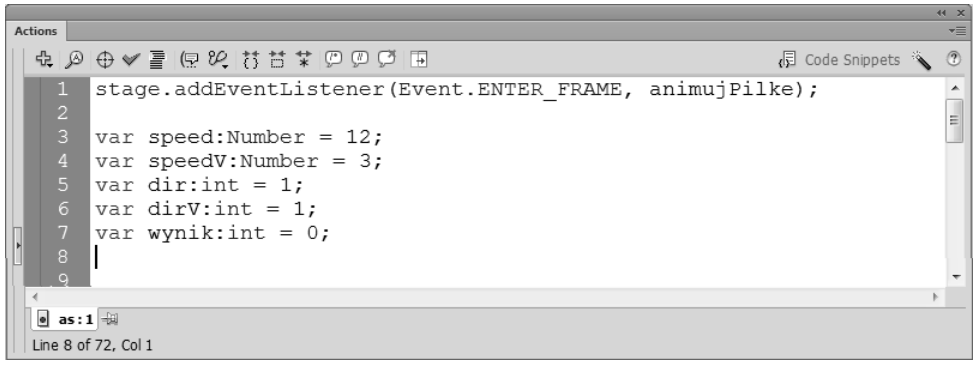
```
var speed:Number = 12;
var speedV:Number = 3;
var dir:int = 1;
var dirV:int = 1;
var wynik:int = 0;
```



Rysunek 8.62. *Korzystając z opcji formatowania tekstu, nadajemy mu odpowiedni kolor, wielkość oraz krój pisma. Jeśli nie zależy nam na szczególnej czcionce, warto wybrać w tym miejscu krój pisma _sans*

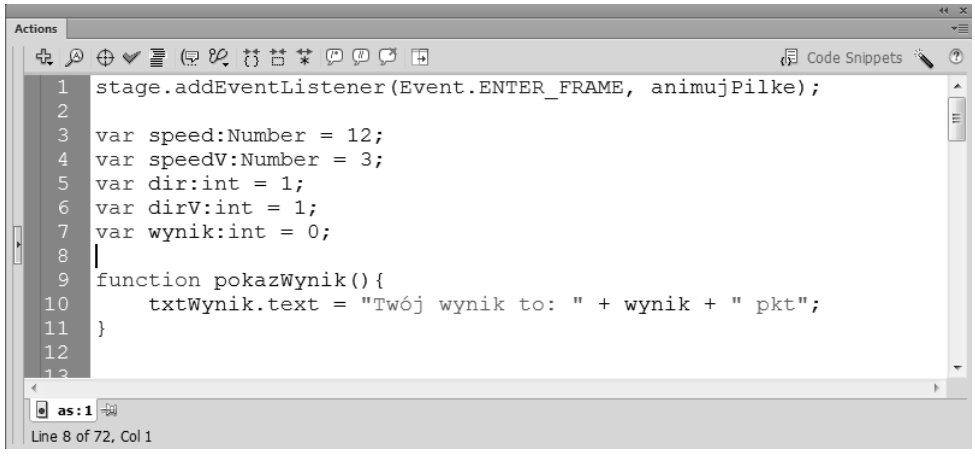


Rysunek 8.63. *Nadajemy nazwę instancji naszego pola — „txtWynik"*



Rysunek 8.64. *Do prezentacji i zliczania punktacji wykorzystamy dodatkową zmienną — „wynik". Dodajemy ją na początku kodu nieopodal deklaracji pozostałych zmiennych*

Aby w naszym polu wyświetlić stosowną informację o uzyskanym wyniku, przygotujemy własną funkcję. W ten sposób będziemy mogli wykorzystać ją także w innym miejscu. Istotą działania funkcji będzie wyłącznie prezentacja specjalnie formatowanego wyniku na scenie (rysunek 8.65).

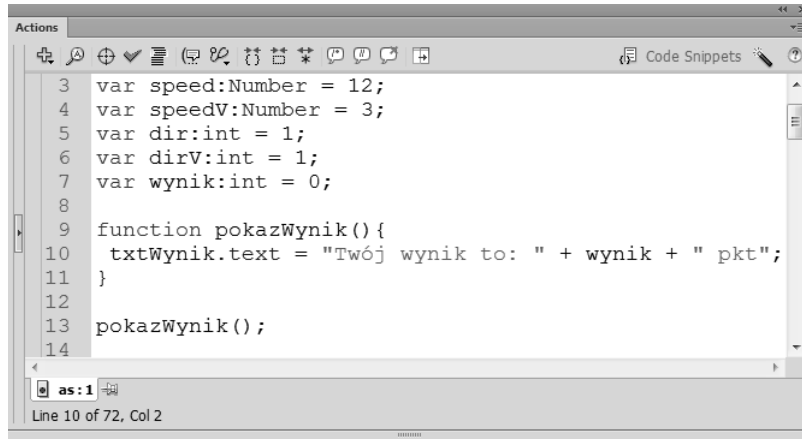


Rysunek 8.65. *Aby w naszym polu wyświetlić stosowną informację o uzyskanym wyniku, przygotujemy własną funkcję. W ten sposób będziemy mogli wykorzystać ją także w innym miejscu*

```
var speed:Number = 12;
var speedV:Number = 3;
var dir:int = 1;
var dirV:int = 1;
var wynik:int = 0;
function pokazWynik(){
  txtWynik.text = "Twój wynik to: " + wynik + " pkt";
}
```

Korzystając ze znaku + (plus), możemy stosunkowo łatwo łączyć różne teksty (i liczby jak w tym przypadku) w jeden ciąg tekstowy. Tego typu działanie zwane jest konkatenacją. Warto zwrócić tu uwagę na spacje zawarte wewnątrz cudzysłowów. To właśnie dzięki nim tekst prezentowany na scenie jest czytelny

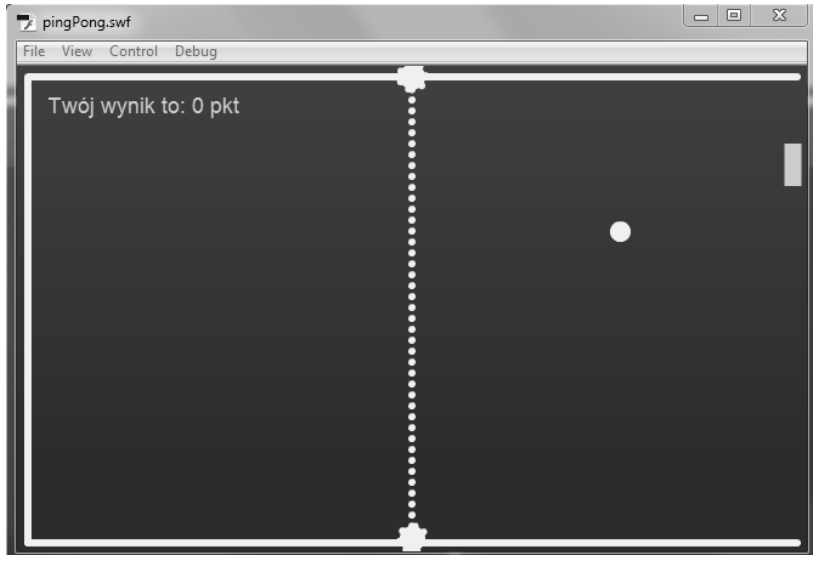
Naturalnie, aby przygotowana właśnie funkcja umożliwiła prezentację wyniku na scenie, musimy ją wywołać. Tuż poniżej dodajemy więc dodatkową linię kodu (rysunek 8.66).



Rysunek 8.66. *Naturalnie, aby przygotowana właśnie funkcja umożliwiła prezentację wyniku na scenie, musimy ją wywołać. Tuż poniżej dodajemy więc dodatkową linię kodu*

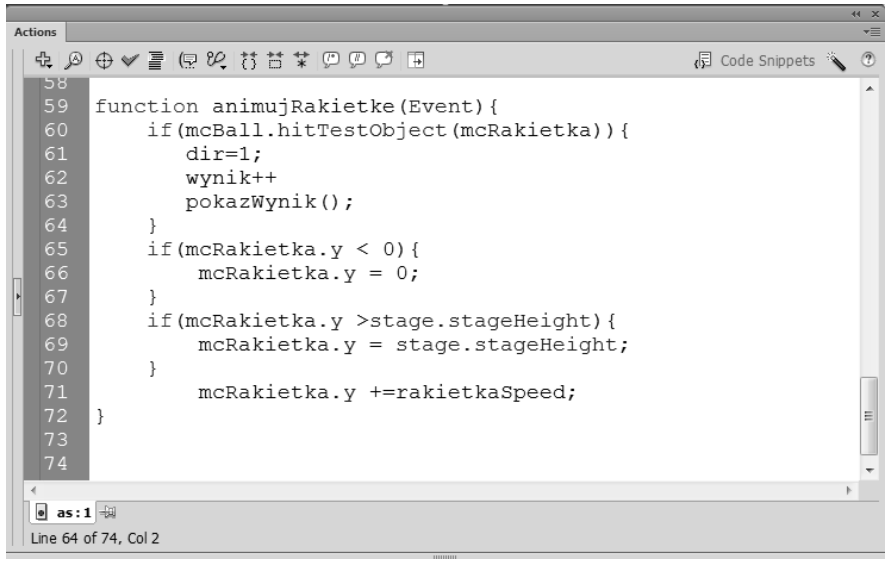
```
function pokazWynik(){
  txtWynik.text = "Twój wynik to: " + wynik + " pkt";
}
pokazWynik();
```

W ten sposób w chwili uruchomienia gry na ekranie widoczny jest tekst informujący o liczbie zdobytych przez nas punktów. Naturalnie w tym momencie jest to zero punktów (rysunek 8.67).



Rysunek 8.67. *W ten sposób w chwili uruchomienia gry na ekranie widoczny jest tekst informujący o liczbie zdobytych przez nas punktów. Naturalnie w tym momencie jest to zero punktów*

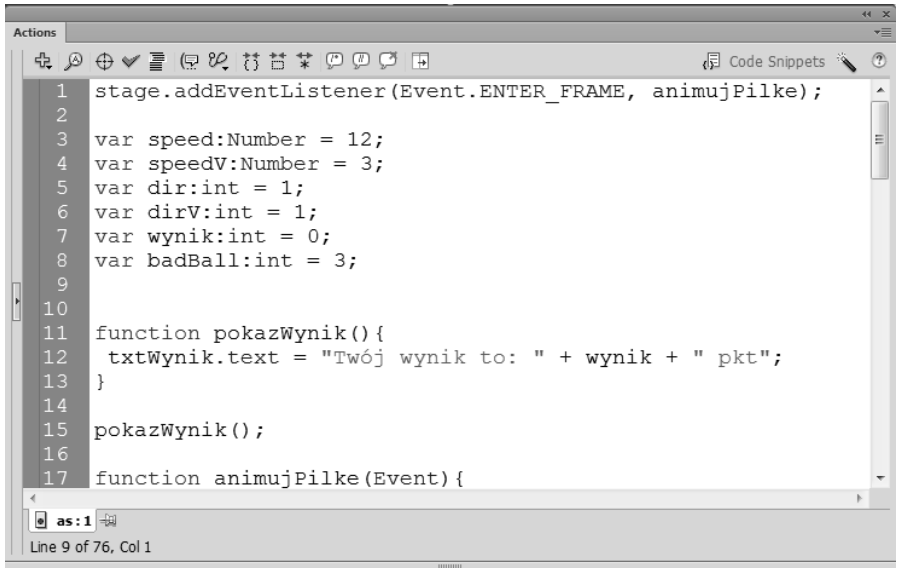
Kolejny fragment kodu odpowiedzialnego za zliczanie punktów wprowadzamy wewnątrz funkcji `animujRakietke()`. Każde odbicie piłki powinno powiększać (inkrementować) wartość zmiennej o 1 i automatycznie wyświetlać odpowiednie dane na ekranie. Lokalizujemy więc fragment z metodą `hitTestObject()` i wprowadzamy niewielkie modyfikacje. W chwili odbicia piłeczki rakietką zapis `wynik++` powiększa zmienną `wynik` o 1. Wywołanie funkcji `pokazWynik()` pozwala na prezentację uaktualnionej punktacji w przygotowanym polu tekstowym (rysunek 8.68).



Rysunek 8.68. *Wywołanie funkcji pokazWynik() pozwala na prezentację uaktualnionej punktacji w przygotowanym polu tekstowym*

```
function animujRakietke(Event){
  if(mcBall.hitTestObject(mcRakietka)){
     dir=1;
     wynik++
     pokazWynik();
     }
  if(mcRakietka.y < 0){
    mcRakietka.y = 0;
  }
  if(mcRakietka.y >stage.stageHeight){
    mcRakietka.y = stage.stageHeight;
  }
    mcRakietka.y +=rakietkaSpeed;
}
```

Ostatni krok to reakcja programu na przekroczenie dopuszczalnej liczby nieodbitych piłek. Aby sprawdzić ten stan, ponownie wykorzystamy dodatkową zmienną — `badBall`. Przechodzimy więc do kodu i w pobliżu pozostałych zmiennych wprowadzamy deklarację kolejnej (rysunek 8.69).



Rysunek 8.69. *Ostatni krok to reakcja programu na przekroczenie dopuszczalnej liczby nieodbitych piłek. Aby sprawdzić ten stan, ponownie wykorzystamy dodatkową zmienną — „badBall"*

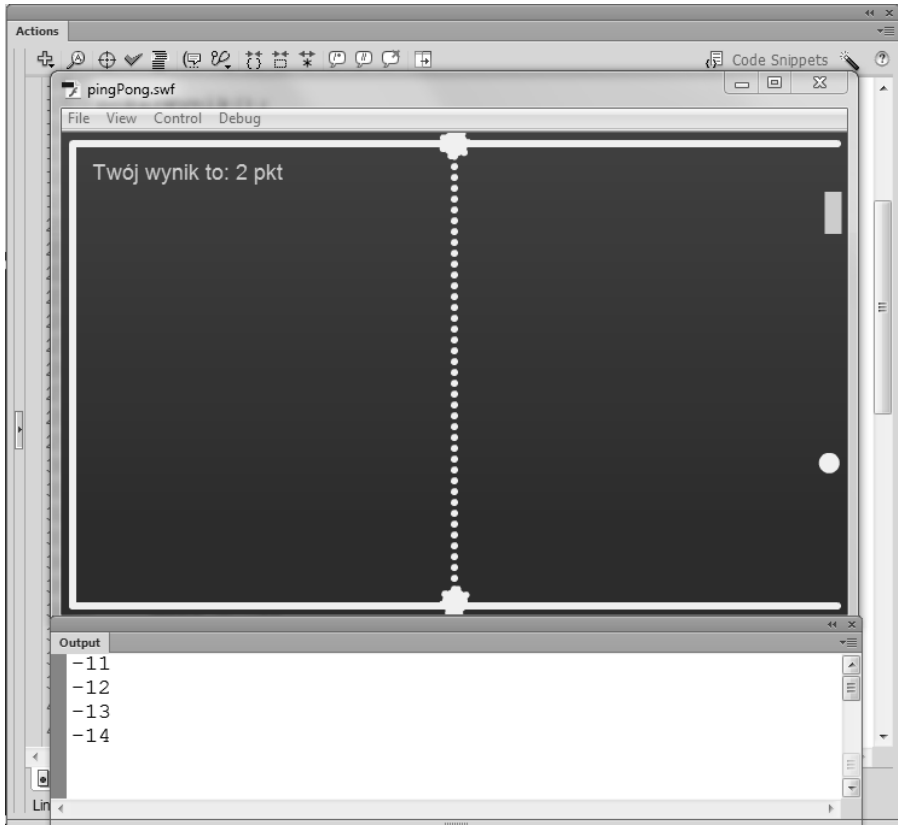
```
var speed:Number = 12;
var speedV:Number = 3;
var dir:int = 1;
var dirV:int = 1;
var wynik:int = 0;
var badBall:int = 3;
```

Aby modyfikować wartość zmiennej `badBall` w chwili, gdy piłeczka wyleci nieodbita poza obszar sceny, musimy dopisać fragment kodu wewnątrz przygotowanej wcześniej instrukcji warunkowej. Lokalizujemy funkcję `animujPilke()` i szukamy instrukcji warunkowej `if`. W jej wnętrzu dodajemy niewielki fragment kodu `baddBall--;`. Aby podejrzeć poprawność naszych działań, możemy łatwo śledzić zachowanie zmiennej `badBall`, wykorzystując metodę `trace(badBall)`.

```
function animujPilke(Event){
mcBall.x -=speed * dir;
mcBall.y +=speed * dirV;

if(mcBall.x <mcBall.width/2 ){
dir = -1;
speed *=1.03
}
if (mcBall.x > stage.stageWidth){
mcBall.x = -20;
dir=1;
badBall--;
trace(badBall);
}
if(mcBall.y < mcBall.width/2){
dirV = 1;
speedV *=1.1
}
```

```
    if(mcBall.y > stage.stageHeight - mcBall.height/2){
    dirV = -1;
    speedV *=1.3
    }
}
```

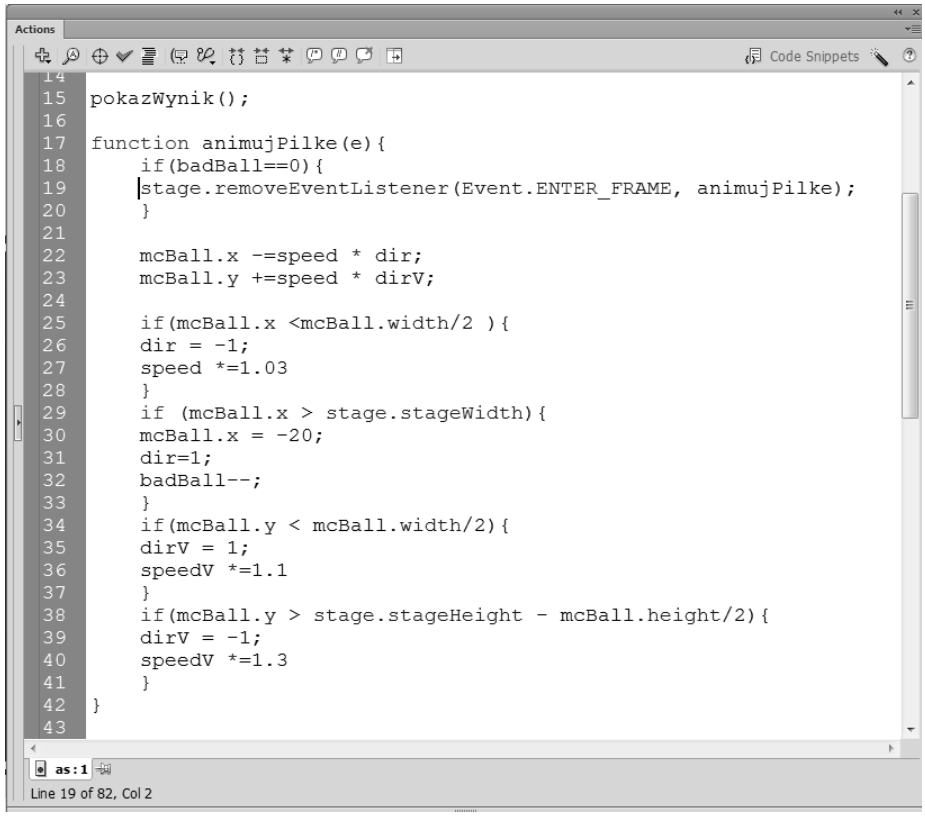
W chwili gdy zmienna osiągnie wartość zero, nie dzieje się nic szczególnego, ba, program pozwala kontynuować grę nawet wtedy, gdy zmienna przyjmuje wartości ujemne (rysunek 8.70). To właśnie należy zmienić. W chwili gdy użytkownik trzykrotnie straci swą piłkę, gra powinna się zatrzymać, a na ekranie powinien pojawić się stosowny komunikat. Dodatkowo, powinna istnieć możliwość ponownego uruchomienia naszej gry zupełnie od początku. Będą to już ostatnie zmiany, które wprowadzimy w naszym kodzie.



Rysunek 8.70. *W chwili gdy zmienna osiągnie wartość zero, nie dzieje się nic szczególnego, ba, program pozwala kontynuować grę nawet wtedy, gdy zmienna przyjmuje wartości ujemne*

W chwili gdy zmienna `badBall` osiągnie wartość `0`, nasza piłka powinna się zatrzymać. Wykonamy to, zdejmując nasłuchiwanie zdarzenia `ENTER_FRAME` odpowiedzialnego za animację piłeczki. Niestety krok ten wymaga dodatkowej zmiany samej funkcji. O co tu chodzi?

Rozpocznijmy od wprowadzenia stosownych zmian, a później przeanalizujemy nasz kod. Zmiany dotyczą w tej chwili jedynie funkcji `animujPilke()` i jej pierwszych czterech wierszy (rysunek 8.71).



Rysunek 8.71. *Rozpocznijmy od wprowadzenia stosownych zmian, a później przeanalizujemy nasz kod. Zmiany dotyczą w tej chwili jedynie funkcji animujPilke() i jej pierwszych czterech wierszy*

```
function animujPilke(e){
  if(badBall==0){
    stage.removeEventListener(Event.ENTER_FRAME, animujPilke);
  }

  mcBall.x -=speed * dir;
  mcBall.y +=speed * dirV;

  if(mcBall.x <mcBall.width/2 ){
  dir = -1;
  speed *=1.03
  }
  if (mcBall.x > stage.stageWidth){
  mcBall.x = -20;
  dir=1;
  badBall--;
  }
  if(mcBall.y < mcBall.width/2){
```

```
    dirV = 1;
    speedV *=1.1
    }
    if(mcBall.y > stage.stageHeight - mcBall.height/2){
    dirV = -1;
    speedV *=1.3
    }
  }
```

Najbardziej widoczna zmiana dotyczy wprowadzenia prostej instrukcji `if`. Jeżeli zmienna `badBall` osiągnie wartość `0`, wówczas zdejmujemy możliwość nasłuchiwania zdarzenia `ENTER_FRAME`, które jest odpowiedzialne za animację piłki. To podstawowa zmiana, jaką tu wprowadziłem. Kolejna dotyczy samej definicji funkcji. W miejsce zapisu `function animujPilke(Event)` pojawił się teraz fragment `function animujPilke(e)`. Jest to konieczna modyfikacja kodu, niezbędna do tego, aby udało się zdjąć nasłuchiwanie zdarzenia `ENTER_FRAME`.

Cały kod naszej gry w tej chwili przedstawia się następująco:

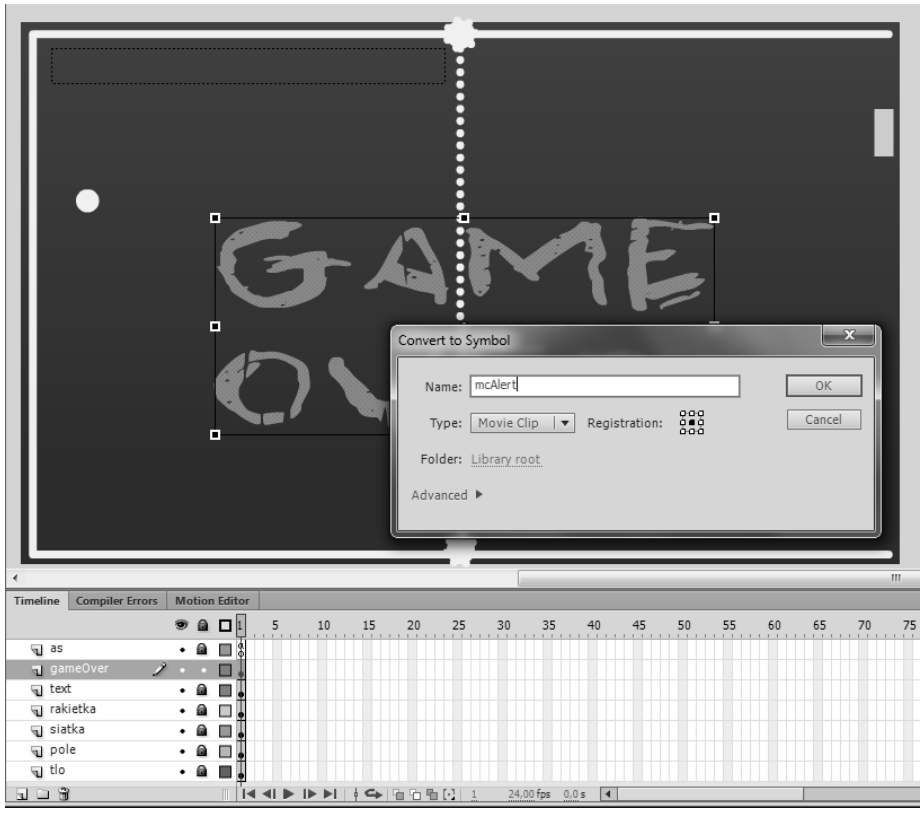
```
stage.addEventListener(Event.ENTER_FRAME, animujPilke);

var speed:Number = 12;
var speedV:Number = 3;
var dir:int = 1;
var dirV:int = 1;
var wynik:int = 0;
var badBall:uint = 3;

function pokazWynik(){
  txtWynik.text = "Twój wynik to: " + wynik + " pkt";
}
pokazWynik();

function animujPilke(e){
  if(badBall==0){
    stage.removeEventListener(Event.ENTER_FRAME, animujPilke);
  }

  mcBall.x -=speed * dir;
  mcBall.y +=speed * dirV;

  if(mcBall.x <mcBall.width/2 ){
  dir = -1;
  speed *=1.03
  }
  if (mcBall.x > stage.stageWidth){
  mcBall.x = -20;
  dir=1;
  badBall--;
  }
  if(mcBall.y < mcBall.width/2){
  dirV = 1;
  speedV *=1.1
  }
  if(mcBall.y > stage.stageHeight - mcBall.height/2){
  dirV = -1;
  speedV *=1.3
```

```
  }
}

var jestWcisniety:Boolean = false;
stage.addEventListener(KeyboardEvent.KEY_DOWN, klawiszWDol);
stage.addEventListener(KeyboardEvent.KEY_UP, klawiszWGore);

var rakietkaSpeed:int = 0;

function klawiszWDol(evt:KeyboardEvent){
  jestWcisniety = true;
    switch(evt.keyCode){
    case Keyboard.UP: rakietkaSpeed = -20;
    break;
    case Keyboard.DOWN: rakietkaSpeed = 20;
    break;
  }
}
function klawiszWGore(KeyboardEvent){
  jestWcisniety = false;
  rakietkaSpeed = 0;
}

mcRakietka.addEventListener(Event.ENTER_FRAME, animujRakietke);

function animujRakietke(Event){
  if(mcBall.hitTestObject(mcRakietka)){
     dir=1;
     wynik++
     pokazWynik();
     }
  if(mcRakietka.y < 0){
    mcRakietka.y = 0;
  }
  if(mcRakietka.y >stage.stageHeight){
    mcRakietka.y = stage.stageHeight;
  }
    mcRakietka.y +=rakietkaSpeed;
}
```

Brakuje jeszcze komunikatu o zakończeniu gry i możliwości rozpoczęcia jej od początku. Rozpoczynamy od komunikatu. Aby upewnić się, że nie zepsujemy żadnych istniejących elementów, blokujemy pozostałe warstwy. Dodajemy w tym celu nową warstwę i rozpoczynamy dalszą pracę.

Zakończenie gry

Korzystając z dowolnych narzędzi graficznych lub statycznego tekstu, rysujemy symbol typu `MovieClip`, który zawiera informację o zakończeniu gry. Możemy wykorzystać tu przykładowo znany i popularny z dawnych czasów slogan typu *Game Over* pisany za pomocą pędzla. Nie jest to jednak najważniejsze. Ważne jest, aby przygotowany moduł przekształcić w symbol `MovieClip` i ustawić w odpowiednim miejscu na scenie. Aby możliwe było użycie przygotowanego klipa, nadajemy mu nazwę instancji `mcAlert` (rysunek 8.72).

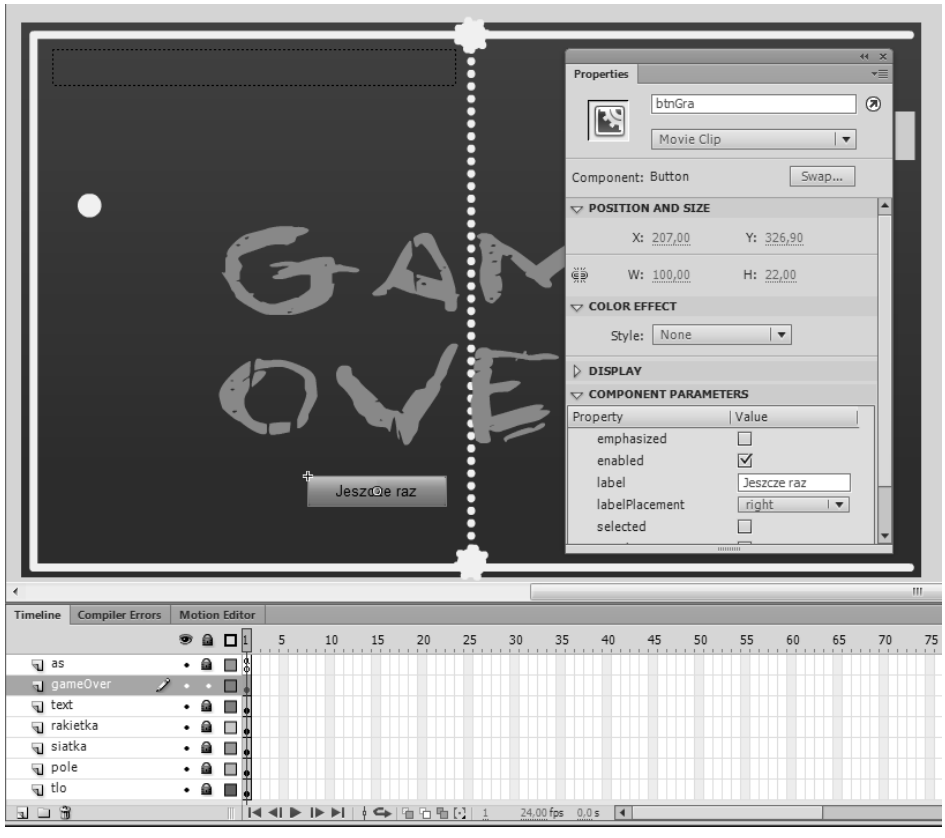


Rysunek 8.72. *Korzystając z dowolnych narzędzi graficznych lub statycznego tekstu, rysujemy symbol typu MovieClip, który zawiera informację o zakończeniu gry. Możemy wykorzystać tu przykładowo znany i popularny z dawnych czasów slogan typu „Game Over" pisany za pomocą pędzla*

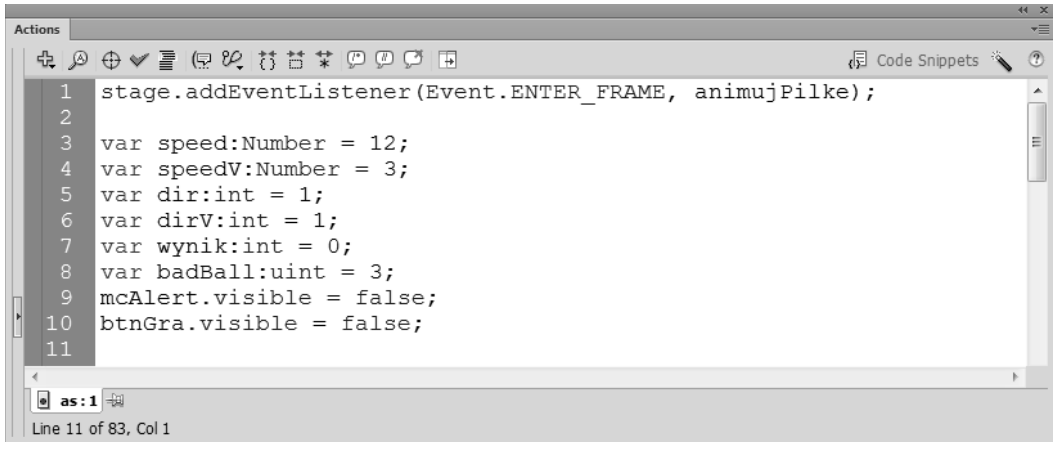
Zupełnie podobnie, korzystając z dowolnych elementów graficznych, budujemy symbol typu `Button` z etykietą *Zagraj ponownie* i nadajemy mu nazwę instancji `btnGra` (rysunek 8.73).

W ten sposób na scenie pojawiły się dwa dodatkowe elementy, które nie powinny być widoczne w chwili uruchomienia naszej gry. Zmienimy to z łatwością, korzystając w tym celu z kodu ActionScript i właściwości `visible`. Na początku naszego kodu wprowadzamy dwie instrukcje (rysunek 8.74).

```
var speed:Number = 12;
var speedV:Number = 3;
var dir:int = 1;
var dirV:int = 1;
var wynik:int = 0;
var badBall:uint = 3;
mcAlert.visible = false;
btnGra.visible = false;
```

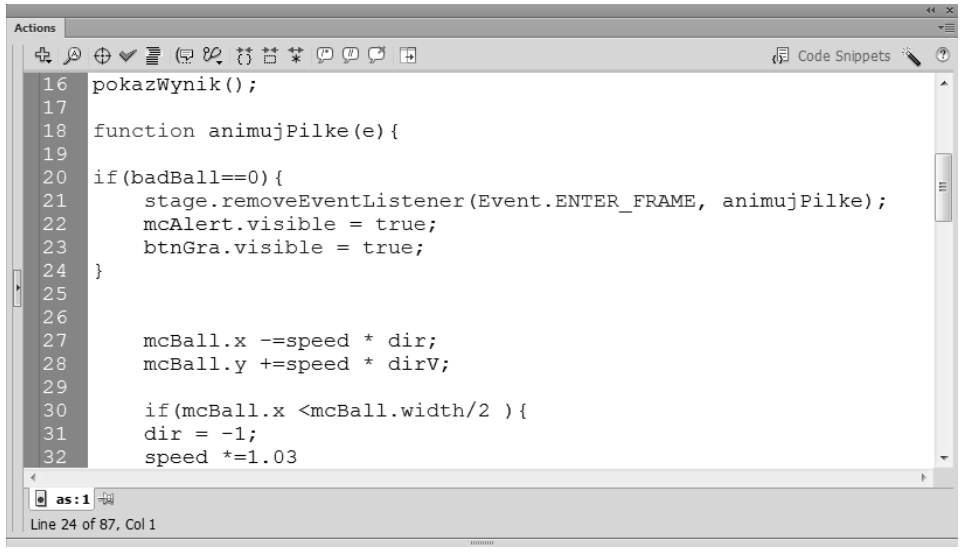


Rysunek 8.73. *Zupełnie podobnie, korzystając z dowolnych elementów graficznych, budujemy symbol typu Button z etykietą „Zagraj ponownie" i nadajemy mu nazwę instancji „btnGra"*



Rysunek 8.74. *W ten sposób na scenie pojawiły się dwa dodatkowe elementy, które nie powinny być widoczne w chwili uruchomienia naszej gry*

Mimo że w trybie roboczym oba obiekty znajdują się na scenie, po uruchomieniu gry nie są już widoczne. Powinny pojawić się jednak tuż po zakończeniu gry. Lokalizujemy więc pierwszą instrukcję warunkową wewnątrz funkcji `animujPilke()` i dodajemy dwie nowe linie kodu (rysunek 8.75).



Rysunek 8.75. *Mimo że w trybie roboczym oba obiekty znajdują się na scenie, po uruchomieniu gry nie są już widoczne. Powinny pojawić się jednak tuż po zakończeniu gry. Lokalizujemy więc pierwszą instrukcję warunkową wewnątrz funkcji animujPilke() i dodajemy dwie nowe linie kodu*

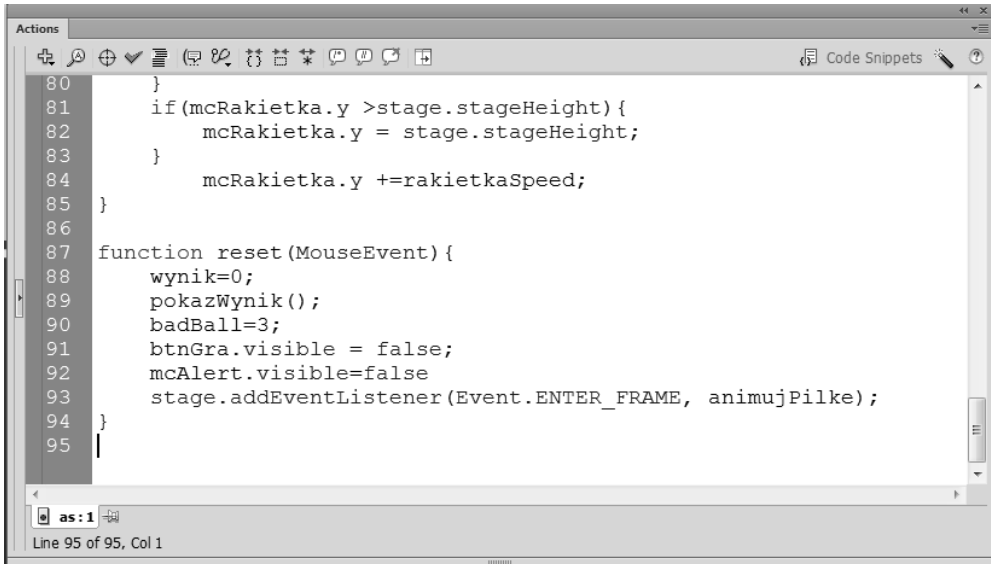
```
if(badBall==0){
    stage.removeEventListener(Event.ENTER_FRAME, animujPilke);
    mcAlert.visible = true;
    btnGra.visible = true;
  }
```

W ten sposób po zakończeniu gry oba elementy stają się widoczne (rysunek 8.76).

Ostatni krok to powrót do początku gry w chwili wciśnięcia przycisku `btnStart`. Działanie to wymaga kilku istotnych modyfikacji. W chwili wciśnięcia przycisku musimy wyzerować wynik naszej gry, wyświetlić stosowny wynik na scenie, przywrócić możliwość użycia trzech kolejnych piłek i oczywiście ukryć ponownie wyświetlone właśnie elementy i uruchomić animację piłeczki. Przygotujemy w tym celu specjalną funkcję. Nazwiemy ją `reset();`. Funkcja ta powinna być wprowadzona tuż poniżej wywołanej wcześniej metody `pokazWynik()` (rysunek 8.77).

```
function reset(MouseEvent){
  wynik=0;
  pokazWynik();
  badBall=3;
  btnGra.visible = false;
  mcAlert.visible=false
  stage.addEventListener(Event.ENTER_FRAME, animujPilke);
}
```

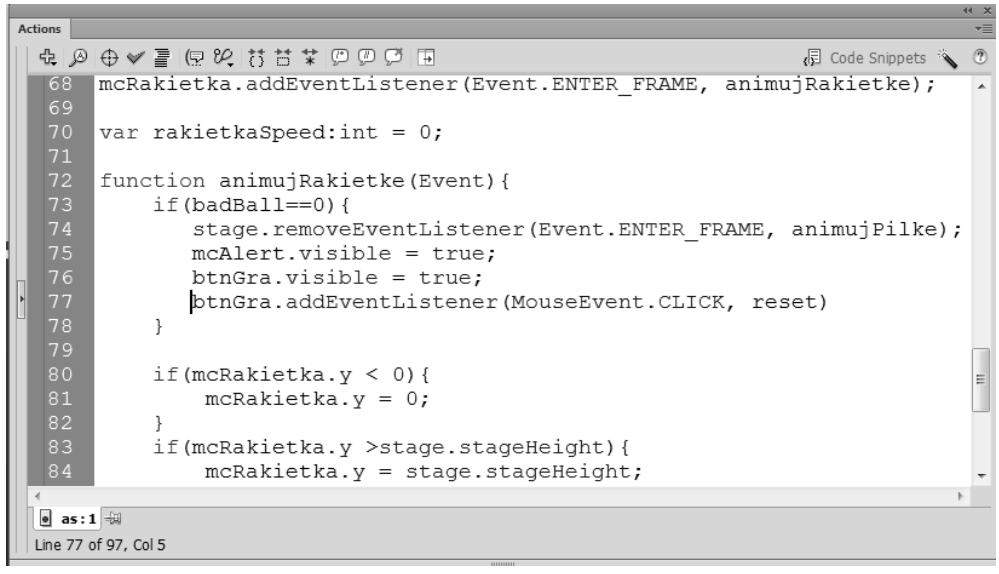
Rysunek 8.76. *W ten sposób po zakończeniu gry oba elementy stają się widoczne*



Rysunek 8.77. *Ostatni krok to powrót do początku gry w chwili wciśnięcia przycisku „btnStart". Działanie to wymaga kilku istotnych modyfikacji Przygotujemy w tym celu specjalną funkcję. Nazwiemy ją reset();. Funkcja ta powinna być wprowadzona tuż poniżej wywołanej wcześniej metody pokazWynik()*

Aby jednak możliwe było wykorzystanie przygotowanej właśnie funkcji, musimy do przycisku `btnGra` dodać możliwość nasłuchiwania zdarzenia `CLICK` i wywołać naszą funkcję `reset()`.

Przechodzimy więc do wnętrza instrukcji warunkowej wewnątrz funkcji `animuj ↪Pilke()` i wprowadzamy dodatkowy (ostatni) fragment (rysunek 8.78).



Rysunek 8.78. *Przechodzimy więc do wnętrza instrukcji warunkowej wewnątrz funkcji animujPilke() i wprowadzamy dodatkowy (ostatni) fragment*

```
if(badBall==0){
    stage.removeEventListener(Event.ENTER_FRAME, animujPilke);
    mcAlert.visible = true;
    btnGra.visible = true;
    btnGra.addEventListener(MouseEvent.CLICK, reset)
  }
```

W ten sposób po utracie trzeciej kolejnej piłki program usuwa nasłuchiwanie zdarzenia `ENTER_FRAME` odpowiedzialnego za animację piłeczki, wyświetla komunikat o zakończeniu gry i przycisk do ponownego jej uruchomienia. Ostatnie działanie to dodanie reakcji na zdarzenie `CLICK` dla wyświetlonego batona. Po kliknięciu wywołuje on funkcję `reset()`, a ta zeruje wynik i wyświetla go na ekranie, następnie resetuje zmienną `badBall`, ukrywa komunikat o zakończeniu gry i przycisk oraz ponownie uruchamia animację piłeczki. Gra rozpoczyna się od początku.

Skorowidz

A

B

C

D

E

F

G

H

I

J

K

L

Ł

M

N

0

P

R

S

Ś

T

U

W

X

Z

Baner?
Animacja?
Gra?
Tylko we Flashu!

Bez technologii Flash trudno byłoby wyobrazić sobie współczesny internet. Banery, animacje, gry, interaktywne prezentacje, a nierzadko i całe skomplikowane serwisy WWW — wszystko to powstaje przy jej użyciu. Z czasem (oraz zwiększaniem się potrzeb twórców i użytkowników) rosły też możliwości środowiska Flash. Jego najnowsza wersja oferuje bardzo rozbudowane narzędzia, dzięki którym opracowanie ciekawych i przykuwających oko projektów staje się naprawdę proste. Oczywiście tylko wtedy, gdy się wie, gdzie znaleźć i jak zastosować odpowiednie funkcje.

Nieocenioną pomocą w nauce posługiwania się środowiskiem okaże się książka *Adobe Flash CS6 i ActionScript 3.0. Interaktywne projekty od podstaw*. Początkujący użytkownicy poznają dzięki niej najnowszą edycję programu Flash, metody używania narzędzi odpowiednich do konkretnych celów, zasady tworzenia różnego rodzaju materiałów, mechanizmy umożliwiające interakcje z użytkownikiem oraz podstawy programowania w języku ActionScript 3.0 i sposoby wykorzystywania go w swoich projektach. Przedstawione w książce informacje są poparte przykładami, co pomaga nie tylko łatwo utrwalić zdobytą wiedzę, lecz również nauczyć się praktycznego stosowania poznanych technik.

- Opis środowiska Flash i jego elementów
- Tworzenie animowanych banerów
- Zapewnianie interakcji z użytkownikiem
- Tworzenie galerii i panoram
- Zastosowanie języka ActionScript w animacjach
- Tworzenie prostych gier i zabawek
- Korzystanie z danych XML
- Budowa prostych aplikacji

Cena: 99,00 zł
ISBN 978-83-246-3865-9
9 788324 638659

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