Macromedia Flash Super Samurai

Autor: praca zbiorowa
Tłumaczenie: Marek Binkowski
ISBN: 83-7197-696-8
Tytuł oryginału: Macromedia Flash Super Samurai
Format: B5, stron: 378
Zawiera CD-ROM

Głównym założeniem tego podręcznika jest nauka poprzez zabawę (bo kreskówki to przede wszystkim duża dawka zabawy). Dlatego wydanie tej książki w formie zwykłego podręcznika byłoby poważnym błędem. Podstawy Flasha można przyswoić sobie z różnych źródeł, więc nie będziemy im tu poświęcać zbytniej uwagi. W związku z tym najlepiej by było, abyś podstawy miał już opanowane, gdyż nie będziemy zajmować się opisem każdego narzędzia czy sposobu jego zastosowania. Czasami najlepiej jest zapoznać się z narzędziem poprzez użycie go w niestandardowy sposób.
W każdym rozdziale znajdziesz ćwiczenia, które umożliwią ci zastosowanie wyłożonych przez nas teorii w praktyce. Do wszystkich ćwiczeń dołączono kompletne pliki w formacie FLA zawierające wszystkie potrzebne elementy. Na płycie CD znajduje się ponad 75 autorskich plików wykonanych we Flashu. Dodatkową atrakcją jest (umieszczony również na płycie CD) plik w formacie FLA z dwuminutową kreskówką wideo z muzyką, Human Blues wykonaną przez Webera.

Jeśli twoim marzeniem jest programowanie gier w piątej wersji Flasha, lecz zniechęca cię złożoność procesu tworzenia, ta książka może przyczynić się do twojego sukcesu. Poprzez wykonywanie ćwiczeń i analizowanie przykładów zrozumiesz nawet najbardziej skomplikowane zagadnienia. Jeśli tylko przyswoisz sobie techniki zastosowane w grach dostarczonych z tą książką na pewno będziesz tworzył własne gry we Flashu.
komunikacji z przeglądarką za pomocą Flasha i skryptów JavaScript.

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

Spis treści

Rozdział 2.
Flash i grafika trójwymiarowa

Michael Brandon Williams
Torben Nielsen

Coraz większa liczba projektantów pracujących we Flashu interesuje się grafiką trójwymiarową. Trójwymiarowe efekty umieszczone w filmach Flasha pozwalają wzbogacić system nawigacyjny, wywrzeć dobre wrażenie na kliencie lub po prostu zabawić widza. Rzeczywiście, w ciągu kilku ostatnich lat grafika trójwymiarowa w filmach Flasha stała się bardzo popularna. Trend ten zapoczątkowały takie witryny jak *www.yugop.com* czy *www.mano1.com* i od tego czasu zainteresowanie tą dziedziną ciągle rośnie.

Jednak Flash jest programem do tworzenia grafiki dwuwymiarowej — nie obsługuje modeli trójwymiarowych. Jak zatem można tworzyć we Flashu animacje trójwymiarowe? To łatwe — trzeba udawać.

Istnieją dwie podstawowe metody tworzenia iluzji trójwymiarowego kształtu i ruchu — za pomocą skryptów ActionScript, gdzie cały projekt jest generowany przez kod; lub przy użyciu wyrenderowanych obiektów trójwymiarowych, przygotowanych w innych programach, a następnie zaimportowanych do Flasha i sterowanych prostymi akcjami ActionScript. W tym rozdziale omówimy oba podejścia.

ActionScript i trzy wymiary

W pierwszej części tego rozdziału skupimy się na zagadnieniach związanych z tworzeniem efektów trójwymiarowych za pomocą skryptów. Wszystkie równania wyprowadzimy w tekście, starając się jednak unikać matematycznej komplikacji tematu. Po omówieniu teoretycznych zagadnień przyjrzymy się kilku demonstracjom, w których pokażemy praktyczne zastosowanie tych zagadnień.

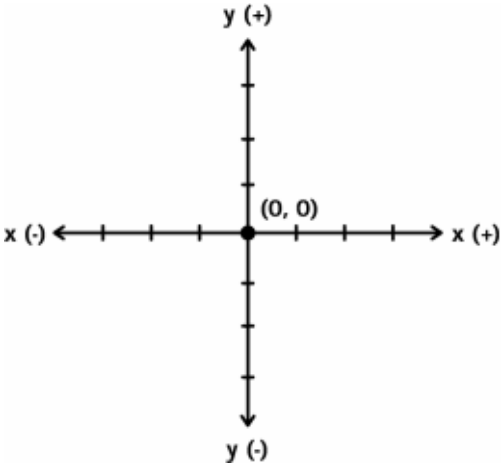
Trzy wymiary

Rozpocznijmy od przypomnienia pewnych zagadnień, które mogą być ci już znane. Pokażę, jak punkty są reprezentowane w przestrzeni trójwymiarowej, jak rzutować je na płaszczyznę dwuwymiarową i jak to podejście działa we Flashu.

Matematyczny układ współrzędnych

Chyba każdy zna dwuwymiarowy układ współrzędnych kartezjańskich. Jego środek znajduje się w punkcie umieszczonym zwykle na środku wykresu, w miejscu gdzie przecinają się zerowe osie *x* i *y*, prostopadłe do siebie. Oś *x* jest pozioma i przebiega od lewej strony (ujemna półoś) do prawej (półoś dodatnia). Oś *y* jest pionowa i przebiega od dołu (ujemna półoś) do góry (półoś dodatnia).

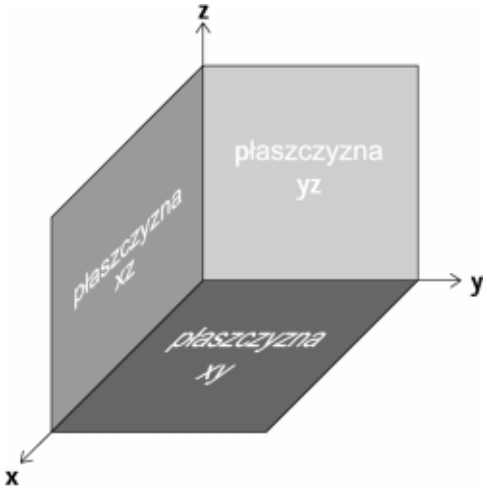
Rysunek 2.1 przedstawia taki kartezjański układ współrzędnych.



Rysunek 2.1. *Dwuwymiarowy kartezjański układ współrzędnych*

Aby określić położenie punktu w przestrzeni trójwymiarowej, definiujemy trzy prostopadłe osie współrzędnych oraz środek układu współrzędnych (0, 0, 0) jako punkt przecięcia tych osi. Są to osie *x*, *y* i *z*. Każda para tych osi może stanowić płaszczyznowy układ współrzędnych.

Rysunek 2.2 przedstawia trójwymiarowy układ współrzędnych.



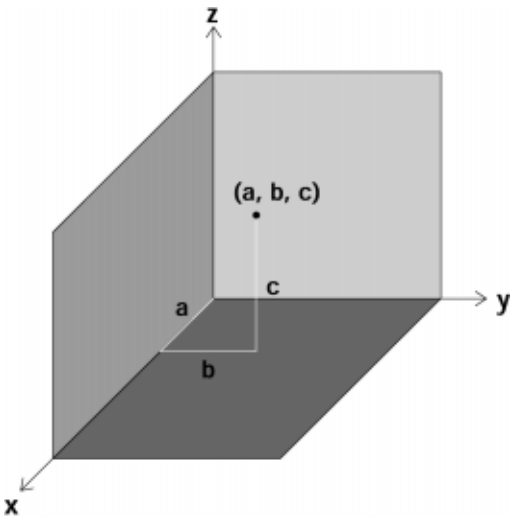
Rysunek 2.2. *Trójwymiarowy układ współrzędnych*

Uporządkowane trójki liczb

Studenci geometrii analitycznej i algebry przez cały czas rysują wykresy. Położenie każdego punktu na wykresie dwuwymiarowym jest określone przez uporządkowaną parę liczb. Pierwsza liczba odpowiada współrzędnej punktu na osi *x* (tak zwana *odcięta*), zaś druga współrzędnej na osi *y* (*rzędna*).

Aby określić położenie punktu w przestrzeni trójwymiarowej, stosujemy uporządkowaną trójkę liczb. Są to trzy liczby, odpowiadające kolejno współrzędnym punktu na osi *x*, *y* i *z*. Stosując trójwymiarowy układ współrzędnych, możemy narysować punkt w przestrzeni o ogólnych współrzędnych (*x*, *y*, *z*) (rysunek 2.3).

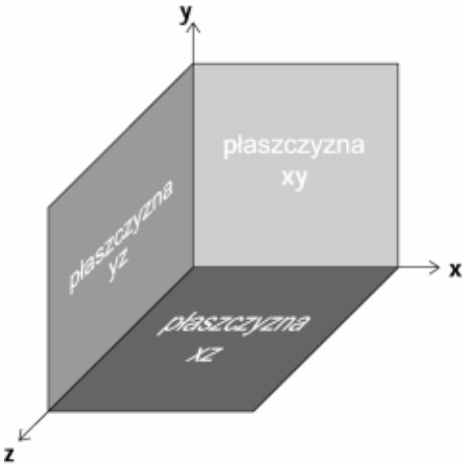
Rysunek 2.3. *Uporządkowana trójka liczb określa położenie punktu w przestrzeni trójwymiarowej*



Uporządkowana trójka liczb rzeczywistych (*a*, *b*, *c*) mówi, byśmy się przesunęli się o *a* jednostek wzdłuż osi *x*, następnie o *b* jednostek wzdłuż osi *y* i wreszcie o *c* jednostek wzdłuż osi *z*, i tam narysowali punkt.

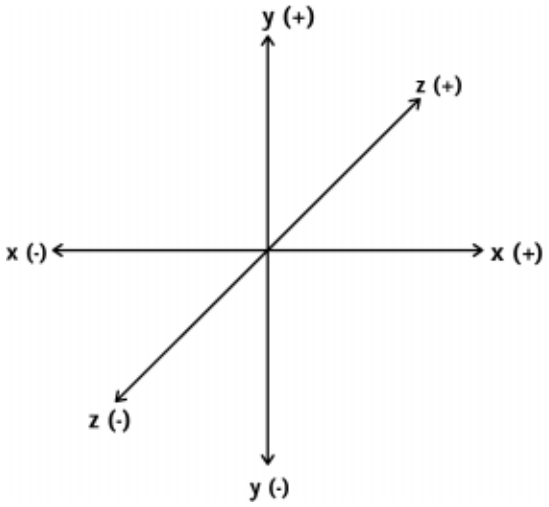
Trójwymiarowy układ współrzędnych we Flashu

Pokazany powyżej trójwymiarowy układ współrzędnych jest standardem w niemal wszystkich dziedzinach matematyki. Zapewne zauważyłeś pewne różnice względem dwuwymiarowego układu kartezjańskiego. Spójrz jeszcze raz na rysunek 2.3. Oś *x* „wychodzi" z płaszczyzny kartki, oś *y* przebiega od lewej do prawej, zaś oś *z* jest prostopadła do obu pozostałych i przebiega od dołu do góry. Innymi słowy, osie *x* i *y* są zamienione miejscami. W naszych obliczeniach taka zamiana wprowadziłaby niepotrzebne komplikacje, dlatego we Flashu będziemy korzystać z innego układu współrzędnych. Oś *x* będzie przebiegać w poziomie, jak w układzie dwuwymiarowym; podobnie oś *y* będzie przebiegać w pionie. Natomiast oś *z* będzie prostopadła do płaszczyzny ekranu (rysunek 2.4).



Rysunek 2.4. *Układ osi współrzędnych w trójwymiarowym układzie, jaki będziemy stosować we Flashu*

We Flashu oś *z* reprezentuje odległość obiektu od płaszczyzny ekranu (w sensie głębokości). Gdy wartość współrzędnej *z* rośnie, oznacza to, że obiekt oddala się od ekranu, w głąb obrazu; gdy wartość współrzędnej *z* maleje, obiekt przybliża się do oczu widza (rysunek 2.5)[1].



Rysunek 2.5. *Orientacja osi w trójwymiarowym układzie współrzędnych stosowanym we Flashu*

Symulacja trzech wymiarów

Po określeniu położenia punktów w przestrzeni chcemy je przedstawić na ekranie. Jednak jak narysować trójwymiarowy punkt na dwuwymiarowym ekranie i jak dobrać rozmiary trójwymiarowych obiektów, aby na płaszczyźnie ekranu wydawały się trójwymiarowe? W tym celu wyprowadzimy kilka prostych wzorów.

[1] W rzeczywistości zwrot osi y w układzie współrzędnych Flasha jest odwrotny, jednak różnicę tę z łatwością skorygujemy w prezentowanych przykładach — *przyp. tłum.*

Perspektywa

Perspektywa to właśnie to, co pozwala nam odróżnić przestrzeń trójwymiarową od płaszczyzny dwuwymiarowej. Jest to wszechobowiązujące pozorne „zakrzywienie" świata, które rozmieszcza obiekty na scenie w zależności od ich odległości od obserwatora. Gdy stoisz na środku prostej drogi, patrząc w najdalszy jej punkt, widzisz jak jej krawędzie stopniowo zbliżają się do siebie, w miarę wzrostu odległości. Gdy spojrzysz odpowiednio daleko, zobaczysz, że zbiegają się one w jedną linię. Perspektywa wpływa też na sposób postrzegania ruchu. Na przykład, gdy obiekt na pierwszym planie porusza się z określoną prędkością, a na dalszym planie widać drugi obiekt poruszający się z taką samą prędkością, pozorna (postrzegana) prędkość ruchu obiektu na pierwszym planie jest większa.

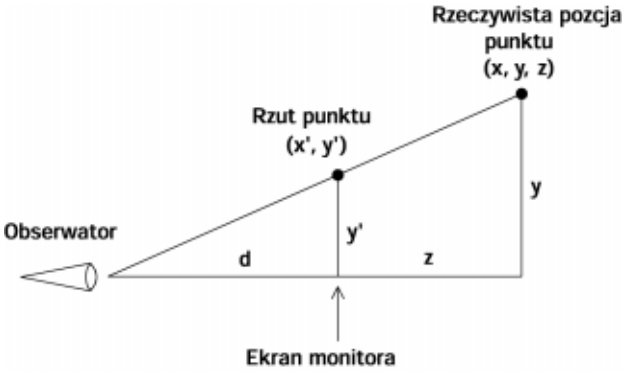
Zamiana uporządkowanych trójek liczb na uporządkowane pary

Problem perspektywy można uprościć do płaszczyzny. Położenie punktu w przestrzeni jest określane przez uporządkowaną trójkę liczb (x, y, z). Naszym zamiarem jest wyeliminowanie głębi, czyli współrzędnej z, ponieważ ekran monitora nie posiada głębi, tylko szerokość i wysokość. W jakiś sposób musimy zrzutować na płaszczyznę monitora wszystkie obiekty w prezentowanej przestrzeni. Położenie każdego z nich wyrazimy za pomocą uporządkowanej pary liczb, obliczając ich wartości z uwzględnieniem pierwotnej pozycji na osi z. Najprostszym rozwiązaniem jest zwykłe odrzucenie informacji z, lecz w tym przypadku zatracisz efekt perspektywy, czyli krawędzie drogi nie będą się zbiegały, lecz pozostaną równoległe.

Problem ten rozwiążemy z użyciem prostej geometrii. Zanim jednak omówimy rzutowanie trójwymiarowego punktu na dwuwymiarowy ekran monitora, przyjmijmy pewne oznaczenia, którymi będziemy się posługiwać.

Rysunek 2.6 przedstawia punkt w przestrzeni, posiadający ogólne współrzędne (x, y, z). Po zrzutowaniu tego punktu na ekran monitora, jego obraz ma współrzędne (x', y'). Oznaczamy je znakiem apostrofu[2], by odróżnić współrzędne x i y w reprezentowanej przestrzeni od współrzędnych x' i y' na płaszczyźnie monitora. Ponieważ na rysunku 2.6 mogę przedstawić tylko przypadek dwuwymiarowy, muszę zrezygnować z jednego z wymiarów — rezygnuję zatem ze współrzędnej x (rysunek ten przedstawia obserwatora, ekran monitora i reprezentowaną przestrzeń obserwowaną „z boku"). Wielkość oznaczona literą d jest odległością oka obserwatora od ekranu komputera. Wartość ta odegra kluczową rolę w naszych obliczeniach dotyczących wyglądu środowiska trójwymiarowego. Współrzędna z punktu w przestrzeni jest wyznaczana względem płaszczyzny ekranu monitora, czyli płaszczyzna ekranu ma współrzędną z równą zero (tam znajduje się początek trójwymiarowego układu współrzędnych oraz zerowe osie x i y). Dodajmy też, że współrzędna z obserwatora ma wartość ujemną, ponieważ oś z jest skierowana „w głąb" monitora.

[2] Tak oznaczone symbole czytaj „x prim" lub „y prim" — *przyp. tłum.*



Rysunek 2.6. *Punkt trójwymiarowej przestrzeni rzutujemy na dwuwymiarowy ekran monitora, by wyświetlić go we Flashu*

Posługując się tym rysunkiem, rozwiążmy problem perspektywy. Wykorzystamy w tym celu znane nam wielkości *d*, *y* i *z*, by znaleźć poszukiwaną wartość *y'*. Na rysunku widać dwa trójkąty podobne. Zgodnie z prostymi prawami geometrii, możemy porównać boki obu trójkątów, stosując proporcję jak na rysunku 2.7.

Rysunek 2.7. *Boki trójkątów podobnych są proporcjonalne*

$$\frac{d}{y'} = \frac{d+z}{y}$$

Stosując elementarne przekształcenia algebraiczne, przekształćmy powyższe równanie, by znaleźć wartość *y'*. Mnożymy obie strony równania przez *y* i przez *y'*, a następnie dzielimy przez sumę *(d+z)*. Rysunek 2.8 przedstawia te przekształcenia i ich rezultat.

Rysunek 2.8. *Korzystając z proporcji współrzędnej y w uporządkowanej trójce liczb, przekształcamy ją w uporządkowaną parę*

$$\frac{d}{y'} = \frac{d+z}{y}$$

$$\Downarrow$$

$$d \cdot y = y' (d + z)$$

$$\Downarrow$$

$$y' = \frac{d \cdot y}{d + z}$$

Tym sposobem wyprowadziliśmy równanie, które ma kluczowe znaczenie w symulacji perspektywy.

Równanie to pozwala obliczyć współrzędną *y'* rzutu punktu na ekran monitora. W podobny sposób możemy wyprowadzić równanie służące do obliczenia współrzędnej *x'* tego rzutu. Zapiszmy oba równania, służące do przeliczania uporządkowanej trójki liczb na uporządkowaną parę z uwzględnieniem współrzędnej *z* (rysunek 2.9).

Pracując z równaniami w matematyce, dążymy do przedstawienia ich w jak najprostszej postaci. Na przykład, gdy mamy wyrażenie postaci x^2+x, możemy je uprościć do postaci *x(x+1)*, wyciągając *x* przed nawias. Podobnie w naszych równaniach perspektywy możemy wyciągnąć ułamek *d/(d+z)* i przyjąć, że jest to odrębna zmienna, obliczana przed

Rysunek 2.9. *Równania rzutujące punkt przestrzeni na płaszczyznę monitora*

$$x' = \frac{x \cdot d}{z + d}$$

$$y' = \frac{y \cdot d}{z + d}$$

wyliczeniem uporządkowanej pary współrzędnych rzutowanego punktu. Ta zmienna odegra doniosłą rolę podczas ustalania skali klipów filmowych i uchroni cię przed wykonywaniem dodatkowych operacji matematycznych. Zmiennej tej w naszych przykładach nadamy nazwę `perspective_ratio` (współczynnik perspektywy).

Zapiszmy zatem nasze równania perspektywy w ostatecznej postaci (rysunek 2.10).

Rysunek 2.10. *Ostateczna postać równań oraz współczynnika perspektywy*

$$perspective\_ratio = \frac{d}{z + d}$$

$$x' = x \cdot perspective\_ratio$$

$$y' = y \cdot perspective\_ratio$$

Zastosujmy naszą nowo nabytą wiedzę w skrypcie, który będzie przeliczał uporządkowaną trójkę liczb na uporządkowaną parę.

Uporządkowana trójka liczb określa położenie punktu w symulowanej przestrzeni, zatem gdy mamy dany punkt, znamy wartości tych liczb. Chwili zastanowienia wymaga natomiast wartość parametru *d*, czyli odległość oka obserwatora od ekranu monitora. Wracając do przykładu z drogą, jeśli dobierzemy zbyt małą wartość parametru *d*, uzyskamy efekt rybiego oka i krawędzie drogi będą się zbiegały zbyt szybko. Jeśli z kolei wartość ta będzie zbyt duża, obserwator może mieć trudności z odróżnieniem bliskich i odległych punktów, zaś krawędzie drogi będą zbiegały się zbyt wolno. Zwykle odpowiednie są wartości z przedziału od 200 do 500, choć powinieneś z nimi poeksperymentować, aby lepiej zrozumieć ich wpływ na rezultat.

Poniższy skrypt jest umieszczony w detektorze zdarzenia `load` klipu filmowego. (Klipem filmowym może być dowolny element, który chcesz powielać i umieszczać na obrazie). Skrypt losuje trójkę liczb z przedziału od –100 do 100, będących współrzędnymi punktu w przestrzeni, a następnie oblicza współrzędne rzutu tego punktu na ekranie monitora.

```
onClipEvent (load)
{
   // parametr - zakładana odległość obserwatora od ekranu
   D = 400;
   // losowanie trójki liczb
   x = Math.random () * 200 - 100;
   y = Math.random () * 200 - 100;
   z = Math.random () * 200 - 100;
   // współczynnik perspektywy
   perspective_ratio = D / (D + z);
   // położenie punktu zrzutowanego na ekran monitora
```

```
    perspective_x = x * perspective_ratio;
    perspective_y = y * perspective_ratio;
}
```

Zmiana położenia i skali na obrazie

Choć nasz skrypt oblicza pozycję trójwymiarowego punktu zrzutowanego na dwuwymiarowy ekran, nie jesteśmy jeszcze gotowi do wyrenderowania punktu („renderowanie" w tym kontekście oznacza ustalenie położenia i skali klipu filmowego reprezentującego punkt).

Obliczenie położenia punktu jest łatwiejszą częścią zadania. Jeśli masz punkt o współrzędnych (x, y, z) w przestrzeni i skonwertujesz je na współrzędne ekranu (x', y'), wciąż jeszcze nie wiesz, w którym miejscu obrazu Flasha powinieneś go umieścić. Aby to zrobić, musisz wcześniej zdefiniować początek trójwymiarowego układu współrzędnych (początek układu współrzędnych Flasha (0, 0) mieści się w lewym górnym narożniku obrazu). Początek trójwymiarowego układu współrzędnych to punkt, względem którego wyznaczamy położenie wszystkich innych punktów w przestrzeni; jest to jeden z najważniejszych elementów w naszych rozważaniach. Gdy znajdziesz współrzędne x' i y', przeliczasz je na ostateczne współrzędne obrazu, względem początku układu współrzędnych.

Poniższy przykładowy skrypt umieszcza klip filmowy na obrazie na podstawie podanego początku układu współrzędnych oraz położenia obiektu zrzutowanego w perspektywie. Położenie obiektu w perspektywie jest w tym przykładzie losowane.

```
onClipEvent (load)
{
    // początek trójwymiarowego układu współrzędnych
    // (umiejscowiony w dwuwymiarowym układzie współrzędnych)
    origin_x = 275;
    origin_y = 200;
    // oblicz położenie punktu rzutowanego w perspektywie
    perspective_x = Math.random () * 200 - 100;
    perspective_y = Math.random () * 200 - 100;
    perspective_y = y * perspective_ratio;
    /* ustal pozycję klipu na obrazie; znak minus przy obliczaniu współrzędnej y
    wynika z faktu, że oś y we Flashu jest odwrócona — dodatnia półoś jest skierowana
    w dół. */
    this._x = origin_x + perspective_x;
    this._y = origin_y - perspective_y;
}
```

Gdy pozycja klipu na ekranie jest już ustalona, musimy jeszcze zająć się jego skalą. Rozmiar klipu rzutowanego w perspektywie możesz dobrać, obserwując wartości liczbowe współczynnika perspektywy przy różnych wartościach współrzędnej z. Gdy wartość tej współrzędnej jest duża (czyli punkt jest odległy od ekranu), mianownik wzoru na współczynnik również staje się bardzo duży. Dzielenie przez dużą liczbę daje w wyniku małą liczbę, zatem gdy współrzędna z rośnie, współczynnik perspektywy maleje. Z drugiej strony, gdy z maleje, współczynnik perspektywy rośnie.

Właśnie takiego zachowania oczekujemy po skali obiektu w perspektywie. Gdy obiekt oddala się od nas (zwiększa się wartość z), jego pozorny rozmiar maleje. Gdy obiekt się zbliża, wydaje się nam, że jest większy. Jednak nie możemy wyrazić wymiarów obiektów bezpośrednio za pomocą współczynnika perspektywy — wówczas wszystkie obiekty

miałyby jednakowe rozmiary, ponieważ współczynnik ten nie bierze pod uwagę ich pierwotnych wymiarów. Dlatego mnożymy pierwotne wymiary klipu filmowego przez współczynnik perspektywy, otrzymując wymiary obiektu rzutowanego. W poniższym równaniu wielkość `regular_size` reprezentuje pierwotną wielkość klipu filmowego, zaś wielkość `perspective_size` to wielkość klipu rzutowanego w perspektywie.

```
perspective_size = regular_size * perspective_ratio;
```

Na podstawie naszego pierwszego skryptu, który losował trójkę liczb i konwertował ją na uporządkowaną parę, teraz możemy wyświetlić obiekt. Poniższy zmodyfikowany skrypt również umieszczamy w detektorze zdarzenia klipu filmowego. Inicjalizujemy również dodatkowe stałe, takie jak pierwotny rozmiar klipu filmowego, początek trójwymiarowego układu współrzędnych oraz zakładana odległość obserwatora od ekranu.

```
onClipEvent (load)
{
   // początek trójwymiarowego układu współrzędnych
   // (umiejscowiony w dwuwymiarowym układzie współrzędnych Flasha)
   origin_x = 275;
   origin_y = 200;
   // parametr - zakładana odległość obserwatora od ekranu
   D = 400;
   // pierwotny rozmiar klipu filmowego
   regular_size = this._width;
   // losowanie trójki liczb
   x = Math.random () * 200 - 100;
   y = Math.random () * 200 - 100;
   z = Math.random () * 200 - 100;
   // współczynnik perspektywy
   perspective_ratio = D / (D + z);
   // oblicz położenie punktu rzutowanego w perspektywie
   perspective_x = x * perspective_ratio;
   perspective_y = y * perspective_ratio;
   // aktualizuj położenie klipu na obrazie
   this._x = origin_x + perspective_x;
   this._y = origin_y - perspective_y;
   // aktualizuj skalę klipu
   this._xscale = this._yscale = regular_size * perspective_ratio;
}
```

Przeźroczystość klipu filmowego

Gdy obiekt znajduje się w dużej odległości od obserwatora, jego postrzegane kolory są mniej intensywne i zlewają się z tłem. Podobny efekt możemy uzyskać we Flashu, zmieniając wartość właściwości `_alpha` klipu filmowego, reprezentującego obiekt. Obliczanie przeźroczystości klipu jest podobne do obliczania jego wymiarów. Gdy obiekt znajduje się w swoim pierwotnym położeniu lub pomiędzy nim a obserwatorem, jego właściwość `_alpha` jest równa 100, czyli jest całkowicie kryjący. Gdy obiekt się oddala, właściwość `_alpha` maleje, aż do wartości 0. W taki sam sposób zachowuje się współczynnik perspektywy i rozmiar obiektu. Zatem mnożąc współczynnik perspektywy przez wartość 100, wyznaczymy wartość współczynnika `_alpha` w funkcji odległości na osi *z*.

```
//wartość przeźroczystości w funkcji odległości z
klip_filmowy._alpha = 100 * perspective_ratio;
```

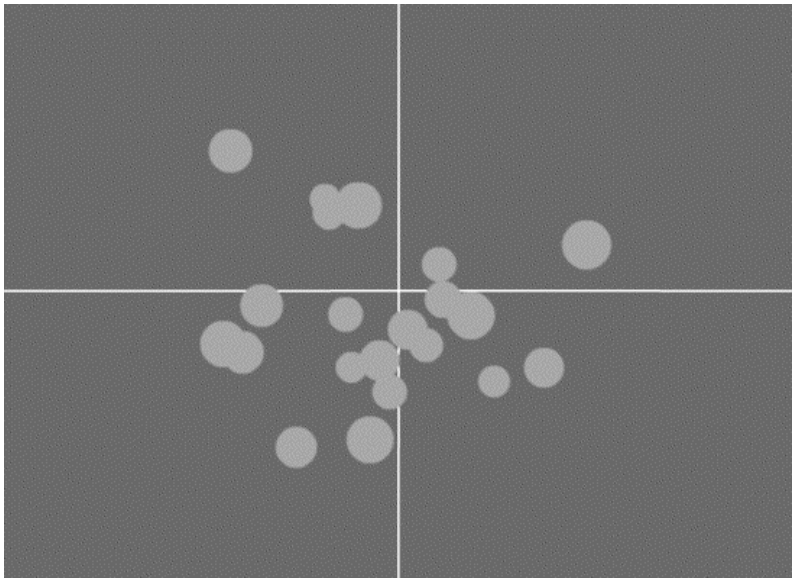
Choć efekt ten jest łatwo zaimplementować, w większości naszych przykładów nie będziemy z niego korzystać.

Losowo rozmieszczone punkty w przestrzeni

Czas na pierwszą praktyczną demonstrację. W tym przykładzie kilkakrotnie powielimy punkt, umieścimy jego klony w losowo wybranych punktach przestrzeni i zrzutujemy je na płaszczyznę ekranu.

Otwórz plik *random_3D_points.swf*, zapisany w katalogu *rozdzial_2* na płycie CD-ROM.

Uruchom plik kilka razy, by zobaczyć, że za każdym razem układ punktów w przestrzeni jest inny (rysunek 2.11).



Rysunek 2.11. *Flash tworzy skupisko punktów losowo rozmieszczonych w przestrzeni*

Otwórz plik *random_3D_points.fla*, zapisany na płycie CD-ROM.

Tworzenie tego projektu rozpocząłem od narysowania koła i skonwertowania go na klip filmowy. Następnie w klipie napisałem skrypt, który umieszcza punkt w przestrzeni i rzutuje go na ekran monitora. Gdy klip zostaje umieszczony na ekranie, losuje uporządkowaną trójkę liczb określającą położenie punktu i konwertuje ją na uporządkowaną parę, zgodnie z perspektywą. W tej sytuacji wystarczy kilkakrotnie zduplikować klipy filmowe, a one same zajmą się resztą. Duplikacja jest realizowana w pierwszej klatce głównej listwy czasowej (`_root`) i powinna być wykonana tylko jeden raz.

Poniższy kod umieszcza na obrazie 20 klonów klipu filmowego. Po zduplikowaniu klipów, zawarte w nich detektory zdarzenia `load` zajmują się obliczaniem ich pozycji i wymiarów. Tworzone klony klipów filmowych noszą nazwę `point`, po której następuje numer klonu.

```
// liczba powielanych punktów
num_points = 20;
// duplikacja klipów
// (pozostałymi czynnościami zajmą się skrypty wewnątrz klipów)
for (var j = 0; j < num_points; j++)
{
   this.point.duplicateMovieClip ("point" + j, j);
}
```

Kolejność nakładania klipów na osi z

Poprzednia demonstracja posiada jedną drobną wadę dotyczącą rzutowania obiektów — nie bierze pod uwagę faktu, że obiekty widoczne na pierwszym planie powinny zasłaniać te, które znajdują się na dalszym planie. Ponieważ w przykładzie tym użyłem jednokolorowych kół, kolejność zasłaniania obiektów nie była widoczna, zatem nie było widać również tej wady. Jednak wypełnienie kół prostym gradientem ujawniłoby tę niekonsekwencję w perspektywie. Rozwiązanie tego problemu jest niezwykle proste i efektywne; nie obciąża też zbytnio procesora.

Metoda swapDepths()

Wbudowana metoda `swapDepths()` obiektu *MovieClip* umożliwia łatwą zmianę poziomu warstwy, na którym mieści się klip filmowy, w zależności od odległości *z* tego klipu. Gdy w bezpośredni sposób przypiszesz klipowi numer poziomu, równy jego współrzędnej *z*, uzyskasz błędny wynik — klip filmowy z dużą współrzędną *z* powinien znajdować się głęboko w tle. Wystarczy jednak, że użyjesz współrzędnej *z* ze zmienionym znakiem — wówczas im dalej klip będzie się znajdował od ekranu, tym niższy będzie numer jego poziomu.

```
// zmiana poziomu klipu na podstawie współrzędnej z
this.swapDepths (-z);
```

Teraz możesz zmienić zwykły punkt (koło) w dowolną grafikę, jaką chciałbyś wyświetlać. Zobaczysz wówczas, że kolejne klony grafiki zasłaniają się w odpowiedniej kolejności, zgodnie z głębią perspektywy.

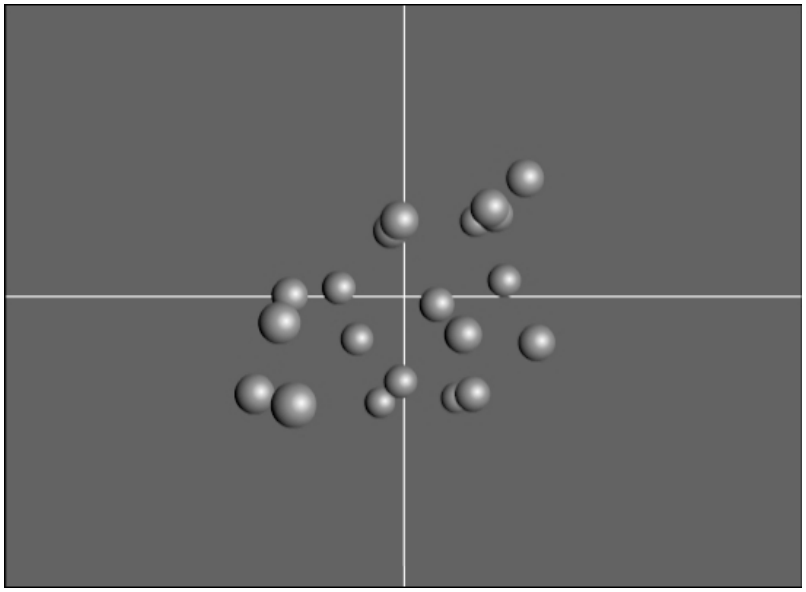
Być może nie podoba ci się rozwiązanie polegające na prostym negowaniu współrzędnej *z* i określaniu w ten sposób numeru poziomu dla klipu, ponieważ tym sposobem można uzyskać ujemne numery poziomów. Choć zajmę się jeszcze tym problemem, jednak możesz po prostu odjąć wartość współrzędnej *z* od dużej liczby, by zachować odpowiednią kolejność poziomów i operować tylko dodatnimi liczbami.

Losowo rozmieszczone obiekty z poprawną kolejnością na osi z

Zastosujmy nowe rozwiązanie i utwórzmy nowy plik, który losowo rozmieszcza obiekty w przestrzeni, a następnie rzutuje je na płaszczyznę ekranu, zachowując kolejność zasłaniania zgodną z położeniem na osi *z*.

Otwórz plik *random_3D_points_depth.swf* zapisany na płycie CD-ROM.

Uruchom ten plik kilka razy, aby zobaczyć, że dalsze (mniejsze) obiekty zawsze są zasłaniane przez bliższe (większe). Teraz znacznie łatwiej można rozróżnić odległości poszczególnych obiektów (rysunek 2.12).



Rysunek 2.12. *W tym przykładzie sortujemy poziomy klipów w zależności od ich współrzędnej z*

Otwórz plik *random_3D_points_depth.fla* zapisany na płycie CD-ROM.

Ten film jest niemal taki sam jak film *random_3D_points.fla*; zawiera on nieco inne elementy graficzne oraz dwa dodatkowe wiersze kodu. Wiersze te mieszczą się na końcu skryptu klipu filmowego. Ich zadaniem jest przypisanie klipowi poziomu warstwy, zgodnego z jego współrzędną *z*.

```
// zmiana poziomu klipu na podstawie współrzędnej z
this.swapDepths (-z);
```

W tym momencie dysponujemy już niemal kompletną procedurą służącą do symulacji trzech wymiarów.

Przesunięcia

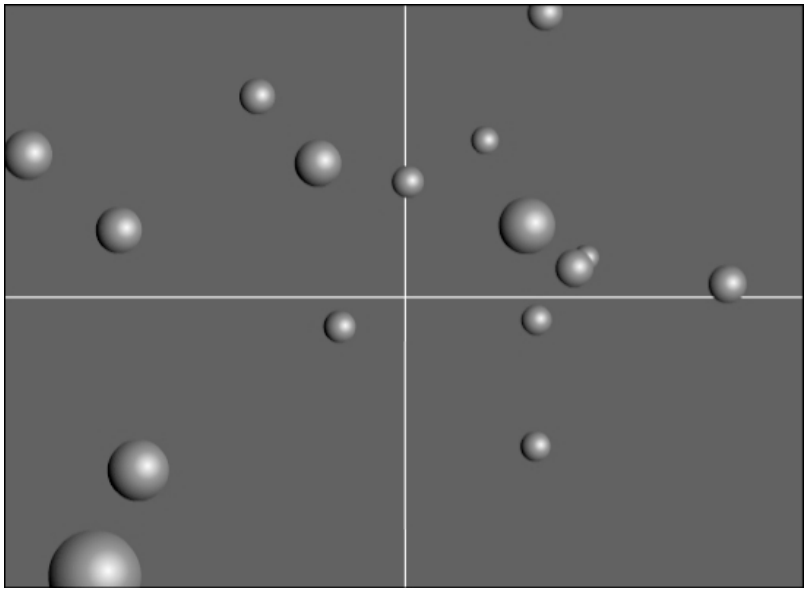
Aby przesunąć punkt w przestrzeni (czyli wykonać operację zwaną w matematyce translacją), zmieniasz wartość współrzędnej *x*, *y* lub *z*. Na przykład możesz przesunąć punkt (1, 3, 2) o cztery jednostki wzdłuż osi *x* do (5, 3, 2), następnie o trzy jednostki wzdłuż osi *y* do (5, 6, 2) i wreszcie o –4 jednostki wzdłuż osi *z* do (6, 5, –2).

Losowy ruch

We Flashu przesunięcia powinny być bardziej płynne. Aby uzyskać płynny ruch, musimy w kolejnych klatkach stopniowo zmieniać współrzędne punktów. Oznacza to, że zmianie ulegnie również układ naszego poprzedniego skryptu, który jedynie umieszczał punkty w przestrzeni i rzutował je na płaszczyznę. W następnym pliku rozmieścimy obiekty w przestrzeni w taki sam sposób jak poprzednio, lecz tym razem wprawimy je w ruch w przypadkowych kierunkach.

Otwórz plik *random_3D_points_trans1.swf* zapisany na płycie CD-ROM.

W chwili uruchomienia tego pliku każdy obiekt otrzymuje prędkość początkową, z jaką porusza się następnie w przypadkowym kierunku. Za każdym razem gdy uruchomisz plik, obiekty rozproszą się w innych kierunkach (rysunek 2.13).



Rysunek 2.13. *Losowo rozmieszczone obiekty poruszają się w przypadkowych kierunkach*

Otwórz plik *random_3D_points_trans1.fla* zapisany na płycie CD-ROM.

Tym razem nie będziemy obsługiwać wszystkich czynności wewnątrz detektora zdarzenia `load`; ustalimy jedynie pozycje, stałe i losowe wartości przyrostów związanych z chwilowymi przesunięciami obiektów. Później możesz umożliwić zmianę przyrostów za pomocą przycisków, zdarzeń klipów filmowych lub innych metod interaktywnych. Na razie jednak pozostałem przy niewielkich przyrostach, byś mógł zaobserwować stopniowe zmiany położenia.

```
onClipEvent (load)
{
   // początek trójwymiarowego układu współrzędnych
   // (umiejscowiony w dwuwymiarowym układzie współrzędnych Flasha)
   origin_x = 275;
```

```
    origin_y = 200;
    // parametr - zakładana odległość obserwatora od ekranu
    D = 400;
    // pierwotny rozmiar klipu filmowego
    regular_size = this._width;
    // losowanie uporządkowanej trójki liczb
    x = Math.random () * 200 - 100;
    y = Math.random () * 200 - 100;
    z = Math.random () * 200 - 100;
    // przyrost współrzędnych na osiach x, y i z
    inc_x = Math.random () * 2 - 1;
    inc_y = Math.random () * 2 - 1;
    inc_z = Math.random () * 2 - 1;
}
```

W powyższym kodzie wyeliminowaliśmy akcje związane z rzutowaniem obiektów na płaszczyznę i dodaliśmy trzy nowe zmienne. Ich wartości w każdej klatce są dodawane do współrzędnych obiektu, co wprawia go w ruch. Choć efekt ten nie jest niczym nadzwyczajnym, pozwala nam zrozumieć zjawisko ruchu.

Detektor zdarzenia `enterFrame` reaguje na przejście klipu do następnej klatki, zatem zawarty w nim kod jest wykonywany w każdej klatce. Kod ten zajmuje się aktualizacją położenia obiektów, czyli dodawaniem do nich przyrostów, zdefiniowanych w powyższym kodzie. Po przesunięciu obiektu w nowe miejsce, jest on rzutowany z zastosowaniem tego samego kodu, co w poprzednim przykładzie.

```
onClipEvent (enterFrame)
{
    // przesuń punkt zgodnie z przyrostami
    x += inc_x;
    y += inc_y;
    z += inc_z;
    // współczynnik perspektywy
    perspective_ratio = D / (D + z);
    // oblicz położenie punktu rzutowanego w perspektywie
    perspective_x = x * perspective_ratio;
    perspective_y = y * perspective_ratio;
    // aktualizuj położenie klipu na obrazie
    this._x = origin_x + perspective_x;
    this._y = origin_y - perspective_y;
    // aktualizuj skalę klipu
    this._xscale = this._yscale = regular_size * perspective_ratio;
     // zmień poziom warstwy dla klipu zgodnie ze współrzędną z
    this.swapDepths (-z);
}
```

Możesz mi wierzyć lub nie, lecz właśnie utworzyłeś skrypt dla następnej demonstracji Flasha! Gdy utworzysz kilka klonów obiektów z tym skryptem, zobaczysz skupisko obiektów poruszających się w przypadkowych kierunkach przestrzeni.

Sterowany ruch

Elementy rozbiegające się w przypadkowych kierunkach prawdopodobnie nie były twoim zamysłem, gdy przystępowałeś do czytania tego rozdziału. Możesz jednak zaangażować

widza w trójwymiarowy świat, dając mu możliwość sterowania ruchem obiektów. Układ następnego skryptu pełni ważną rolę ze względu na jego interaktywne komponenty, dlatego na tym się skoncentrujemy.

Sztuczka z wykrywaniem wciśnięcia klawisza

Stosując prostą sztuczkę, możemy wykrywać wciśnięcie określonych klawiszy podczas odtwarzania zapętlonych pętli, bez tworzenia następnego detektora zdarzenia klipu filmowego. Sztuczka ta jest szczególnie przydatna w sytuacji, gdy wartości zmiennych muszą być zmieniane w sposób ciągły, gdy klawisz jest wciśnięty. Metoda `isDown()` obiektu *Key* zwraca wartość 1, gdy klawisz określony w argumencie jest wciśnięty; w przeciwnym razie zwraca wartość 0. Wynik działania tej metody, czyli liczbę 1 lub 0, możesz przemnożyć przez inną wartość, otrzymując tym sposobem wartość zmiennej, gdy klawisz jest wciśnięty, lub zero, gdy jest zwolniony. Metody tej możesz użyć w następujący sposób:

```
x += Key.isDown(Key.wybrany_klawisz) * inc;
```

Zmienna `x` jest zwiększana o wartość `inc` w każdej klatce, w której `wybrany_klawisz` jest wciśnięty. W podobny sposób możesz zrealizować zmniejszanie wartości zmiennej:

```
x -= Key.isDown(Key.inny_klawisz) * inc;
```

Cały mechanizm modyfikujący wartość zmiennej możesz nawet umieścić w pojedynczym wierszu. Będzie on modyfikował wartość zmiennej, w każdej klatce dodając do niej wynik pewnego wyrażenia. W wyrażeniu tym znajdują się oba czynniki, czyli przyrost wprowadzany przez klawisz inkrementacji (zwiększania) oraz przyrost (z przeciwnym znakiem) wprowadzany przez klawisz dekrementacji (zmniejszania).

```
x += Key.isDown(Key.inc_key) * inc - Key.isDown(Key.dec_key) * inc;
```

Gdy jest wciśnięty klawisz `inc_key`[3], wartość zmiennej `x` w każdej klatce zwiększa się o `inc`. Gdy z kolei jest wciśnięty klawisz `dec_key`, wartość zmiennej `x` zmniejsza się w każdej klatce o `inc`. Jeśli użytkownik wciśnie naraz oba klawisze, wartość zmiennej `x` pozostanie niezmieniona, ponieważ program doda, a następnie odejmie od niej wartość `inc`.

Zobaczmy, jak ta sztuczka działa w praktyce. Poniższy skrypt przesuwa klip filmowy na obrazie, gdy użytkownik wciśnie jeden (lub kilka) z klawiszy strzałek. Utwórz nowy film Flasha, narysuj prostą grafikę i przekształć ją w klip filmowy. Umieść poniższy kod odpowiednio w detektorach zdarzeń `load` i `enterFrame` klipu filmowego.

```
onClipEvent (load)
{
   // jednostkowy przyrost - o tyle klip przesunie się w każdej klatce,
   // w której użytkownik wciśnie odpowiedni klawisz
   inc = 4;
}
```

[3] Oczywiście za tą nazwą, podobnie jak za nazwą `dec_key`, kryją się nazwy konkretnych klawiszy — *przyp. tłum.*

Zdarzenie `load` klipu filmowego, czyli jego wczytanie, inicjalizuje zmienną, której wartość jest przyrostem współrzędnych klipu filmowego. Z kolei zdarzenie `enterFrame`, czyli przejście do nowej klatki klipu filmowego, uruchamia skrypt zajmujący się modyfikowaniem współrzędnych *x* i *y* klipu; działanie tego skryptu omówiliśmy przed chwilą.

Zanim jednak utworzysz ten skrypt, musisz zastanowić się nad doborem klawiszy, które będą służyły do sterowania ruchem klipu filmowego. Ponieważ oś *x* we Flashu ma taki sam przebieg jak w tradycyjnym układzie współrzędnych, możemy przyjąć, że klawisz lewej strzałki będzie powodował zmniejszenie wartości współrzędnej *x*, natomiast klawisz prawej strzałki będzie powodował jej zwiększenie. Z kolei oś *y* we Flashu jest zwrócona w przeciwną stronę, niż w tradycyjnym układzie współrzędnych, dlatego klawisz strzałki w górę będzie powodował zmniejszenie wartości współrzędnej *y*, zaś klawisz strzałki w dół jej zwiększenie.

```
onClipEvent (enterFrame)
{
   // przesuń klip filmowy zgodnie z wciśniętymi klawiszami
   this._x += Key.isDown (Key.RIGHT) * inc - Key.isDown (Key.LEFT) * inc;
   this._y += Key.isDown (Key.DOWN) * inc - Key.isDown (Key.UP) * inc;
}
```

Otwórz plik *key_move_sample.fla* zapisany na płycie CD-ROM.

Plik ten demonstruje działanie omówionej przez nasz sztuczki z wykrywaniem wciśnięcia klawiszy. Zawiera on jedynie 13 wierszy kodu, zawartych wewnątrz detektorów zdarzeń klipu filmowego. Skrypt jest identyczny z utworzonym przez nas.

Sterowanie ruchem za pomocą klawiszy

Zastosujmy poznaną technikę, tworząc następną aplikację z udziałem trójwymiarowej grafiki Flasha. Obiekty powielamy i rozmieszczamy w przestrzeni tak jak poprzednio, jednak tym razem użytkownik może przesuwać je we wszystkie strony.

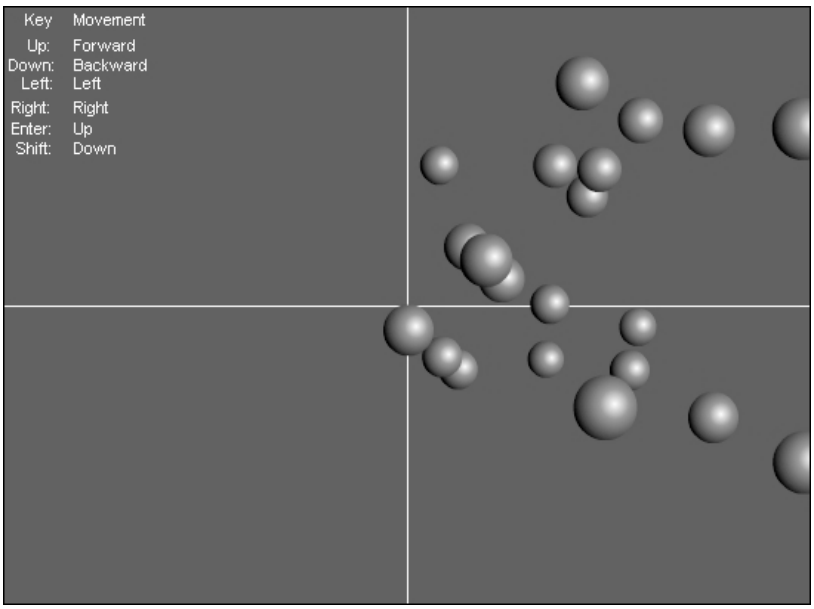
Otwórz plik *random_3D_points_trans2.swf* zapisany na płycie CD-ROM.

Używając klawiszy *Shift*, *Enter* oraz klawiszy strzałek, możesz przemieszczać grupę obiektów w przestrzeni (rysunek 2.14).

Otwórz plik *random_3D_points_trans2.fla* zapisany na płycie CD-ROM.

Cały kod, oprócz początkowych akcji związanych z klonowaniem, umieścimy w klipie filmowym obiektu. Trzy zdefiniowane wcześniej zmienne, przechowujące losowe przyrosty, zastąpimy jedną wartością, reprezentującą jednakowe przyrosty współrzędnych we wszystkich kierunkach. W związku z tym kod wewnątrz detektora zdarzenia `load` ulegnie niewielkiej zmianie.

```
onClipEvent (load)
{
   // początek trójwymiarowego układu współrzędnych
```



Rysunek 2.14. *Steruj ruchem obiektów za pomocą klawiszy strzałek oraz klawiszy Shift i Enter*

```
    // (umiejscowiony w dwuwymiarowym układzie współrzędnych Flasha)
    origin_x = 275;
    origin_y = 200;
    // parametr - zakładana odległość obserwatora od ekranu
    D = 400;
    // pierwotny rozmiar klipu filmowego
    regular_size = this._width;
    // losowanie uporządkowanej trójki liczb
    x = Math.random () * 200 - 100;
    y = Math.random () * 200 - 100;
    z = Math.random () * 200 - 100;
    // jednostkowy przyrost - o tyle klip przesunie się w każdej klatce,
    // w której użytkownik wciśnie odpowiedni klawisz
    inc = 4;
}
```

Bezpośrednio po utworzeniu klonu klipu filmowego inicjalizujemy stałe, takie jak położenie początku trójwymiarowego układu współrzędnych, pierwotny rozmiar klipu filmowego czy zakładana odległość obserwatora od ekranu. Skrypt losuje również wartości dla trójki współrzędnych oraz definiuje stały przyrost, o jaki zmieni się współrzędna klipu, gdy użytkownik wciśnie odpowiedni klawisz. W poprzednim przykładzie losowaliśmy wartości przyrostu oddzielnie dla każdej współrzędnej; teraz ograniczamy się do ustalenia pojedynczej wartości.

Detektor zdarzenia `enterFrame` klipu filmowego zawiera kod, który zajmuje się rzutowaniem i ruchem punktów. W poprzednim przykładzie wylosowane przyrosty były dodawane do współrzędnych klipu w każdej klatce. W tym przykładzie modyfikowanie wartości współrzędnych jest uzależnione od klawiszy wciśniętych przez użytkownika. Ponieważ nie mamy już do czynienia z ruchem na płaszczyźnie dwuwymiarowej, musiałem zmienić funkcje poszczególnych klawiszy. Klawisze strzałek w lewo i w prawo

sterują ruchem wzdłuż osi *x*; klawisze *Enter* i *Shift* sterują ruchem wzdłuż osi *y*; natomiast klawisze strzałek w górę i w dół sterują ruchem wzdłuż osi *z*. Część skryptu zajmująca się rzutowaniem obiektów na płaszczyznę ekranu pozostaje niezmieniona.

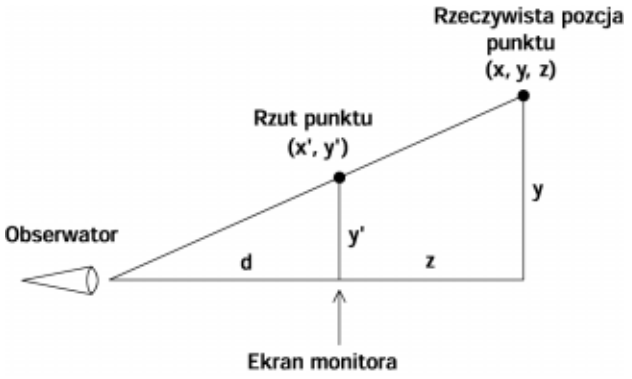
```
onClipEvent (enterFrame)
{
   // przesuń klip filmowy zgodnie z wciśniętymi klawiszami
   x += Key.isDown (Key.LEFT) * inc - Key.isDown (Key.RIGHT) * inc;
   y += Key.isDown (Key.SHIFT) * inc - Key.isDown (Key.ENTER) * inc;
   z += Key.isDown (Key.DOWN) * inc - Key.isDown (Key.UP) * inc;
   // współczynnik perspektywy
   perspective_ratio = D / (D + z);
   // oblicz położenie punktu rzutowanego w perspektywie
   perspective_x = x * perspective_ratio;
   perspective_y = y * perspective_ratio;
   // aktualizuj położenie klipu na obrazie
   this._x = origin_x + perspective_x;
   this._y = origin_y - perspective_y;
   // aktualizuj skalę klipu
   this._xscale = this._yscale = regular_size * perspective_ratio;
   // zmień poziom warstwy dla klipu zgodnie ze współrzędną z
   this.swapDepths (-z);
}
```

Może po wyglądzie skryptu tego nie widać, jednak utworzyliśmy naprawdę imponujący plik!

Problem z mechanizmem rzutowania

Gdy dajesz użytkownikowi pełną kontrolę nad środowiskiem trójwymiarowym, może się okazać, że mechanizm symulacji perspektywy posiada pewne słabe punkty. Jeden z problemów może się pojawić, gdy użytkownik spróbuje wejść zbyt „głęboko" w przestrzeń i jeden lub kilka obiektów w symulowanej przestrzeni znajdzie się „za" jego głową. Wówczas obiekty te niespodziewanie wracają na ekran, przy czym są odwrócone względem wszystkich osi. Rozwiązanie tego problemu polega na wyłączeniu widzialności tych obiektów, które przesuną się poza zakładane położenie oka obserwatora.

Jak można sprawdzić, czy obiekt znalazł się z tyłu obserwatora? Wróćmy do rysunku, który pomógł nam w wyprowadzeniu równań perspektywy (rysunek 2.15).



Rysunek 2.15. *Obiekt znajduje się z tyłu obserwatora, gdy jego współrzędna z jest mniejsza od ujemnej wartości odległości d*

Początek trójwymiarowego układu współrzędnych mieści się bezpośrednio na płaszczyźnie ekranu monitora. Wszystko, co mieści się po twojej stronie ekranu, posiada ujemną wartość współrzędnej *z*; z kolei wszystko, co mieści się po drugiej jego stronie (czyli „wewnątrz" monitora), ma dodatnią wartość współrzędnej *z*. Zakładaną odległość od twojego oka do monitora określiliśmy jako *d* (wpływa ona na obliczany współczynnik perspektywy `perspective_ratio`). Innymi słowy, jeśli chcielibyśmy określić położenie twojego oka w symulowanej przestrzeni trójwymiarowej, miałoby ono współrzędne (0, 0, –*d*). Jeśli zatem współrzędna *z* obiektu przyjmie wartość mniejszą od –*d*, obiekt ten znajdzie się za tobą (a dokładniej, za twoim okiem).

Korzystając z tej informacji, możemy utworzyć wyrażenie warunkowe, które będzie sprawdzać, czy obiekt znalazł się z tyłu użytkownika; jeśli tak, wyłączymy widzialność obiektu. Ponieważ w sytuacji, gdy klip mieści się poza użytkownikiem, w ogóle nie musi być on wyświetlany na obrazie, możesz zaoszczędzić Flashowi obliczeń, umieszczając wewnątrz klauzuli `else` cały fragment skryptu, zajmujący się rzutowaniem klipu na ekran.

Poniższy skrypt będzie bardziej wymowny niż moje słowa; zawartość detektora `onClipEvent (enterFrame)` rzutuje trójwymiarowe obiekty na płaszczyznę ekranu.

```
onClipEvent (enterFrame)
{
   // przesuń klip filmowy zgodnie z wciśniętymi klawiszami
   x += Key.isDown (Key.LEFT) * inc - Key.isDown (Key.RIGHT) * inc;
   y += Key.isDown (Key.SHIFT) * inc - Key.isDown (Key.ENTER) * inc;
   z += Key.isDown (Key.DOWN) * inc - Key.isDown (Key.UP) * inc;
   // sprawdź, czy obiekt znalazł się poza obserwatorem
   if (z < -D)
   {
      // wyłącz widzialność klipu
      this._visible = false;
   }
   else
   {
      // włącz widzialność klipu
      this._visible = true;
       // współczynnik perspektywy
      perspective_ratio = D / (D + z);
      // oblicz położenie punktu rzutowanego w perspektywie
      perspective_x = x * perspective_ratio;
      perspective_y = y * perspective_ratio;
      // aktualizuj położenie klipu na obrazie
      this._x = origin_x + perspective_x;
      this._y = origin_y - perspective_y;
      // aktualizuj skalę klipu
      this._xscale = this._yscale = regular_size * perspective_ratio;
      // zmień poziom warstwy dla klipu zgodnie ze współrzędną z
      this.swapDepths (-z);
   }
}
```

Zaprezentowany skrypt działa bardzo podobnie do poprzedniego przykładu. Jedyna różnica polega na tym, że dodatkowe wyrażenie warunkowe wyłącza widzialność klipu, gdy znajduje się on z tyłu obserwatora. Przykład ten znajdziesz w pliku *random_3D_points_trans3.fla* na płycie CD-ROM.

Obrót

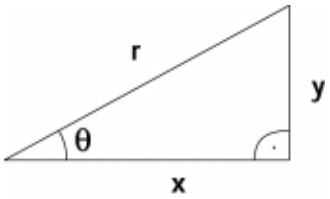
Obrót jest jednym z typowych efektów przeprowadzanych w przestrzeni trójwymiarowej. Daje wrażenie rzeczywistego przebywania w środowisku trójwymiarowym, ponieważ pozwala się w niej obracać i oglądać ją ze wszystkich stron. Choć jest to najbardziej skomplikowany aspekt omawiany do tej pory w tym rozdziale, jest zrozumiały nawet dla matematycznego nowicjusza.

Wstęp do obrotu

Zanim przystąpimy do tworzenia kodu, przypomnijmy kilka podstawowych zagadnień, głównie z trygonometrii.

Dwie trygonometryczne funkcje, z których będziemy teraz intensywnie korzystać, to sinus (symbol *sin*) oraz kosinus (symbol *cos*). Obie funkcje jako argument przyjmują wartość kąta. Aby zobrazować wyniki zwracane przez te funkcje, narysujmy trójkąt prostokątny (rysunek 2.16).

Rysunek 2.16. *Wartości funkcji sinus i kosinus są obliczane na podstawie długości boków trójkąta prostokątnego*



Wartość sinusa kąta θ jest równa długości *y* przyprostokątnej[4] leżącej naprzeciw kąta θ, podzielonej przez długość *r* przeciwprostokątnej[5]. Kosinus kąta θ jest z kolei równy długości *x* przyprostokątnej leżącej przy kącie θ, podzielonej przez długość *r* przeciwprostokątnej. Definicje te dla utrwalenia zanotujemy na rysunku 2.17.

Rysunek 2.17. *Równania funkcji sinus i kosinus*

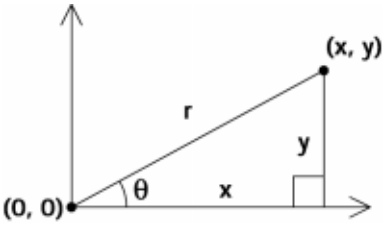
$$\sin\theta = \frac{y}{r}$$

$$\cos\theta = \frac{x}{r}$$

Na podstawie tych prostych definicji możemy sformułować nasze pierwsze równania, których użyjemy w obliczeniach. Wielkości *x* i *y* reprezentują podstawę i wysokość trójkąta prostokątnego, powstałego przez ustawienie boku *r* pod kątem θ. Gdy umieścimy początek układu współrzędnych w wierzchołku kąta θ, możemy również interpretować wielkości *x* i *y* jako współrzędne wierzchołka drugiego kąta ostrego w trójkącie prostokątnym (rysunek 2.18).

[4] Czyli boku, który styka się z kątem prostym — *przyp. tłum.*

[5] Czyli boku, który nie styka się z kątem prostym — *przyp. tłum.*



Rysunek 2.18. *Trójkąt prostokątny z wierzchołkami osadzonymi w układzie współrzędnych*

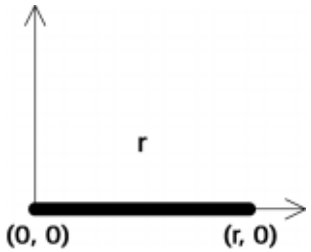
Przekształcając równania sinusa i kosinusa, możemy utworzyć równania na długość boków *x* i *y* (rysunek 2.19).

$$\cos\theta = \frac{x}{r} \longrightarrow x = r\cos\theta$$

$$\sin\theta = \frac{y}{r} \longrightarrow y = r\sin\theta$$

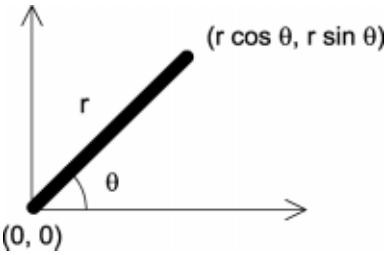
Rysunek 2.19. *Proste przekształcenie równań sinusa i kosinusa daje równania na długości boków x i y*

Powyższy zestaw równań pozwala nam znaleźć współrzędne *x* i *y* punktu, gdy znamy długość boku *r* i wartość kąta θ. Jednak jakie znaczenie mają wielkości *r* i θ? Wyobraź sobie prosty patyk leżący na płaskiej podłodze, z jednym z wierzchołków umieszczonym w punkcie (0, 0) układu współrzędnych, zaś drugim wierzchołkiem w punkcie (0, *r*), przy czym *r* jest długością patyka (rysunek 2.20).



Rysunek 2.20. *Patyk z jednym z wierzchołków w początku układu współrzędnych, zaś drugim wierzchołkiem na osi x*

Gdybyś obrócił patyk o kąt θ wokół środka układu współrzędnych, jeden z wierzchołków pozostałby w początku układu współrzędnych, zaś drugi znalazłby się gdzieś na płaszczyźnie *xy*. Możemy dokładnie ustalić, jakie jest położenie drugiego wierzchołka. Równania sinusa i kosinusa, przekształcone do postaci przedstawionej na rysunku 2.19, pozwalają nam odnaleźć współrzędne *x* i *y*, jeśli znamy wielkości *r* i θ (rysunek 2.21).



Rysunek 2.21. *Za pomocą wyprowadzonych równań możesz znaleźć współrzędne wierzchołka obróconego patyka*

Z powyższego rysunku wynika, że gdy przesuniemy się o długość *r* w kierunku θ, będziemy mogli odnaleźć współrzędne punktu, w którym się znajdziemy, mnożąc wartość *r* przez sinus lub kosinus kąta θ. Słowem, uporządkowaną parę liczb (*x*, *y*) można też zapisać w postaci (*r* cos θ, *r* sin θ).

W trygonometrii istnieje również pojęcie *tożsamości trygonometrycznej*. Jest równanie zawierające funkcje trygonometryczne, które ułatwia rozwiązywanie problemów i upraszcza specyficzne obliczenia. Tożsamości trygonometryczne są szczególnie przydatne wówczas, gdy potrzebujesz ogólnego wzoru na sinus lub kosinus sumy, różnicy lub iloczynu dwóch kątów. Dla nas przydatne będą dwie tożsamości trygonometryczne, zwane tożsamościami dla sumy kątów (rysunek 2.22).

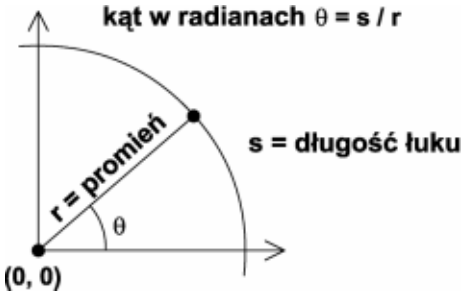
Rysunek 2.22. *Tożsamość sinusa i kosinusa sumy kątów*

$$\sin(a + b) = \sin a \cos b + \cos a \sin b$$
$$\cos(a + b) = \cos a \cos b - \sin a \sin b$$

Jeśli jesteś dociekliwy, możesz znaleźć dowody powyższych tożsamości w większości książek matematycznych. My poprzestaniemy na ich użyciu, co nastąpi już wkrótce.

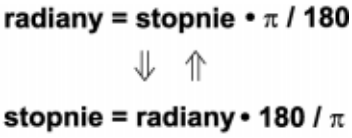
Skoro zajmujemy się kątami, sinusami i kosinusami, powinienem wspomnieć o pewnym dziwactwie Flasha związanym z obsługą funkcji trygonometrycznych. Istnieją dwie różne miary kątów — stopnie i radiany. Niemal wszystkim znane są stopnie i każdy wie, że w okręgu mieści się 360 stopni, czyli pełny kąt. Mniej znane są radiany. Kąt wyrażony w radianach to stosunek długości łuku o określonym promieniu do długości tego promienia (rysunek 2.23).

Rysunek 2.23. *Kąt wyrażony w radianach*



Aby ułatwić sobie życie, przyjmijmy, że promień łuku *r* ma długość 1. W tym wyjątkowym przypadku miara radianowa kąta upraszcza się do długości łuku *s*. Długość łuku w pełnym kącie (czyli po prostu obwód okręgu) wynosi 2π*r*. Jeśli promień okręgu *r* wynosi 1, wówczas jego obwód jest równy 2π. Stąd wynika, że kąt 360 stopni jest równy kątowi 2π radianów. Jeśli zatem mamy kąt 360 stopni, możemy przeliczyć go na radiany, mnożąc go przez wartość 2π i dzieląc przez 360; jeśli uprościmy to działanie, wystarczy przemnożenie kąta przez π i podzielenie przez 180. Korzystając z tego współczynnika możesz przeliczać stopnie na radiany (mnożąc stopnie przez współczynnik) lub radiany na stopnie (dzieląc radiany przez współczynnik). Rysunek 2.24 przedstawia wzór na przeliczanie kątów w stopniach na radiany i z powrotem.

Rysunek 2.24.
Konwersja pomiędzy stopniami i radianami



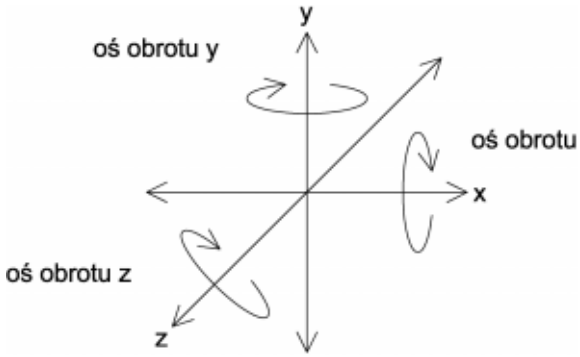
Obiekt *Math* Flasha, który udostępnia wszystkie funkcje trygonometryczne, korzysta z radianowej miary kątów, dlatego przed skorzystaniem z dowolnej funkcji trygonometrycznej musisz przeliczyć stopnie na radiany. Jak się dalej przekonasz, najlepiej jest operować kątami obrotów w stopniach i przeliczać je na radiany bezpośrednio przed użyciem funkcji trygonometrycznych Flasha.

Do czego sprowadza się obrót

Jeśli chcesz wyprowadzić równania obrotu punktu w przestrzeni, najpierw musisz wiedzieć, jak sformułować problem. Ponadto musisz sobie dokładnie uświadomić, co oznacza obrót wokół osi *x* lub dowolnej innej osi.

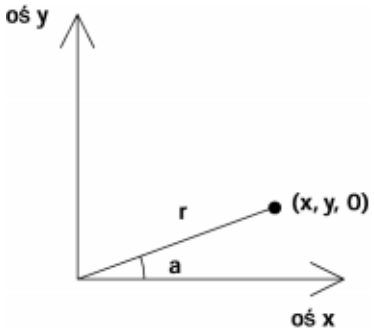
Gdy obracasz punkt wokół osi *x*, oznacza to, że ta oś układu współrzędnych staje się jednocześnie osią obrotu. Podobnie jest w przypadku obrotu wokół innych osi. Wart podkreślenia jest fakt, że podczas obrotu wokół osi *x* punkt nie zmienia wartości swojej współrzędnej *x*, zmienia natomiast wartości współrzędnych *y* i *z*. Podobnie, gdy obracasz punkt wokół osi *z*, zmieniają się jego współrzędne *x* i *y*, natomiast współrzędna *z* pozostaje niezmieniona. Na obrazie Flasha obrót wokół osi *z* ma taki sam skutek, jak obrót wokół środka układu współrzędnych w dwuwymiarowym, kartezjańskim układzie współrzędnych (rysunek 2.25).

Rysunek 2.25.
Obroty punktu wokół poszczególnych osi trójwymiarowego układu współrzędnych



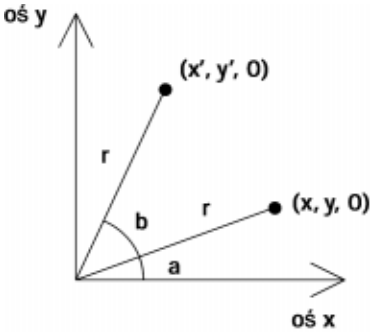
Obrót wokół osi z

Sformułujmy równania opisujące obrót punktu wokół osi *z*; dzięki temu będzie nam łatwiej opisać następnie obrót wokół pozostałych osi. Wyobraź sobie punkt na płaszczyźnie *xy*, którego współrzędna *z* jest chwilowo nieistotna (przyjmijmy, że wynosi 0). Punkt ma zatem współrzędne (*x*, *y*, 0), jest oddalony o *r* od początku układu współrzędnych, zaś łącząca go z nim linia jest nachylona do osi *x* pod kątem *a* (rysunek 2.26).



Rysunek 2.26. *Linia łącząca punkt na płaszczyźnie xy z początkiem układu współrzędnych tworzy kąt z osią x*

Teraz obróć ten punkt o kąt *b* wokół początku układu współrzędnych (lub wokół osi *z* prostopadłej do płaszczyzny *xy*). Obrócony punkt przyjmie współrzędne (*x'*, *y'*, 0), zachowując przy tym pierwotną odległość *r* od początku układu współrzędnych (rysunek 2.27).



Rysunek 2.27. *Punkt obrócony wokół osi z*

Zwróć uwagę, że teraz linia łącząca obrócony punkt z początkiem układu współrzędnych tworzy z osią *x* kąt o wartości *a+b*, a nie tylko *b*. Kąt *b* to kąt, o jaki obróciliśmy pierwotną konfigurację punktu. Zatem problem obrotu punktu wokół osi *z* sprowadza się do znalezienia nowych współrzędnych (*x'*, *y',* 0), przy czym znane są wartości *a*, *b*, *r* oraz pierwotne współrzędne (*x, y*, 0).

Obrót z użyciem trygonometrii

Rozpocznijmy od zapisania wzorów funkcji sinus i kosinus, przekształconych do postaci umożliwiającej znalezienie współrzędnych *x* i *y* pierwotnego i obróconego punktu (rysunek 2.28).

```
x = r cos a
y = r sin a
z = 0

x' = r cos (a + b)
y' = r sin (a + b)
z' = 0
```

Rysunek 2.28. *Wzory umożliwiające znalezienie współrzędnych pierwotnego i obróconego punktu*

Być może zaczynasz się już domyślać, dlaczego wcześniej wspomnieliśmy o tożsamościach trygonometrycznych dla sumy kątów. Powyższe równania nie są dla nas żadną nowością. Musimy jedynie pamiętać, że obrócony punkt tworzy z osią *x* kąt *a+b*. Choć równania te mogłyby być użyteczne podczas obracania punktu, jednak ich zastosowanie byłoby uciążliwe. Musielibyśmy cały czas śledzić odległość punktu od pierwotnej pozycji oraz pierwotnego kąta nachylenia. Jest to możliwe do zrealizowania, lecz nie jest konieczne. Korzystając z tożsamości trygonometrycznych dla sumy kątów, wyprowadzimy bardziej praktyczne wzory (rysunek 2.29).

Rysunek 2.29. *Korzystamy z tożsamości trygonometrycznych dla sumy kątów, podstawiając je do wzorów*

```
x = r cos a
y = r sin a
z = 0

x' = r cos (a + b) = r (cos a cos b - sin a sin b)
y' = r sin (a + b) = r (sin a cos b + cos a sin b)
z' = 0
```

Zwróć uwagę, że tożsamości dla sumy kątów nie zawierają wielkości *r*, zatem podczas podstawiania tożsamości do wzorów wielkość *r* musi pozostać poza nawiasami. Jednak aby uprościć nasze wyrażenia, możemy przemnożyć wyraz *r* przez wnętrze nawiasów (rysunek 2.30).

Rysunek 2.30. *Mnożymy wyraz r przez zawartość nawiasów*

```
x = r cos a
y = r sin a
z = 0

x' = r cos (a + b) = r cos a cos b - r sin a sin b
y' = r sin (a + b) = r sin a cos b + r cos a sin b
z' = 0
```

Jeśli jesteś spostrzegawczy, z pewnością zauważysz, że w ostatnim zestawie równań występują pewne wspólne elementy. Równania na *x'*, *y'* i *z'* zawierają człony, które są identyczne z równaniami na *x*, *y* i *z*. Mówimy oczywiście o członach (*r* cos *a*) oraz (*r* sin *a*), które są odpowiednio równe wielkościom *x* i *y*. Korzystając z tego faktu, możemy zastąpić te człony wielkościami *x* i *y* (rysunek 2.31).

Tym sposobem otrzymałeś równania opisujące obrót punktu wokół osi *z*. Jeśli znasz pierwotne położenie punktu (*x*, *y*, 0) i chcesz obrócić go wokół osi *z* o *b* stopni, po prostu użyj ostatnich dwóch równań. Zwróć uwagę, że pozbyliśmy się w nich wszelkich kłopotliwych wielkości, takich jak *r* czy *a*.

Przykład obrotu wokół osi z

W powyższych rozważaniach było widać, że obrót wokół osi *z* nie angażuje trzeciego wymiaru (ponieważ obracane punkty pozostają na płaszczyźnie *xy*). Przyjrzyjmy się kilku niewielkim demonstracjom, aby rozwiać wszelkie wątpliwości.

Rysunek 2.31. *Uprość równania, zastępując podobne człony*

$$x = r \cos a$$
$$y = r \sin a$$
$$z = 0$$

$$x' = r \cos (a + b) = x \cos b - y \sin b$$
$$y' = r \sin (a + b) = y \cos b + x \sin b$$
$$z' = 0$$

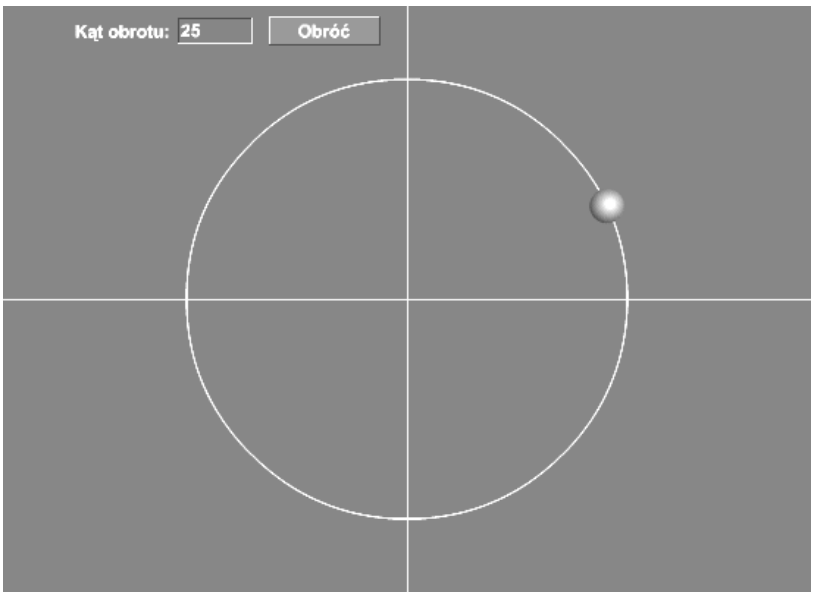
$$\Downarrow$$

$$x' = x \cos b - y \sin b$$
$$y' = y \cos b + x \sin b$$

Otwórz plik *z_axis_rotation1.swf*, zapisany na płycie CD-ROM.

Za każdym razem gdy klikniesz przycisk *Obróć*, punkt obróci się wokół początku układu współrzędnych o kąt podany w polu *Kąt obrotu* (rysunek 2.32).

Rysunek 2.32. *Obrót punktu wokół osi z o kąt podany przez użytkownika*



Otwórz plik *z_axis_rotation1.fla* zapisany na płycie CD-ROM.

W filmie utworzyłem pole tekstowe (związane ze zmienną `rotation_angle`), służące do wpisywania kąta obrotu punktu, przycisk wykonujący obrót oraz klip filmowy (o nazwie *point*) reprezentujący obracany punkt. Film zawiera dwa skrypty. Jeden z nich mieści się w pierwszej klatce głównej listwy czasowej (`_root`), zaś drugi w przycisku wykonującym obroty.

W głównej listwie czasowej utworzyłem nową warstwę, przeznaczoną wyłącznie na skrypty. Umieściłem w niej skrypt deklarujący wszystkie potrzebne stałe i umieszczający punkt w początkowym położeniu. Wartość `trans` służy do konwertowania stopni na radiany, przeprowadzanego przed zastosowaniem funkcji trygonometrycznych obiektu *Math*. Następne dwie stałe deklarują położenie środka trójwymiarowego układu współrzędnych na obrazie Flasha. Wreszcie w ostatniej części skryptu deklaruję początkową pozycję punktu i umieszczam go w tej pozycji.

```
// współczynnik konwersji stopni na radiany
trans = Math.PI / 180;
// pozycja środka trójwymiarowego układu współrzędnych
origin_x = 275;
origin_y = 200;
// początkowe położenie punktu
x = 150;
y = 0;
// umieść punkt w początkowym położeniu
point._x = origin_x + x;
point._y = origin_y - y;
```

Akcje zajmujące się obracaniem punktu mieszczą się w skrypcie przycisku, wewnątrz detektora `on(release)`. Po wykryciu kliknięcia przycisku *Obróć* i zwolnienia klawisza myszy, obliczamy sinus i kosinus kąta obrotu (podanego przez użytkownika w polu tekstowym). Ponieważ obiekt *Math* wymaga podawania kąta w radianach, konwertujemy stopnie na radiany bezpośrednio w argumencie funkcji.

```
// znajdź sinus i kosinus kąta obrotu
sin_angle = Math.sin (Number (rotation_angle) * trans);
cos_angle = Math.cos (Number (rotation_angle) * trans);
```

Zwróć uwagę, że wartość zmiennej `rotation_angle` jest wstępnie przetwarzana przez funkcję `Number`. Jest to konieczne, ponieważ wartość pochodząca bezpośrednio z pola tekstowego może być traktowana jako wartość tekstowa, na której nie można wykonywać operacji matematycznych. Dla pewności konwertujemy ją na wartość liczbową.

W następnych kilku wierszach skryptu odbywa się obliczanie nowego położenia obracanego punktu. Nie możemy jednak zmieniać bezpośrednio współrzędnych *x* i *y* punktu za pomocą wyprowadzonych wcześniej równań. Gdybyśmy od razu zaktualizowali współrzędną *x* przed obliczeniem wartości współrzędnej *y*, tożsamości nie zadziałałyby poprawnie. Dlatego tworzymy dwie pomocnicze zmienne, `rotated_x` i `rotated_y`, i dopiero po zakończeniu obliczeń aktualizujemy zmienne `x` i `y`.

```
// oblicz położenie punktu po obrocie,
// używając tożsamości trygonometrycznych dla sumy kątów
rotated_x = x * cos_angle - y * sin_angle;
rotated_y = y * cos_angle + x * sin_angle;
// zaktualizuj położenie punktu
x = rotated_x;
y = rotated_y;
```

Po obliczeniu nowych wartości współrzędnych punktu pozostaje jeszcze umieszczenie punktu w tych współrzędnych. Oczywiście bierzemy przy tym pod uwagę położenie środka trójwymiarowego układu współrzędnych, stosując taki sam kod jak poprzednio:

```
// umieść punkt w nowym położeniu
point._x = origin_x + x;
point._y = origin_y - y;
```

Łącząc wszystkie elementy, otrzymujemy skrypt przycisku, obracający punkt o współrzędnych (*x*, *y*) o kąt `rotation_angle` wokół środka dwuwymiarowego układu współrzędnych.

```
on (release)
{
   // znajdź sinus i kosinus kąta obrotu
   sin_angle = Math.sin (Number (rotation_angle) * trans);
   cos_angle = Math.cos (Number (rotation_angle) * trans);
   // oblicz położenie punktu po obrocie,
   // używając tożsamości trygonometrycznych dla sumy kątów
   rotated_x = x * cos_angle - y * sin_angle;
   rotated_y = y * cos_angle + x * sin_angle;
   // zaktualizuj położenie punktu
   x = rotated_x;
   y = rotated_y;
   // umieść punkt w nowym położeniu
   point._x = origin_x + x;
   point._y = origin_y - y;
}
```

Inny wariant obrotu wokół osi z

W następnym przykładzie korzystamy z takiego samego mechanizmu obrotu. Różnica polega na tym, że kąt obrotu jest ustalany przez użytkownika za pomocą klawiszy strzałek (bez udziału pola tekstowego) i obrót jest wykonywany w każdej klatce filmu (wyeliminowano przycisk *Obróć*). W rezultacie powstaje efekt ciągłego ruchu obrotowego. Przytrzymując klawisz strzałki, możesz płynnie zwiększać lub zmniejszać prędkość ruchu obrotowego. Plik korzysta z techniki wykrywania wciśnięć klawiszy, którą omówiliśmy wcześniej.

Otwórz plik *z_axis_rotation2.swf*, zapisany na płycie CD-ROM.

Przytrzymując klawisze strzałek w lewo lub w prawo, możesz modyfikować kąt, o jaki punkt obraca się w każdej klatce. Zanim zajrzysz do kodu, zastanów się, jak zrealizujesz takie zadanie. W jaki sposób umożliwisz użytkownikowi modyfikowanie kąta obrotu i obracanie punktu o ten zmodyfikowany kąt?

Otwórz plik *z_axis_rotation2.fla*, zapisany na płycie CD-ROM.

Plik ten działa podobnie jak poprzedni przykład — położenie punktu jest obliczane w taki sam sposób. Plik nie zawiera jednak przycisku *Obróć*, pola tekstowego ani dodatkowej warstwy dla skryptów w głównej listwie czasowej; wszystkie akcje są zawarte wewnątrz detektorów zdarzeń klipu filmowego.

Detektor zdarzenia `load` deklaruje kilka niezbędnych wartości — stałą `trans` służącą do konwertowania stopni na radiany, położenie początku trójwymiarowego układu współrzędnych, początkowe położenie punktu oraz początkowy kąt obrotu. Ponadto

deklarujemy przyrost, o jaki będziemy modyfikować kąt obrotu po wykryciu wciśnięcia klawisza. Ponieważ w każdej klatce wykonujemy obrót, czyli modyfikujemy wartości współrzędnych *x* i *y* punktu, nie jest konieczne ciągłe modyfikowanie kąta obrotu w celu poruszania punktu. Tak naprawdę wartość *rotation_angle* określa teraz prędkość obrotową punktu. Gdy zwiększasz lub zmniejszasz tę wartość, zmieniasz prędkość ruchu.

```
onClipEvent (load)
{
   // współczynnik konwersji stopni na radiany
   trans = Math.PI / 180;
   // pozycja środka trójwymiarowego układu współrzędnych
   origin_x = 275;
   origin_y = 200;
   // początkowe położenie punktu
   x = 150;
   y = 0;
   // przyrost kąta obrotu przy wciśnięciu klawisza
   inc = .1;
   // początkowy kąt obrotu
   rotation_angle = 0;
}
```

Kod wewnątrz detektora zdarzenia `enterFrame` jest bardzo podobny do kodu przycisku z poprzedniego przykładu. Przed przystąpieniem do obliczania nowej pozycji, modyfikujemy kąt obrotu. Ponieważ kąt ten jest modyfikowany przez użytkownika za pomocą klawiszy strzałek, korzystamy z takiego samego rozwiązania, jak w jednym z wcześniejszych skryptów tego rozdziału. Po zmodyfikowaniu kąta obrotu obliczamy jego sinus i kosinus, następnie wstawiamy te wartości do równań obrotu i wyznaczamy nowe położenie punktu.

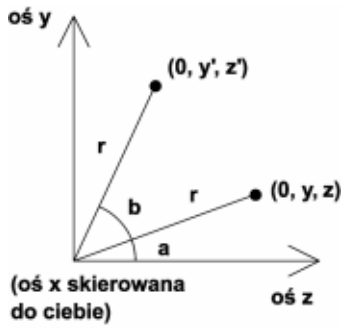
```
onClipEvent (enterFrame)
{
   // zmieniaj kąt obrotu zgodnie z wciśniętymi klawiszami
   rotation_angle += Key.isDown (Key.LEFT) * inc - Key.isDown (Key.RIGHT) * inc;
   // znajdź sinus i kosinus kąta obrotu
   sin_angle = Math.sin (rotation_angle * trans);
   cos_angle = Math.cos (rotation_angle * trans);
   // oblicz położenie punktu po obrocie,
   // używając tożsamości trygonometrycznych dla sumy kątów
   rotated_x = x * cos_angle - y * sin_angle;
   rotated_y = y * cos_angle + x * sin_angle;
   // zaktualizuj położenie punktu
   x = rotated_x;
   y = rotated_y;
   // umieść punkt w nowym położeniu
   this._x = origin_x + x;
   this._y = origin_y - y;
}
```

Obrót wokół pozostałych osi

Obrót wokół pozostałych osi wygląda bardzo podobnie do obrotu wokół osi *z*, zatem nie musimy dla nich powtarzać tak wnikliwego omówienia. Po wyprowadzeniu wszystkich równań zobaczysz, że różnią się one tylko pojedynczymi wartościami.

Wcześniej stwierdziliśmy, że obrót wokół osi *x* nie wpływa na współrzędną *x* punktu. W związku z tym, aby wyobrazić sobie punkt obracający się wokół osi *x*, „przewracamy" trójwymiarowy układ współrzędnych w ten sposób, by oś *x* była skierowana w naszą stronę, prostopadle do płaszczyzny kartki, na której leżą pozostałe osie. Obracany punkt leży teraz gdzieś na płaszczyźnie *yz*, zaś linia łącząca go z początkiem układu współrzędnych tworzy kąt *a* z osią *z*. Zamierzamy obrócić ten punkt wokół osi *x* (lub inaczej wokół początku dwuwymiarowego układu współrzędnych *yz*) o kąt *b*. Sytuacja jest bardzo podobna do omawianej poprzednio, lecz tym razem modyfikujemy współrzędne *y* i *z* (rysunek 2.33).

Rysunek 2.33. *Punkt obracany wokół osi x*



Na podstawie tego rysunku możemy sformułować pierwszy zestaw równań, podobnie jak zrobiliśmy to poprzednio. Ponownie zwróć uwagę, że po obróceniu punktu o kąt *b*, linia łącząca go z początkiem układu współrzędnych tworzy z osią *z* kąt *a*+*b* (rysunek 2.34).

Rysunek 2.34. *Równania opisujące położenie punktu w pierwotnej pozycji oraz po obrocie*

$$x = 0$$
$$y = r \sin a$$
$$z = r \cos a$$

$$x' = 0$$
$$y' = r \sin (a + b)$$
$$z' = r \cos (a + b)$$

Pominiemy pracę związaną z przekształceniami, którą wykonaliśmy już poprzednio. Korzystając z tożsamości trygonometrycznych oraz upraszczając równania, otrzymujesz ostateczną postać równań (rysunek 2.35).

Tym sposobem otrzymałeś równania, za pomocą których możesz obrócić punkt wokół osi *x* w trójwymiarowym układzie współrzędnych. Jeśli znasz początkowe położenie punktu (*x*, *y*, *z*) i chcesz go obrócić wokół osi *x* o kąt *b*, po prostu użyj ostatnich dwóch równań. Również tym razem udało nam się wyeliminować problematyczne wielkości *r* i *a*.

Wyprowadzenie równań opisujących obrót punktu wokół osi *y* wygląda niemal identycznie. Dlatego rezygnuję z geometrycznej reprezentacji problemu i przedstawiam jedynie algebraiczne przekształcenia (rysunek 2.36).

Rysunek 2.35. *Równania opisujące obrót punktu wokół osi* x

$$x = r \cos a$$
$$y = r \sin a$$
$$z = 0$$

$$x' = 0$$
$$y' = r \sin (a + b) = r \sin a \cos b + r \cos a \sin b$$
$$z' = r \cos (a + b) = r \cos a \cos b - r \sin a \sin b$$

$$\Downarrow$$

$$y' = y \cos b - z \sin b$$
$$z' = z \cos b + y \sin b$$

Rysunek 2.36. *Równania opisujące obrót punktu wokół osi y w trójwymiarowym układzie współrzędnych*

$$x = r \cos a$$
$$y = 0$$
$$z = r \sin a$$

$$\Downarrow$$

$$x' = r \cos (a + b)$$
$$y' = 0$$
$$z' = r \sin (a + b)$$

$$\Downarrow$$

$$x' = r \cos (a + b) = r \cos a \cos b - r \sin a \sin b$$
$$y' = 0$$
$$z' = r \sin (a + b) = r \sin a \cos b + r \cos a \sin b$$

$$\Downarrow$$

$$x' = x \cos b - z \sin a$$
$$z' = z \cos b + x \sin a$$

Wyprowadziliśmy równania opisujące obroty składowe wokół wszystkich trzech osi przestrzennego układu współrzędnych. Zbierzmy rezultaty tych wyprowadzeń na rysunku 2.37.

Choć oznaczenia przy zmiennych *x*, *y* i *z* — jeden, dwa lub trzy apostrofy — są stosunkowo drobne, nie wolno ich lekceważyć. Powyższe równania demonstrują zasadę, która ma ogromne znaczenie podczas obracania obiektów względem trzech osi — zawsze przekształcaj punkt będący wynikiem poprzedniego przekształcenia. Oznacza to, że jeśli chcesz obrócić punkt wokół osi *x* i *y*, powinieneś obrócić pierwotny punkt wokół osi *x*, a następnie wynik tego obrotu obrócić wokół osi *y*.

Oto plan działania skryptu, który obraca wiele punktów wokół wszystkich trzech osi (*x*, *y* oraz *z*) układu współrzędnych, odpowiednio o kąty *a*, *b* i *c*:

Rysunek 2.37. *Równania opisujące obroty składowe wokół poszczególnych osi przestrzennego układu współrzędnych*

Dany punkt: (x, y, z)
Obrót punktu wokół osi x: (x, y', z')
Obrót punktu wokół osi y: (x", y′, z")
Obrót punktu wokół osi z: (x"', y"', z")

obrót wokół osi x
y' = y cos b - z sin b
z' = z cos b + y sin b

obrót wokół osi y
x" = x cos b - z' sin b
z" = z′ cos b + x sin b

obrót wokół osi z
x"' = x" cos b - y' sin
y"' = y' cos b + x″ sin

1. Znajdź sinus i kosinus kątów obrotu *a*, *b* i *c*.
2. Rozpocznij pętlę, by obliczyć współrzędne obracanych punktów i umieścić punkty w nowych współrzędnych przestrzennych.
3. Obróć punkt wokół osi *x*.
4. Weź wynik obrotu punktu wokół osi *x* i obróć go wokół osi *y*.
5. Weź wynik obrotu punktu wokół osi *y* i obróć go wokół osi *z*.
6. Oblicz rzut punktu przestrzeni na płaszczyznę ekranu.
7. Zmodyfikuj inne parametry klipu filmowego reprezentującego punkt.

Optymalizacja

Posiadamy wszystkie informacje potrzebne do utworzenia we Flashu kilku spektakularnych efektów. Jednak nie mniej ważne jest, by Flash zdołał wykonać wszystkie potrzebne obliczenia. Bardziej zaawansowane metody optymalizacji obliczeń i redukcji obciążenia procesora nie mają zastosowania we Flashu z dwóch powodów: Flash nie obsługuje optymalizowanych obliczeń, a w środowisku nie ma wystarczającej liczby obiektów, by w pełni korzystać z tych algorytmów. W tym podrozdziale omówimy jednak kilka technik optymalizacji, które są użyteczne we Flashu.

Wyłączanie niewidocznych obiektów

Wcześniej w tym rozdziale zastosowaliśmy prostą metodę optymalizacji obliczeń związanych z grafiką trójwymiarową — jeśli czegoś nie widać, nie renderuj tego. Gdy punkt

znajdował się z tyłu obserwatora, nie wykonywaliśmy dla niego obliczeń, lecz wyłączaliśmy jego widzialność. Możemy rozwinąć tę ideę i wyłączać obiekty, które nie są widoczne w obszarze ekranu.

Obiekt znajduje się poza widzialnym obszarem, jeśli jego współrzędna *x* lub *y* na obrazie jest większa od szerokości lub wysokości obrazu, lub mniejsza od 0. Możemy to sprawdzić za pomocą kilku wyrażeń warunkowych i w zależności od wyniku przypisać właściwości `_visible` odpowiedniego klipu wartość `true` lub `false`.

```
onClipEvent (load)
{
   //szerokość i wysokość obrazu
   stage_width = 550;
   stage_height = 400;
}
onClipEvent (enterFrame)
{
   // sprawdź, czy klip znalazł się poza obrazem
   if ((this._x > stage_width) || (this._x <0))
      {
      //klip znalazł się poza lewą lub prawą krawędzią obrazu
      this._visible = false;
      }
   else if ((this._y > stage_height) || (this._y <0))
      {
      //klip znalazł się poza górną lub dolną krawędzią obrazu
      this._visible = false;
      }
   else
      {
      //klip znajduje się na obrazie
      this._visible = true;
      }
}
```

Pamiętaj, że wszelkie przesunięcia, obroty i rzuty perspektywiczne musisz wykonać przed sprawdzeniem, czy punkt znajduje się na obrazie. Wyłączanie niewidocznych obiektów nie zaoszczędzi ci zatem obliczeń, jednak pozwoli zminimalizować liczbę obiektów jednocześnie wyświetlanych przez Flasha.

Minimalizuj ilość obliczeń

Optymalizacja jest nierozerwalnie związana z techniką programowania. Aby przyspieszyć wykonywanie dowolnych procedur, musisz zminimalizować ilość wykonywanych obliczeń. Oznacza to możliwie rzadkie używanie obiektu *Math*, tylko do tych celów, do których jest absolutnie niezbędny. Oznacza to również upraszczanie wyrażeń, tak aby zawierały jak najmniejszą liczbę operacji mnożenia i dzielenia (które są znacznie bardziej pracochłonne od dodawania i odejmowania).

Obiekt Math

Choć działanie obiektu *Math* nie jest nadzwyczaj powolne, jednak powinieneś się do niego odwoływać nie więcej niż kilka razy, traktując go jako obciążenie dla procesora. Wiele osób popełnia błąd polegający na obliczaniu sinusów i kosinusów bezpośrednio we wzorach obrotów, wewnątrz pętli obracającej wszystkie punkty. Ponieważ w konkretnej klatce wszystkie punkty najprawdopodobniej są obracane o ten sam kąt, sinus i kosinus tego kąta wystarczy obliczyć jeden raz.

Oto kod z komentarzami opisującymi tę strategię:

```
onClipEvent (enterFrame)
{
// zmodyfikuj kąt obrotu lub przyrost przesunięcia
// oblicz sinus i kosinus kąta obrotu
for (var j = 0; j < liczba_punktów; j++)
   {
   // przesuń punkt
   // obróć punkt
   // wykonaj rzut perspektywiczny
   // sprawdź, czy punkt mieści się na obrazie
   // wyrenderuj punkt
   }
}
```

Niektórzy tworzą tablice wartości sinusów i kosinusów, dzięki czemu w ogóle nie muszą korzystać z obiektu *Math*. Choć praktyka ta była popularnie stosowana w przeszłości w językach C++ i innych językach programowania, jednak nie jest potrzebna w przypadku dzisiejszych szybkich komputerów.

Studium 1. Obracanie skupiska punktów

W pierwszej części studium przedstawię przykład, nad którym pracowaliśmy już we wcześniejszej części rozdziału. Tworzy on skupisko losowo rozmieszczonych punktów i oddaje do dyspozycji użytkownika zestaw przycisków umożliwiających ich obracanie. Struktura skryptu i filmu jest jednak całkowicie odmienna, zatem przystąpmy do ich omawiania.

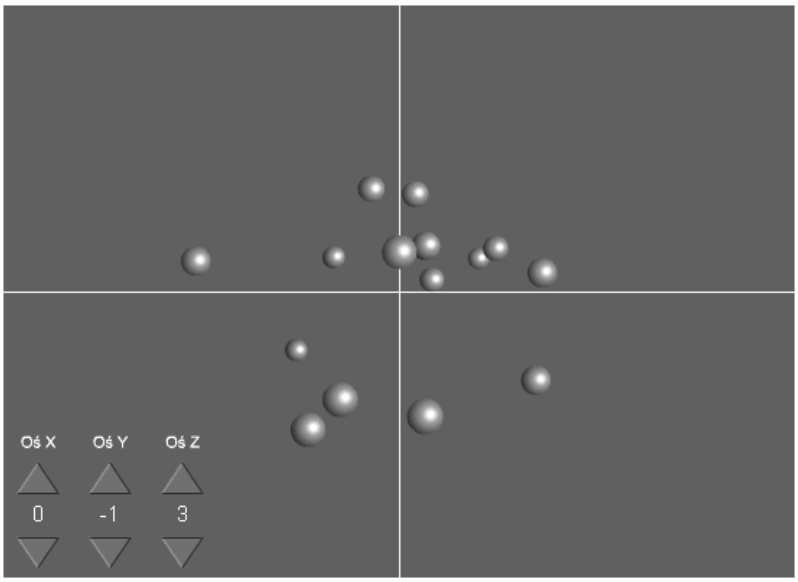
Otwórz plik *demo_one1.swf*, zapisany na płycie CD-ROM.

Film wyświetla składowe kąty obrotu wokół poszczególnych osi i w każdej klatce obraca skupisko punktów o ustawione kąty (rysunek 2.38).

Otwórz plik *demo_one1.fla*, zapisany na płycie CD-ROM.

Układ

Dokument zawiera siedem głównych elementów graficznych. Sześć z nich to przyciski sterujące cząstkowymi kątami obrotu wokół trzech osi. Ostatnia grafika to klip filmowy



Rysunek 2.38. *Za pomocą przycisków ustaw składowe kąty obrotu wokół poszczególnych osi*

reprezentujący punkt. W odróżnieniu od poprzednich przykładów, większość kodu nie mieści się wewnątrz klipu filmowego punktu. Ze względu na nabyte doświadczenie w programowaniu w języku C++, preferuję umieszczanie kodu w jednym miejscu, gdy tylko jest to możliwe. Dlatego utworzyłem oddzielny klip filmowy, któremu nadałem nazwę `dummy` i w którym umieściłem większość akcji.

Zasięg działania zmiennych

Większość kodu mieści się w detektorach zdarzeń klipu filmowego `dummy`. Jedynie trzy zmienne mieszczą się poza nimi — są to kąty obrotu wokół poszczególnych osi. Zmienne te są przechowywane na głównej listwie czasowej, dzięki czemu przyciski mają do nich łatwy dostęp. Skrypt klipu filmowego również korzysta z tych zmiennych podczas obliczania sinusów i kosinusów kątów.

Usuwamy niepotrzebny klip

Klip filmowy o nazwie `point` jest wielokrotnie duplikowany podczas tworzenia skupiska punktów. Chcemy, aby były widoczne jedynie zduplikowane klony, jednak pierwotny klon powinien być niewidoczny, ponieważ mieści się dokładnie w środku trójwymiarowego układu współrzędnych i obrót nie wpływa na jego pozycję. Dlatego w detektorze zdarzenia `load` klipu filmowego `point` umieściłem następujący skrypt, który wyłącza widzialność pierwotnego klonu i pozostawia jedynie klony zduplikowane.

```
onClipEvent (load)
{
   // wyłącz pierwotny klip filmowy
   if (this._name == "point")
```

```
    {
        this._visible = false;
    }
}
```

Inicjalizacja zmiennych

W pierwszej klatce głównej listwy czasowej (_root) inicjalizujemy zmienne rotation_angle_x, rotation_angle_y i rotation_angle_z, reprezentujące kąty obrotu wokół poszczególnych osi. Wartości tych zmiennych są modyfikowane przez użytkownika za pomocą sześciu przycisków umieszczonych na obrazie. Następnie są odczytywane przez klip filmowy dummy. Ponieważ bezpośrednio po uruchomieniu filmu punkty nie powinny być obracane, zmiennym tym nadajemy początkowe wartości 0.

Pozostałe akcje powiązałem bezpośrednio z klipem filmowym dummy. Odczytywanie współrzędnych punktu w poprzednich przykładach nie stanowiło żadnego problemu, ponieważ wszelkie informacje o punkcie mieściły się w nim samym. Tymi parametrami punktów sterujemy z zewnętrznego skryptu, zatem musimy śledzić pozycje wielu punktów. Moglibyśmy utworzyć wiele zmiennych o nazwach x1, x2, y1, y2 i tak dalej dla każdego punktu, lecz takie rozwiązanie byłoby mało wydajne. Zamiast tego tworzymy tablicę obiektów. Każdy element tablicy reprezentuje jeden punkt, zaś poszczególne właściwości punktów stanowią wartości poszczególnych współrzędnych.

W związku z powyższym w detektorze zdarzenia load klipu filmowego dummy tworzymy tablicę, a następnie w pętli umieszczamy obiekt w każdym elemencie tej tablicy. Wreszcie losujemy początkowe wartości współrzędnych punktu. W tej samej pętli duplikujemy również same punkty.

```
onClipEvent (load)
{
    // liczba losowo rozmieszczanych punktów
    num_points = 15;
    // tablica z pozycjami punktów
    point_position = new Array (num_points);
    // powielanie punktów i losowanie początkowych pozycji
    for (var j = 0; j < num_points; j++)
    {
        // duplikacja klipów filmowych
        _parent.point.duplicateMovieClip ("point" + j, j);
        // utwórz obiekt dla współrzędnych x, y i z punktu
        point_position[j] = new Object ();
        // wylosuj początkowe współrzędne punktu
        point_position[j].x = Math.random () * 200 - 100;
        point_position[j].y = Math.random () * 200 - 100;
        point_position[j].z = Math.random () * 200 - 100;
    }
```

Pozostały kod w detektorze zdarzenia load deklaruje zmienne, których używaliśmy już wcześniej — położenie początku trójwymiarowego układu współrzędnych, początkowy rozmiar klipu reprezentującego punkt, zakładaną odległość obserwatora od ekranu oraz wartość współczynnika, służącego do konwersji stopni na radiany.

```
    // położenie początku trójwymiarowego układu współrzędnych
    origin_x = 275;
    origin_y = 200;
    // zakładana odległość obserwatora od ekranu
    D = 300;
    // początkowy rozmiar klipu filmowego
    regular_size = this._width;
    // współczynnik konwersji stopni na radiany
    trans = Math.PI / 180;
}
```

Skrypty przycisków

Skrypty przycisków są w tym przykładzie bardzo proste. Gdy użytkownik kliknie przycisk przedstawiający strzałkę w górę, jego skrypt zwiększy wartość odpowiedniego kąta obrotu; gdy z kolei kliknie przycisk skierowany w dół, jego skrypt zmniejszy wartość odpowiedniego kąta obrotu. Możesz dowolnie dobrać wartość jednorazowej zmiany wartości kąta; ja zdecydowałem się na zastosowanie operatorów inkrementacji ++ i dekrementacji --, które zmieniają wartość zmiennej o 1.

Oto skrypt wykonywany po kliknięciu przycisku skierowanego w górę, odpowiadającego osi *x*.

```
on (release)
{
    // zwiększ kąt obrotu wokół osi x o 1
    rotation_angle_x++;
}
```

Główny skrypt

Główny skrypt filmu mieści się w detektorach zdarzeń klipu filmowego `dummy`. W skrypcie tym są wykonywane obroty i rzuty perspektywiczne punktów. Choć te 50 wierszy kodu nie zawiera żadnych nowości, jednak wiąże się z nim kilka pułapek, na które trzeba uważać. Ponadto możesz zaobserwować zastosowane w nim techniki optymalizacji.

> Wiele osób popełnia błąd polegający na obracaniu obiektów wokół wszystkich osi, nawet gdy nie jest to potrzebne. W wielu przypadkach wystarcza obrót wokół jednej bądź dwóch osi. Obrót wokół większej liczby osi, niż jest to konieczne, jedynie zwiększa ilość obliczeń wykonywanych przez skrypt.

Ponieważ w tym przykładzie nie zajmujemy się żadnymi przesunięciami, możemy od razu przejść do ruchu obrotowego. Na początku są obliczane sinusy i kosinusy kątów obrotu wokół poszczególnych osi. Pamiętaj o konieczności przeliczenia kątów ze stopni na radiany.

```
onClipEvent (enterFrame)
{
    // oblicz sinus i kosinus składowych kątów obrotu
    sin_a = Math.sin (_parent.rotation_angle_x * trans);
    cos_a = Math.cos (_parent.rotation_angle_x * trans);
    sin_b = Math.sin (_parent.rotation_angle_y * trans);
```

```
cos_b = Math.cos (_parent.rotation_angle_y * trans);
sin_c = Math.sin (_parent.rotation_angle_z * trans);
cos_c = Math.cos (_parent.rotation_angle_z * trans);
```

Następnie w pętli odczytujemy po kolei wszystkie elementy tablicy `point_position` i obracamy punkty wokół poszczególnych osi. Zdecydowałem się na użycie pętli `for`, ponieważ liczba elementów w tablicy raczej się nie zmieni, zaś do czytania tablicy potrzebny jest indeks, którym jest licznik pętli `j`.

```
// w pętli obróć i zrzutuj wszystkie punkty
for (var j = 0; j < num_points; j++)
{
// akcje
}
```

Pierwszymi operacjami wykonywanymi w pętli są obroty. Pamiętaj, że najpierw obracamy punkt wokół osi *x*, następnie wynik tego obrotu obracamy wokół osi *y* i wreszcie wynik tego obrotu obracamy wokół osi *z*. Ważne jest, byś zawsze przekształcał wynik poprzedniego przekształcenia i nie modyfikował tablicy `point_position` przed zakończeniem wszystkich przekształceń. W razie niepewności wróć do rysunku 2.37 — nie jest łatwo zapamiętać sześciu równań!

```
// obrót wokół osi x
rx1 = point_position[j].x;
ry1 = point_position[j].y * cos_a - point_position[j].z * sin_a;
rz1 = point_position[j].z * cos_a + point_position[j].y * sin_a;
// obrót wokół osi y
rx2 = rx1 * cos_b - rz1 * sin_b;
ry2 = ry1;
rz2 = rz1 * cos_b + rx1 * sin_b;
// obrót wokół osi z
rx3 = rx2 * cos_c - ry2 * sin_c;
ry3 = ry2 * cos_c + rx2 * sin_c;
rz3 = rz2;
// aktualizacja tablicy współrzędnych
point_position[j].x = rx3;
point_position[j].y = ry3;
point_position[j].z = rz3;
```

W tym miejscu skryptu mamy już wykonane obliczenia związane z obrotem wszystkich współrzędnych punktów wokół osi *x*, *y* i *z*. Teraz pozostaje zrzutowanie punktów na płaszczyznę ekranu i wyświetlenie ich. Rozpoczynamy od obliczenia współczynnika perspektywy, a następnie obliczamy położenie rzutu punktu na ekranie.

```
// współczynnik perspektywy
perspective_ratio = D / (D + rz3);
// rzutowanie punktu przestrzeni na ekran
perspective_x = rx3 * perspective_ratio;
perspective_y = ry3 * perspective_ratio;
```

Być może pogubiłeś się już w tym, do czego potrzebne są nam te wszystkie obliczenia, zatem przypomnijmy. Współczynnik perspektywy jest kluczem do wyznaczenia położenia punktu na obrazie i zmodyfikowania jego skali. Aby obliczyć współczynnik perspektywy, musisz znać współrzędne punktu w przestrzeni. Aby znaleźć aktualne współrzędne punktu,

wykonujesz wszelkie przesunięcia i obroty. Gdy już dotrzesz do współczynnika perspektywy i uporządkowanej pary liczb, po prostu wpisujesz odpowiednie wartości do właściwości klipu filmowego.

```
// umieść punkt w nowym miejscu
_parent["point" + j]._x = origin_x + perspective_x;
_parent["point" + j]._y = origin_y - perspective_y;

// modyfikuj wymiary klipu
_parent["point" + j]._xscale = _parent["point" + j]._yscale = regular_size *
➥perspective_ratio;

// modyfikuj poziom warstwy dla klipu zgodnie z jego współrzędną z
_parent["point" + j].swapDepths (-rz3);
```

Inny wariant sterowania

Sterowanie ruchem obrotowym za pomocą przycisków na obrazie może być mało wygodne. Zamiast tego możemy zdefiniować zestaw klawiszy na klawiaturze, służących do modyfikowania kątów obrotu wokół poszczególnych osi. Przyjmijmy, że klawisze strzałek w górę i w dół posłużą do sterowania kątem obrotu wokół osi *x*, klawisze strzałek w lewo i w prawo będą zmieniały kąt obrotu wokół osi *y*, natomiast za pomocą klawiszy *Ctrl* i *0* (zero na klawiaturze numerycznej) użytkownik może zmieniać kąt obrotu wokół osi *z*.

Otwórz plik *demo_one2.fla* na płycie CD-ROM.

W tym pliku stosujemy te same rozwiązania co poprzednio, jedynie nieznacznie modyfikujemy działanie skryptu z pliku *demo_one1.fla*. Przede wszystkim usuwamy przyciski i obsługujące je skrypty, jak również skrypt w pierwszej klatce głównej listwy czasowej (`_root`). Zmienne kątów obrotu będą teraz przechowywane w klipie filmowym `dummy`. W związku z tym wewnątrz detektora zdarzenia `load` klipu musimy umieścić takie dodatkowe akcje:

```
// początkowe kąty obrotu wokół poszczególnych osi
rotation_angle_x = 0;
rotation_angle_y = 0;
rotation_angle_z = 0;
```

Ponadto w chwili wyświetlenia klipu inicjalizujemy zmienną definiującą przyrost kąta obrotu po wykryciu wciśnięcia klawisza. Wypróbuj różne wartości, by wybrać taką, która ci najbardziej odpowiada.

```
// przyrost kąta obrotu po wykryciu wciśnięcia klawisza
rotation_inc = .5;
```

Inicjalizacja kolejnej zmiennej może się początkowo wydawać pozbawiona sensu. Obiekt *Key* zawiera wiele predefiniowanych stałych (właściwości). Posiadają one nazwy poszczególnych klawiszy i przechowują odpowiadające im kody numeryczne. Na przykład stała `Key.LEFT` zawiera wartość 37 (kod klawisza strzałki w lewo). Za pomocą tych stałych możemy z łatwością sprawdzić, czy został wciśnięty określony klawisz,

korzystając z metody `isDown()`. Stałe obiektu *Key* obejmują wszystkie klawisze używane w naszym przykładzie, oprócz klawisza *0* (zero) na klawiaturze numerycznej. Aby wykryć wciśnięcie tego klawisza, jako argument metody `isDown()` należy podać wartość 96. Jednak zgodnie z zasadami eleganckiego programowania, w kodzie powinniśmy unikać stosowania stałych. Dlatego zdefiniujemy w obiekcie *Key* nową stałą, odpowiadającą klawiszowi *0* na klawiaturze numerycznej.

```
// kod klawisza 0 na klawiaturze numerycznej
Key.NUMPAD_0 = 96;
```

To wszystkie dodatkowe wartości, jakich używamy w tym wariancie. Wewnątrz detektora zdarzenia `load` klipu filmowego musimy dodać akcje zwiększające lub zmniejszające wartości kątów obrotu po wykryciu wciśnięcia odpowiedniego klawisza. Robimy to przed obliczeniem sinusów i kosinusów kątów, by obliczone wartości były aktualne.

```
// modyfikuj wartości kątów obrotu w zależności od wciśniętych klawiszy
rotation_angle_x += Key.isDown (Key.UP) * rotation_inc - Key.isDown (Key.DOWN) *
↳rotation_inc;
rotation_angle_y += Key.isDown (Key.RIGHT) * rotation_inc - Key.isDown (Key.LEFT) *
↳rotation_inc;
rotation_angle_z += Key.isDown (Key.CONTROL) * rotation_inc - Key.isDown
↳(Key.NUMPAD_0) * rotation_inc;
```

Ostatnia modyfikacja dotyczy zastosowania obiektu *Math*. W poprzednim wariancie kąty obrotów były dostępne na głównej listwie czasowej i w skrypcie klipu musieliśmy odnosić się do nich za pomocą ścieżki `_root`. Teraz kąty obrotu są dostępne na tej samej listwie czasowej co skrypt, zatem możemy pominąć ścieżkę adresową.

```
// oblicz sinus i kosinus składowych kątów obrotu
sin_a = Math.sin (rotation_angle_x * trans);
cos_a = Math.cos (rotation_angle_x * trans);
sin_b = Math.sin (rotation_angle_y * trans);
cos_b = Math.cos (rotation_angle_y * trans);
sin_c = Math.sin (rotation_angle_z * trans);
cos_c = Math.cos (rotation_angle_z * trans);
```

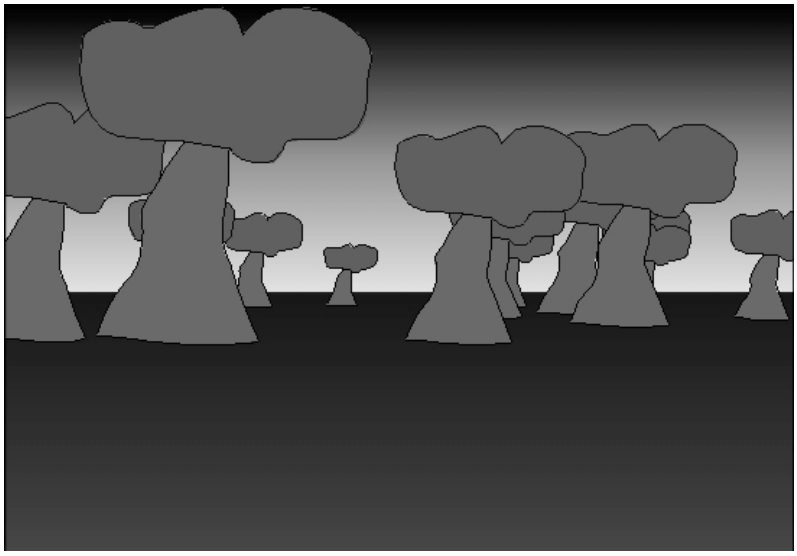
I to wszystkie modyfikacje w tym wariancie przykładu. Obrót wokół poszczególnych osi jest w nim sterowany za pomocą sześciu klawiszy na klawiaturze.

Studium 2. Trójwymiarowy świat

Efekt prezentowany w tym przykładzie może się wydawać skomplikowany, lecz w rzeczywistości łatwo go uzyskać. Zastosujemy całą wiedzę na temat trójwymiarowości, jaką dotąd poznaliśmy; przedstawimy też kilka dodatkowych informacji związanych z tworzeniem trójwymiarowego świata we Flashu.

Otwórz plik *demo_two1.swf* zapisany na płycie CD-ROM.

Za pomocą klawiszy strzałek możesz poruszać się w trójwymiarowym świecie (rysunek 2.39). Zastanów się, jak mógłbyś uzyskać taki efekt, zanim zajrzysz do pliku. Jakiego typu obrót realizujemy? Jakie zachodzą przesunięcia?



Rysunek 2.39. *Trójwymiarowy świat utworzony we Flashu*

Trójwymiarowy świat utworzony we Flashu

Trójwymiarowy świat we Flashu musi być pod każdym względem prosty. Obiekty powinny być statyczne i wyglądać tak samo z każdego miejsca. Można oczywiście użyć obiektu prerenderowanego pod różnymi kątami, tak aby w zależności od położenia w świecie było widoczne inne jego ujęcie. Jednak rzadko kiedy takie rozwiązanie jest warte zachodu, ponieważ znacznie zwiększa rozmiar pliku i obniża wydajność.

W związku z tymi ograniczeniami tworzenie trójwymiarowego świata we Flashu wymaga zastosowania jedynie kilku technik, które tu poznaliśmy. Ponieważ chcemy, aby świat otaczał bohatera z wszystkich stron, jedyny obrót, jaki może on wykonać na płaszczyźnie *xz*, to obrót wokół osi *y*. Z kolei ruch (w sensie przesunięcia) może się odbywać jedynie wzdłuż osi *x* i *z*. Ruch wzdłuż osi *y* umożliwiałby wznoszenie się lub opadanie, co dałoby niepożądany rezultat, ponieważ obiekty są statyczne.

W tym przykładzie powielamy na obrazie rysunek tego samego drzewa, rozmieszczając jego klony w różnych miejscach świata. Klip filmowy drzewa nosi nazwę `objects`. W przypadku takiego obiektu jak drzewo trudno jest wybrać punkt odniesienia, który reprezentowałby położenie drzewa w trójwymiarowym świecie. Aby zachować wrażenie ciągłości świata, musimy wybrać pojedynczy punkt reprezentujący całe drzewo. Najlepiej zdefiniowanym punktem drzewa w świecie jest ten, w którym oba się stykają. Dlatego zdecydowałem się umieścić punkt odniesienia klipu filmowego u podstawy drzewa. Uporządkowana trójka liczb będzie zatem określała położenie podstawy drzewa.

Złudzenie przestrzeni

Za pomocą dwóch prostych przekształceń uzyskamy iluzję obiektów otaczających bohatera. Gdy wyprowadzaliśmy równania określające położenie trójwymiarowego punktu na

dwuwymiarowej płaszczyźnie, zakładaliśmy, że obserwator siedzi w pewnej odległości (*d*) od ekranu. Jednak gdy bohater jest „zanurzony" w świecie, nie ma odległości pomiędzy nim a ekranem. Pierwszym naszym przekształceniem będzie zatem przesunięcie wszystkich punktów świata w stronę obserwatora o odległość *d*. Ponieważ to przekształcenie wpływa jedynie na sposób przedstawiania świata, nie powinno wpływać na położenie punktów w symulowanym trójwymiarowym świecie.

Drugie przekształcenie nadaje bohaterowi pewną wysokość w trójwymiarowym świecie. Ponieważ położenie drzewa jest określone przez położenie jego podstawy, bez dodatkowego przesunięcia głowa obserwatora znajdowałaby się na poziomie gruntu. Jednak jesteśmy przyzwyczajeni, że świat obserwujemy z pewnej wysokości. Aby uzyskać ten efekt, przesuwamy wszystkie punkty świata w dół wzdłuż osi *y* o odległość odpowiadającą wysokości bohatera. To przekształcenie również nie powinno wpływać na rzeczywiste położenia punktów, jedynie na sposób ich prezentacji. Poeksperymentuj z wysokością bohatera, aby dobrać taką, jaka jest dla ciebie zadowalająca.

Drobiazgi związane z budową filmu

Kilka spraw związanych z budową filmu wymaga komentarza. Pierwszą z nich jest rysunek tła. Narysowałem proste zielone tło reprezentujące grunt, stykające się z rysunkiem nieba na środku obrazu. Miejsce, w którym grunt styka się z niebem, nie jest przypadkowe; na linii horyzontu znajduje się punkt zbiegu perspektywy dla wszystkich obiektów świata. Oznacza to, że gdy obiekty oddalają się od obserwatora, zbliżają się do linii horyzontu. Położenie punktu zbiegu jest jednoznacznie określone przez współrzędne *x* i *y* trójwymiarowego układu współrzędnych. Punkt zbiegu zwykle powinien się znajdować na linii horyzontu, zatem linia horyzontu powinna się pokrywać ze współrzędną *y* trójwymiarowego układu współrzędnych.

W tym pliku również użyliśmy techniki polegającej na umieszczeniu większości kodu w niewidzialnym klipie filmowym, odpowiedzialnym za wszystkie najważniejsze obliczenia. W nim są przechowywane wszelkie zmienne oraz wykonywane wszystkie obliczenia związane z trójwymiarowością i rzutowaniem obiektów. Operacje, które są wykonywane tylko jeden raz na początku, mieszczą się w detektorze zdarzenia `load`, natomiast akcje wykonywane w każdej klatce znajdują się wewnątrz detektora zdarzenia `enterFrame`.

Wszystkie drzewa umieszczane w filmie są klonami jednego pierwotnego drzewa. Klip filmowy drzewa zawiera skrypt, który usuwa oryginał z obrazu — jest to jedyny klon tego klipu, który nie będzie wyświetlany. Skrypt odpowiedzialny za usunięcie oryginału znamy już z poprzednich przykładów.

```
onClipEvent (enterFrame)
{
   if (this._name == "objects")
   {
      this._visible = false;
   }
}
```

Inicjalizacja trójwymiarowego świata

Zmienne, obiekty, powielanie klipów filmowych i warunki początkowe są zdefiniowane wewnątrz detektora zdarzenia `load` klipu filmowego `dummy`. Niemal wszystkich inicjalizowanych wartości używaliśmy już wcześniej.

Pierwsze zadanie polega na powieleniu drzew i rozmieszczeniu ich w przypadkowy sposób. Każdy klon drzewa otrzymuje również losową skalę, dzięki czemu uzyskujemy bardziej naturalny wygląd lasu. Na początku definiujemy liczbę klonów drzew, jakie znajdą się w filmie, oraz tablicę, która będzie przechowywała pozycje i skale poszczególnych drzew.

```
onClipEvent (load)
{
   // liczba tworzonych obiektów
   num_objects = 20;
   // tablica przechowująca pozycje obiektów
   object_position = new Array (num_objects);
```

Następnie w pętli klonujemy drzewa. Dla każdego drzewa tworzymy w tablicy `object_position` obiekt, w którym umieścimy jego współrzędne i skalę. Wielkości te będą właściwościami obiektu w tablicy.

```
   // powiel obiekty i wylosuj ich właściwości
   for (var j = 0; j < num_objects; j++)
   {
      // duplikacja obiektu
      _parent.objects.duplicateMovieClip ("objects" + j, j);
      // utwórz w tablicy obiekt, zawierający uporządkowaną trójkę współrzędnych
      ↳i skalę obiektu
      object_position[j] = new Object ();
      // wylosuj początkowe położenie obiektu
      object_position[j].x = Math.random () * 1500 - 750;
      object_position[j].y = 0;
      object_position[j].z = Math.random () * 1500 - 750;
      // wylosuj początkową skalę obiektu
      object_position[j].regular_size = Math.random () * 50 + 50;
   }
```

Pozostałe inicjalizowane wielkości są stałymi związanymi z tworzonym światem. Podobnie jak we wcześniejszych demonstracjach musimy określić położenie trójwymiarowego układu współrzędnych, odległość obserwatora od ekranu oraz stałą potrzebną przy konwersji stopni na radiany.

```
   // położenie trójwymiarowego układu współrzędnych
   origin_x = 275;
   origin_y = 200;
   // zakładana odległość obserwatora od ekranu
   D = 500;
   // współczynnik konwersji stopni na radiany
   trans = Math.PI / 180;
```

Pojawia się też nowa zmienna, określająca wysokość bohatera. Poeksperymentuj z tą wartością, by sprawdzić, jaka ci najbardziej odpowiada. Możesz nawet umożliwić użytkownikowi modyfikowanie tej wielkości. Na przykład, gdy użytkownik trzyma wciśniętą spację, wartość ta może się zmniejszać, co da efekt przykucnięcia.

```
    // wysokość bohatera wędrującego po trójwymiarowym świecie
    player_height = 30;
```

Następne trzy wielkości są związane z przesunięciem. Pierwsza z nich stanowi przyrost, o jaki jest zwiększane lub zmniejszane przesunięcie w przypadku wykrycia wciśnięcia odpowiedniego klawisza. Zwróć uwagę, że tym razem przyrost ten nie decyduje o prędkości ruchu, lecz o jednokrokowym przesunięciu, które ustaje po zwolnieniu klawisza. Dwie pozostałe wielkości określają początkowe przesunięcie wzdłuż osi *x* i *z*.

```
    // przyrost używany przy przesunięciach wzdłuż osi x i z
    translation_inc = 10;
    // początkowe przesunięcie
    translation_x = 0;
    translation_z = 0;
```

Podobną inicjalizację przeprowadzamy dla ruchu obrotowego. Pierwsza wartość określa przyrost kąta obrotu w przypadku wykrycia wciśnięcia odpowiedniego klawisza. Druga z kolei określa początkowy obrót wokół osi *y*.

```
    // przyrost kąta obrotu wokół osi y
    rotation_inc = 2;
    // początkowy kąt obrotu wokół osi y
    rotation_angle_y = 0;
```

Wreszcie, wewnątrz obiektu *Key* definiujemy stałą odpowiadającą kodowi klawisza *0* (zero) na klawiaturze numerycznej. Za chwilę omówimy klawisze sterujące ruchem i obrotem bohatera.

```
    // kod klawisza 0 (zero) na klawiaturze numerycznej
    Key.NUMPAD_0 = 96;
}
```

Klawisze sterujące

Dobór odpowiednich klawiszy sterujących umożliwi użytkownikowi wygodne poruszanie się w świecie. Musimy umożliwić mu sterowanie obrotami wokół osi *y* oraz przemieszczanie się wzdłuż osi *z* (w przód i w tył) oraz osi *x* (na boki). Dobór klawiszy sterujących obrotem wokół osi *y* oraz ruchem wzdłuż osi *z* jest oczywisty — najbardziej odpowiednie do tego celu są klawisze strzałek. Mniej oczywiste jest sterowanie ruchem na boki; zdecydowałem się na wybranie klawiszy *Ctrl* i *0* na klawiaturze numerycznej.

Główna część skryptu

Pozostała, główna część skryptu mieści się w detektorze zdarzenia `enterFrame` klipu filmowego `dummy`. W tych 70 czy 80 wierszach kodu korzystamy z wszystkich omówionych do tej pory technik, łącznie z optymalizacją.

Zanim przystąpimy do przesuwania, obracania i rzutowania punktów, musimy zmodyfikować aktualny kąt obrotu oraz przesunięcia, zgodnie z klawiszami wciśniętymi przez użytkownika. Stosujemy w tym celu technikę, którą poznaliśmy już wcześniej.

```
onClipEvent (enterFrame)
{
   // modyfikuj przesunięcie x i z w zależności od wykrytego wciśnięcia klawisza
   translation_x = Key.isDown (Key.CONTROL) * translation_inc - Key.isDown
   ↳(Key.NUMPAD_0) * translation_inc;
   translation_z = Key.isDown (Key.DOWN) * translation_inc - Key.isDown (Key.UP) *
   ↳translation_inc;
   // modyfikuj kąt obrotu wokół osi y w zależności od wykrytego wciśnięcia klawisza
   rotation_angle_y = Key.isDown (Key.RIGHT) * rotation_inc - Key.isDown (Key.LEFT) *
   ↳rotation_inc;
```

Gdy znamy już aktualny kąt obrotu, musimy obliczyć jego sinus i kosinus na użytek równań obrotu. Pamiętaj, że wartości te wystarczy obliczyć jeden raz przed skorzystaniem z równań obrotu. Nie obliczaj sinusa i kosinusa kąta obrotu wewnątrz pętli `for`, zajmującej się obracaniem i rzutowaniem poszczególnych punktów.

```
// sinus i kosinus kąta obrotu
sin_y = Math.sin (rotation_angle_y * trans);
cos_y = Math.cos (rotation_angle_y * trans);
```

Po zmodyfikowaniu kątów obrotu i przesunięć oraz pomocniczym obliczeniu sinusa i kosinusa, w pętli obliczamy położenie poszczególnych punktów w przestrzeni, poddając je tym przekształceniom. Do tego celu najlepiej nadaje się pętla `for`.

```
for (var j = 0; j < num_objects; j++)
{
   //akcje związane z przekształceniami
}
```

Wnętrze pętli `for` stanowi najważniejszą część skryptu. Przed rzutowaniem punktu i sprawdzeniem, czy mieści się na ekranie, musimy poddać go przesunięciu i obrotowi. Rozpoczynamy od przesunięcia punktu i tymczasowy rezultat tego przekształcenia umieszczamy w tymczasowych zmiennych; tablica `object_position` pozostaje na razie niezmieniona. Następnie przesunięty punkt obracamy wokół osi *y*. Po wykonaniu tego przekształcenia umieszczamy nowe wartości w tablicy `object_position`.

```
// przesuń obiekt wzdłuż osi x i z
tx = object_position[j].x + translation_x;
ty = object_position[j].y;
tz = object_position[j].z + translation_z;

// obróć obiekt wokół osi y
rx = tx * cos_y - tz * sin_y;
ry = ty;
rz = tz * cos_y + tx * sin_y;

// aktualizuj współrzędne obiektu w tablicy
object_position[j].x = rx;
object_position[j].y = ry;
object_position[j].z = rz;
```

W poprzednich przykładach w tym miejscu przystępowaliśmy już do rzutowania punktu. Symulując trójwymiarowy świat, musimy jednak pamiętać, by dokładnie otoczyć nim bohatera. Stąd wynikają dwa dodatkowe przekształceniach, o których wspomnieliśmy wcześniej. Są to przesunięcia, które dotyczą jedynie zmiennych `ry` i `rz`. Położenie

obiektu określone w tablicy `object_position` pozostaje niezmienione, ponieważ zawiera rzeczywiste pozycje obiektów w przestrzeni, które już obliczyliśmy. Wykonywane tu przesunięcia dotyczą jedynie rzutowania obiektów i wyświetlenia ich na ekranie.

```
// przesunięcia zwiększające wiarygodność świata
// nie dotyczą rzeczywistych pozycji punktów
// przesuń przesunięty i obrócony punkt w dół wzdłuż osi y, gdyż bohater ma
↳pewną wysokość
ry -= player_height;
// przesuń przesunięty i obrócony punkt wzdłuż osi z, aby uzyskać perspektywę
↳pierwszej osoby
rz -= D;
```

Tym sposobem uzyskujemy ostateczne współrzędne punktów po przekształceniach. Teraz pozostaje nam zrzutowanie ich na płaszczyznę ekranu.

Podczas rzutowania staramy się wykonać jak najmniej operacji. Z jednej z takich oszczędności korzystaliśmy już wcześniej — sprawdzaliśmy, czy obiekt znajduje się z tyłu obserwatora. Jeśli aktualna współrzędna *z* rzutowanego punktu jest mniejsza niż ujemna wartość odległości obserwatora od ekranu, nie musimy rzutować ani wyświetlać punktu.

```
// sprawdź, czy obiekt znalazł się z tyłu widza
if (rz < -D)
{
   // obiekt z tyłu widza, wyłącz widzialność
   _parent["objects" + j]._visible = false;
}
else
{
// inne akcje
}
```

Nawet jeśli obiekt nie znajduje się z tyłu obserwatora, nie jest powiedziane, że znajdzie się na ekranie. Jeśli obiekt mieści się poza ekranem, nadal nie musimy go wyświetlać. Żeby to jednak stwierdzić, musimy zrzutować go na ekran, czyli poznać jego współrzędne i skalę. Pamiętaj, że te operacje mieszczą się wewnątrz klauzuli `else` wyrażenia warunkowego i są wykonywane tylko wówczas, gdy obiekt znajduje się z przodu obserwatora.

```
   // obiekt nie jest z tyłu widza, lecz może być poza ekranem,
   //więc nie włączaj jeszcze widzialności
   // współczynnik perspektywy
   perspective_ratio = D / (D + rz);
   // oblicz położenie punktu na obrazie
   perspective_x = origin_x + rx * perspective_ratio;
   perspective_y = origin_y - ry * perspective_ratio;
   // skala perspektywiczna obiektu
   perspective_scale = object_position[j].regular_size * perspective_ratio;
```

Po obliczeniu współrzędnych zrzutowanego obiektu możemy sprawdzić, czy znajduje się on na obrazie. Obiekt znajduje się poza lewą krawędzią ekranu, jeśli jego prawa krawędź ma współrzędną *x* mniejszą od zera; z kolei gdy lewa krawędź obiektu ma współrzędną *x* większą niż szerokość ekranu, oznacza to, że znajduje się on w całości poza prawą krawędzią obrazu. Bierzemy zatem współrzędną *x* obiektu, dodajemy lub odejmujemy jego szerokość i sprawdzamy, czy znalazł się za którąś z bocznych krawędzi. W zależności od wyniku tego testu decydujemy, czy włączyć, czy też wyłączyć jego widzialność.

```
            // sprawdź, czy obiekt znajduje się poza ekranem
            if (((perspective_x + perspective_scale) < 0) || ((perspective_x -
            ↳perspective_scale) > origin_x*2))
            {
               // obiekt poza ekranem, wyłącz widzialność
               _parent["objects" + j]._visible = false;
            }
            else
            {
               // pozostałe akcje
            }
```

Wszelkie akcje związane z aktualizacją położenia obiektu mieszczą się wewnątrz klauzuli `else` tego wyrażenia warunkowego. Przede wszystkim włączamy widzialność (`_visible`) obiektu, ponieważ znajduje się on na ekranie. Następnie przepisujemy obliczone wartości współrzędnych i skali do jego odpowiednich właściwości.

```
               // obiekt na ekranie, włącz widzialność
               _parent["objects" + j]._visible = true;
               // aktualizuj współrzędne klipu na obrazie
               _parent["objects" + j]._x = perspective_x;
               _parent["objects" + j]._y = perspective_y;
               // aktualizuj skalę klipu
               _parent["objects" + j]._xscale = _parent["objects" + j]._yscale =
               ↳perspective_scale;
               // aktualizuj poziom klipu zgodnie ze współrzędną z
               _parent["objects" + j].swapDepths (-rz);
            }
         }
      }
   }
```

Domykamy wszystkie nawiasy klamrowe, kończąc skrypt i tworzenie świata. Długość skryptu nie przekroczyła 150 wierszy, zaś efekt trójwymiarowego świata utworzonego w całości we Flashu jest fascynujący.

Studium 3. Trójwymiarowe menu

Ostatni z prezentowanych przeze mnie przykładów to trójwymiarowe menu z przyciskami otwierającymi adresy URL w nowych oknach przeglądarki. Projekt ten może również posłużyć jako galeria fotografii, prac lub inna prezentacja.

Otwórz plik *demo_three1.swf*, zapisany na płycie CD-ROM.

W tym przykładzie losowo rozmieszczamy w trójwymiarowej przestrzeni kilka napisów. Użytkownik może poruszać się w przestrzeni i wybierać napisy, które są jednocześnie przyciskami. Nie obracamy całego środowiska, ponieważ nie jesteśmy w stanie przedstawiać elementów menu pod różnymi kątami (rysunek 2.40).

Otwórz plik *demo_three1.fla*, zapisany na płycie CD-ROM.

Rysunek 2.40. *Ruchome elementy menu rozrzucone w przestrzeni*

Rozpoczynamy demonstrację

Ta demonstracja zawiera bardzo małą liczbę komponentów, zaś w jej mechanizmach nie znajdziesz żadnych sztuczek. Aby podkreślić perspektywiczny charakter demonstracji, możesz utworzyć rysunek tła, obejmujący całą powierzchnię obrazu. W takim przypadku pamiętaj o tym, by punkt zbiegu perspektywy na rysunku tła pokrywał się z punktem zbiegu perspektywy napisów (szczególnie ważna jest współrzędna *y* punktu zbiegu, która pokrywa się z linią horyzontu). Utwórz również klip filmowy (o nazwie `menu_item`), który będzie zawierał elementy menu. W klipie tym umieściłem długie dynamiczne pole tekstowe (takie, by mieściło stosunkowo długie nazwy elementów menu) i powiązałem je ze zmienną. Ponadto pod polem tekstowym utworzyłem niewidzialny przycisk, który po kliknięciu otwiera wybrany adres URL w nowym oknie przeglądarki. Jak zwykle, oddzielny klip filmowy `dummy` zawiera najważniejsze części kodu.

Skrypty

Jedynie w trzech miejscach tego pliku znajdują się skrypty. Pierwszym jest przycisk wewnątrz klipu filmowego `menu_item`, drugim są detektory zdarzeń tego klipu, zaś trzecim są detektory zdarzeń klipu filmowego `dummy`. Najważniejszy skrypt mieści się w trzecim z wymienionych miejsc; pozostałe dwa zawierają po kilka wierszy kodu.

Ponieważ klip filmowy `menu_item` będzie wielokrotnie klonowany, umieściłem w nim krótki skrypt, który wyłącza pierwotny klon tego klipu (ten, z którego będą duplikowane wszystkie następne). Ten pierwszy klon nie ma brać udziału w prezentacji, jedynie dać początek pozostałym klonom.

```
onClipEvent (load)
{
   // wyłącz pierwotny element menu
   if (this._name == "menu_item")
   {
      this._visible = false;
   }
}
```

Główna część skryptu w detektorach zdarzeń klipu `dummy` różni się nieco od kodu, z którym pracowaliśmy do tej pory. Rozpoczynamy od zadeklarowania liczby elementów menu, które pojawią się na obrazie. Dla każdego z nich musimy zapamiętać takie informacje jak wyświetlana nazwa, związany z nią adres URL oraz współrzędne przestrzenne. Aby uporządkować te informacje, tworzymy tablicę zawierającą oddzielny obiekt dla każdego elementu menu. Każdy z tych obiektów będzie zawierał wszystkie informacje o jednym elemencie menu.

```
onClipEvent (load)
{
   // liczba elementów menu
   num_menu_items = 10;
   // tablica z informacjami o poszczególnych elementach menu
   menu_item_info = new Array (num_menu_items);
   // utwórz obiekt dla każdego elementu w tablicy
   for (var j = 0; j < num_menu_items; j++)
   {
      menu_item_info[j] = new Object ();
   }
```

Następnie inicjalizujemy wyświetlane nazwy elementów menu oraz związane z nimi adresy URL. Możesz je wpisać bezpośrednio we Flashu lub wczytać z zewnętrznego pliku tekstowego. Bez względu na sposób ich określenia, upewnij się, że skrypt przepisze wszystkie informacje do tablicy `menu_item_info`, z której skorzystamy w dalszej części. W tym przykładzie zdefiniowałem dziesięć dowolnych nazw elementów menu i odpowiadające im adresy URL.

```
   // nazwy elementów menu
   menu_item_info[0].name = "Home";
   menu_item_info[1].name = "News";
   menu_item_info[2].name = "Portfolio";
   menu_item_info[3].name = "Locations";
   menu_item_info[4].name = "Staff";
   menu_item_info[5].name = "History";
   menu_item_info[6].name = "Awards";
   menu_item_info[7].name = "Business";
   menu_item_info[8].name = "About";
   menu_item_info[9].name = "Contact";
   // adresy url związane z poszczególnymi elementami menu
   menu_item_info[0].url = "http://www.home.com";
   menu_item_info[1].url = "http://www.news.com";
   menu_item_info[2].url = "http://www.portfolio.com";
   menu_item_info[3].url = "http://www.locations.com";
   menu_item_info[4].url = "http://www.staff.com";
   menu_item_info[5].url = "http://www.history.com";
   menu_item_info[6].url = "http://www.awards.com";
```

```
menu_item_info[7].url = "http://www.business.com";
menu_item_info[8].url = "http://www.about.com";
menu_item_info[9].url = "http://www.contact.com";
```

Po ustaleniu tych informacji musimy powielić element menu i ustalić warunki początkowe. Nazwa każdego elementu i odpowiadający mu adres URL są przesyłane do klonu klipu filmowego, gdzie mogą być następnie wyświetlone i użyte w przycisku.

Przestrzenne współrzędne elementów menu są inicjalizowane w specjalny sposób. Tworzony przez nas efekt wymaga, by elementy menu były rozrzucone przed obserwatorem. Oznacza to, że elementy należy rozmieścić jedynie wzdłuż osi *x* i *z*. Dodałem również niewielki czynnik losowy dla współrzędnej *y*, by zwiększyć wrażenie przestrzeni.

```
// duplikacja elementów menu i losowe rozmieszczanie
for (var j = 0; j < num_menu_items; j++)
{
   // duplikacja elementu menu
   _parent.menu_item.duplicateMovieClip ("menu_item" + j, j)
   // umieść nazwę menu w polu tekstowym wewnątrz klipu filmowego
   _parent["menu_item" + j].field = menu_item_info[j].name;
   // prześlij adres url do klipu filmowego, by wywołać go w chwili kliknięcia
   ↳przycisku
   _parent["menu_item" + j].url = menu_item_info[j].url;
   // wylosuj położenie klipu filmowego
   menu_item_info[j].x = Math.random () * 1000 - 500;
   menu_item_info[j].y = Math.random () * 100 - 50;
   menu_item_info[j].z = Math.random () * 500;
}
```

Pozostały jeszcze do zainicjalizowania wielkości wpływające na zmianę położenia elementów. Ponadto definiujemy pierwotną skalę elementów menu. Bezwzględna wielkość wszystkich napisów (czyli ich wielkość po odrzuceniu perspektywy) powinna być taka sama; dzięki temu nie będziesz musiał śledzić skali poszczególnych napisów w tablicy `menu_item_info`.

```
// położenie początku trójwymiarowego układu współrzędnych
origin_x = 275;
origin_y = 200;
// zakładana odległość obserwatora od ekranu
D = 300;
// współczynnik konwersji stopni na radiany
trans = Math.PI / 180;
// przyrost przesunięcia na osiach x i z po wciśnięciu odpowiedniego klawisza
translation_inc = 10;
// początkowe przesunięcie wzdłuż osi x i z
translation_x = 0;
translation_z = 0;
// początkowy rozmiar elementu menu
menu_item_size = 200;
}
```

Pozostałą część skryptu związanego z klipem `dummy` mieści się w detektorze zdarzenia `enterFrame`. Kod ten zajmuje się przesunięciami elementów w przestrzeni i rzutowaniem ich na płaszczyznę obrazu. W tym przykładzie nie wykonujemy żadnych obrotów, nie ma więc potrzeby śledzenia kątów obrotu ani obliczania ich sinusów i kosinusów.

Rozpoczynamy od zmodyfikowania przesunięć wzdłuż osi *x* i *z*, zgodnie w wciśniętymi klawiszami sterującymi. (Możesz również pozwolić użytkownikom przesuwać napisy wzdłuż osi *y*, jeśli nie uważasz, że zbytnio skomplikuje to nawigację).

```
onClipEvent (enterFrame)
{
   // modyfikuj przesunięcie wzdłuż osi x i z zgodnie z wciśniętymi klawiszami
   translation_x = Key.isDown (Key.LEFT) * translation_inc - Key.isDown (Key.RIGHT) *
   ↳translation_inc;
   translation_z = Key.isDown (Key.DOWN) * translation_inc - Key.isDown (Key.UP) *
   ↳translation_inc;
```

Po odczytaniu informacji związanych z przesunięciem punktów, przechodzimy do pętli, w której przesuwamy poszczególne punkty i rzutujemy je na płaszczyznę obrazu. Położenia przesuniętych punktów umieszczamy najpierw w dwóch pomocniczych zmiennych, `tx` i `tz`, a dopiero potem przepisujemy je do tablicy `menu_item_info`.

```
   // w pętli przesuń i zrzutuj wszystkie punkty
   for (var j = 0; j < num_menu_items; j++)
   {
      // przesuń obiekt wzdłuż osi x i z
      tx = menu_item_info[j].x + translation_x;
      tz = menu_item_info[j].z + translation_z;
      // aktualizuj dane w tablicy
      menu_item_info[j].x = tx;
      menu_item_info[j].z = tz;
```

Po przesunięciu punktów sprawdzasz pierwszy warunek widzialności. Jeśli punkt znajduje się z tyłu obserwatora, nie ma potrzeby rzutowania go i wyświetlania. Jeśli jednak znajduje się z przodu obserwatora, musimy dodatkowo sprawdzić, czy znajduje się na obrazie, czy poza nim.

```
      // sprawdź, czy element menu leży z tyłu obserwatora
      if (tz < -D)
      {
         // jeśli tak, wyłącz widzialność elementu
         _parent["menu_item" + j]._visible = false;
      }
      else
      {
         //pozostałe akcje
      }
```

Wewnątrz klauzuli `else` skrypt oblicza położenie i skalę zrzutowanego elementu. Posiadając te informacje, możesz sprawdzić, czy element menu mieści się na obrazie. Ponieważ współrzędna *y* elementu nigdy się nie zmienia, podczas rzutowania współrzędnej *y* możemy skorzystać bezpośrednio z informacji zawartej w tablicy `menu_item_info`.

```
         // element nie znajduje się z tyłu obserwatora, lecz nie znaczy to jeszcze,
         // że znajduje się na obrazie,
         // zatem nie włączaj jeszcze widzialności
         // współczynnik perspektywy
         perspective_ratio = D / (D + tz);
         // oblicz położenie zrzutowanego punktu na obrazie
         perspective_x = origin_x + tx * perspective_ratio;
         perspective_y = origin_y - menu_item_info[j].y * perspective_ratio;
         // oblicz skalę elementu
         perspective_scale = menu_item_size * perspective_ratio;
```

Sprawdzamy, czy element znajduje się na obrazie Flasha, stosując tę samą technikę co poprzednio. Jeśli prawa krawędź elementu jest wysunięta poza lewą krawędź obrazu, bądź też lewa krawędź elementu jest wysunięta poza prawą krawędź obrazu, możemy być pewni, że obiekt znajduje się poza obrazem.

```
// sprawdź, czy obiekt znajduje się poza krawędzią ekranu
if (((perspective_x + perspective_scale) < 0) || ((perspective_x -
↳perspective_scale) > origin_x*2))
{
   // obiekt poza ekranem, wyłącz widzialność
   _parent["menu_item" + j]._visible = false;
}
else
{
   //pozostałe akcje
}
```

Skoro dotarliśmy do tego miejsca struktury decyzyjnej, oznacza to, że element menu znajduje się gdzieś na obrazie. Musimy zatem włączyć jego widzialność (właściwości `_visible` przypisać wartość `true`) oraz zaktualizować pozycję i inne właściwości.

```
            // obiekt znajduje się na ekranie, włącz jego widzialność
            _parent["menu_item" + j]._visible = true;

            // aktualizuj położenie klipu na obrazie
            _parent["menu_item" + j]._x = perspective_x;
            _parent["menu_item" + j]._y = perspective_y;

            // aktualizuj skalę klipu
            _parent["menu_item" + j]._xscale = _parent["menu_item" + j]._yscale =
            ↳perspective_scale;

            // ustal poziom warstwy dla klipu zgodnie ze współrzędną z
            _parent["menu_item" + j].swapDepths (-tz);
         }
      }
   }
}
```

Po zamknięciu wszystkich nawiasów klamrowych możesz przystąpić do testowania przykładu.

Inny wariant

Istnieje wiele metod udoskonalenia interaktywności w prezentowanym menu. W tym wariancie wprowadzimy nawigację za pomocą kursora myszy.

Otwórz plik *demo_three2.swf*, zapisany na płycie CD-ROM.

Przesuwając kursor myszy w górę i w dół, użytkownik może przesuwać się w tył i w przód (a raczej w głąb) świata; przesuwając kursor w prawo i w lewo, może przesuwać się na boki (rysunek 2.41).

Rysunek 2.41. *Przesuwając kursor myszy, możesz poruszać się w trójwymiarowym świecie*

Otwórz plik *demo_three2.fla*, zapisany na płycie CD-ROM.

Mechanizmy, jakie utworzyliśmy do tej pory, można łatwo zaadaptować do tego wariantu. Musimy jedynie zmodyfikować kilka wierszy kodu, obliczających przyrosty przesunięć, które do tej pory robiły to na podstawie wykrytych wciśnięć klawiszy. Teraz przyrosty powinny być obliczane na podstawie położenia kursora myszy.

Bardzo ważne jest, by nawigacja była intuicyjna. Stwierdziłem, że najlepszym rozwiązaniem będzie przesuwanie świata wzdłuż osi *x* w kierunku przeciwnym do położenia kursora. Gdy zatem kursor znajduje się po lewej stronie środka układu współrzędnych, przyrost przesunięcia w poziomie (wzdłuż osi *x*) powinien być dodatni, by napisy przesuwały się w prawo. Z kolei gdy kursor znajduje się po prawej stronie obrazu, przyrost powinien być ujemny, by uzyskać ruch w lewo. Takie zachowanie realizujemy za pomocą jednego prostego wiersza:

```
translation_x = origin_x - _parent._xmouse;
```

Gdy kursor znajduje się nad środkiem układu współrzędnych, przyrost przesunięcia w głąb (wzdłuż osi *z*) powinien być ujemny, tak aby napisy przybliżały się do obserwatora. Napisy powinny się oddalać od obserwatora, gdy kursor znajduje się poniżej środka układu współrzędnych, co uzyskujemy przez dobranie dodatniego przyrostu przesunięcia wzdłuż osi *z*. Poniższy wiersz realizuje pożądane zachowanie:

```
translation_z = _parent._ymouse - origin_y;
```

Gdy zastąpisz poprzednie wyrażenia powyższym kodem, zauważysz, że elementy menu poruszają się zbyt szybko. Przyrosty mają zbyt duże wartości, ponieważ położenie kursora na obrazie jest wyrażane w pikselach. Aby elementy menu poruszały się wolniej, musimy podzielić powyższe różnice przez pewien współczynnik. Jego wartość zależy od tego, czy chcesz uzyskać sterowanie energiczne, czy bardziej spokojne. Poeksperymentuj z tym

współczynnikiem, by znaleźć wartość, która najbardziej ci odpowiada. Aby unikać wartości numerycznych w kodzie, w detektorze zdarzenia `load` definiuję parametr `trans_regulate`, którego zadaniem jest właśnie regulacja prędkości sterowania.

```
// współczynnik regulujący przyrost przesunięcia na podstawie położenia kursora myszy
trans_regulate = 20;
```

Gdy mamy zdefiniowany ten parametr, możemy zmodyfikować oba wiersze obliczające przesunięcia.

```
// modyfikuj przesunięcie wzdłuż osi x i z zgodnie z położeniem kursora myszy
translation_x = (origin_x - _parent._xmouse) / trans_regulate;
translation_z = (_parent._ymouse - origin_y) / trans_regulate;
```

I to wszystko!

Otwórz plik *demo_three3.swf*, zapisany na płycie CD-ROM.

Oto kolejny wariant — jeden z wielu wartych zbadania. Ułożyliśmy w nim wszystkie elementy menu wzdłuż osi *z* i umożliwiliśmy użytkownikowi przesuwanie się pomiędzy nimi. Klawisze strzałek w górę i w dół sterują ruchem wzdłuż osi *z*, zaś kursor myszy pozwala na wykonywanie nieznacznych przesunięć wzdłuż osi *x* i *y* (rysunek 2.42).

Rysunek 2.42. *Możesz tworzyć przeróżne warianty sterowania*

Dalsze przykłady

Na płycie CD-ROM umieściłem jeszcze kilka przykładów zastosowania programowanej grafiki trójwymiarowej we Flashu. Jedna z demonstracji to obracające się trójwymiarowe menu z prostymi elementami nawigacyjnymi witryny. W innym pliku znajdziesz generator terenu z polem sił oraz kulką poruszającą się w tym polu. Mam nadzieję, że badanie tych plików sprawi ci tyle radości, ile mnie sprawiło ich tworzenie. Pliki z tymi przykładami znajdziesz w katalogu *rozdzial_2/_Dalsze_przyklady*.

Podsumowanie

Gratulacje — nauczyłeś się tworzyć nowe, fascynujące efekty. Bez wątpienia pojawią się u ciebie pytania i problemy związane z tworzeniem podobnych efektów. Być może zechcesz je omówić z innymi członkami społeczności flashowych. Jeśli potrafisz posługiwać się językiem angielskim, jednym z dobrych miejsc dla takich dyskusji jest witryna Were-Here (*www.were-here.com*). Na forum matematycznym tej witryny możesz podyskutować z wiodącymi twórcami prezentacji flashowych; są oni znani z tego, że przedyskutowali już niemal wszystkie aspekty tworzenia grafiki trójwymiarowej.

Renderowane obiekty trójwymiarowe

W drugiej części tego rozdziału poznasz magię renderowanych obiektów trójwymiarowych. Poznasz możliwości, jakie daje połączenie technik Flasha z realizmem grafiki trójwymiarowej. Przedstawione tu informacje powinny zainspirować cię do tworzenia w przyszłości własnych, znakomitych projektów tego typu.

Oprogramowanie niezależnych producentów

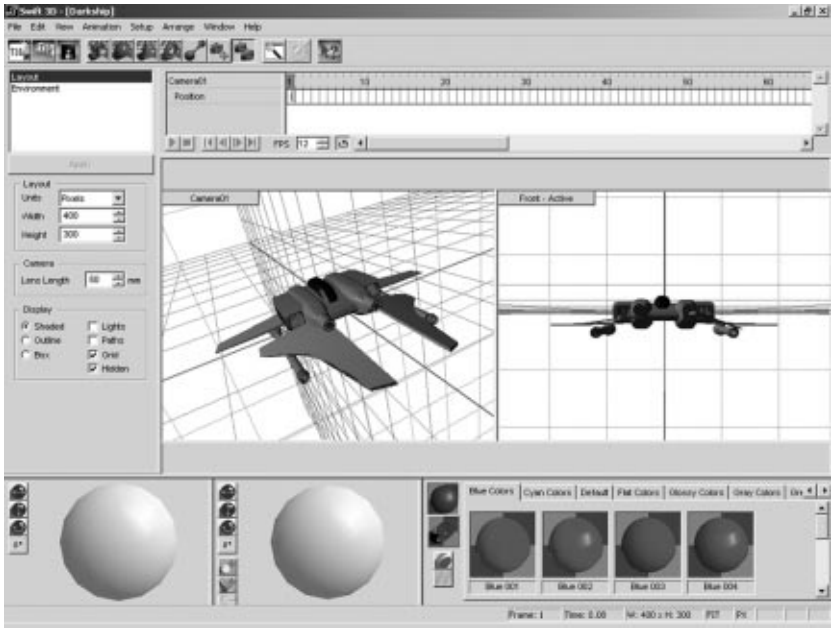
Na rynku istnieje wiele aplikacji do tworzenia i renderowania grafiki trójwymiarowej. Dwa programy szczególne nadają się do tworzenia obiektów trójwymiarowych na potrzeby Flasha. Pierwszy z nich nosi nazwę Swift 3D, a jego producentem jest firma Electric Rain (*www.swift3d.com*), natomiast drugi to Vectra3D, stworzony przez firmę Idea Works (*www.vectra3d.com*). Te dwa niedrogie programy są wspaniałymi narzędziami dla projektantów pracujących we Flashu. Pozwalają one modyfikować i animować trójwymiarowe obiekty, importować pliki z bardziej zaawansowanych (i znacznie droższych) programów 3D, takich jak 3ds max (wcześniej występujący pod nazwą 3D Studio Max) czy LightWave 3D, a także importować grafikę wektorową z programów ilustracyjnych takich jak Macromedia FreeHand. Co ważniejsze, są to jedyne dwie znane mi aplikacje, które potrafią eksportować pliki w formacie SWF. W obu przypadkach eksportowane pliki SWF zawierają wektorowe rysunki, które można umieścić w oddzielnych ujęciach kluczowych filmu Flasha (tworząc tak zwaną animację „klatka po klatce").

Zarówno program Swift 3D, jak i Vectra3D jest dostępny w wersjach dla Windows i Macintosha. Swift 3D jest ponadto dostępny jako moduł rozszerzający (*plugin*) dla programu 3ds max (Windows) i LightWave 3D (Windows i Macintosh); Vectra 3D występuje jako moduł rozszerzający jedynie dla programu 3ds max dla Windows.

Poza powyższymi cechami obu programów nie łączy zbyt wiele podobieństw. Praca w każdym z nich przebiega zupełnie inaczej.

Swift 3D

Program Swift 3D korzysta z interfejsu typowego dla aplikacji 3D (rysunek 2.43).



Rysunek 2.43. *Interfejs programu Swift 3D*

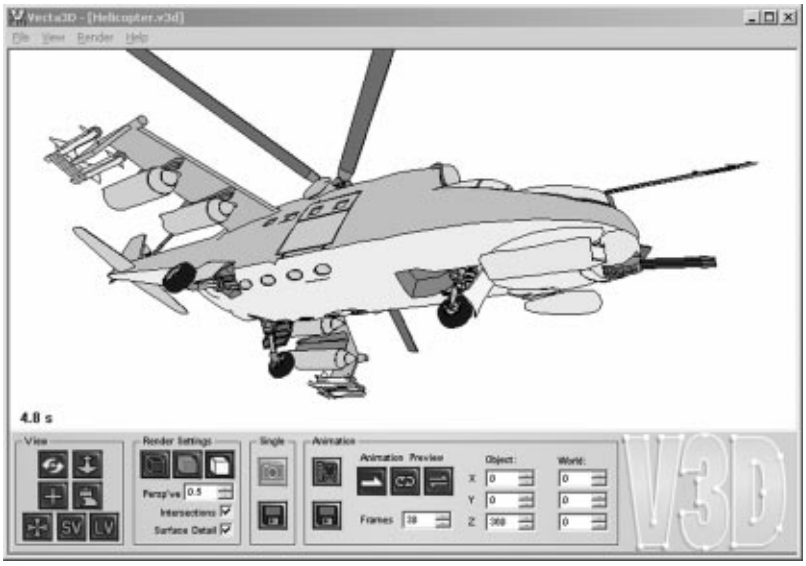
Program Swift 3D udostępnia trzy metody tworzenia animacji. Możesz importować trójwymiarową siatkę utworzoną w programie 3D Studio Max (format 3DS); możesz importować wektorowe kształty z programu ilustracyjnego, takiego jak FreeHand (format EPS) i konwertować je na trójwymiarowe obiekty za pomocą narzędzi edycyjnych programu Swift 3D; możesz wreszcie od podstaw tworzyć trójwymiarowe obiekty (w tym kształty podstawowe, takie jak sfery, stożki i pierścienie oraz tekst) wewnątrz programu.

Intuicyjny interfejs programu pozwala edytować kolory, typy fazowania, głębię, obroty i położenia. Dostępne są predefiniowane schematy kolorów i sekwencje animacyjne, z których możesz korzystać, po prostu przeciągając je i upuszczając. Program oferuje również interesujące rozwiązanie — obrotową kamerę — umożliwiającą animowanie kamery, a nie obiektu. (Dobry samouczek dotyczący stosowania tej funkcji znajdziesz pod adresem *http://www.erain.com/tutorials.asp*).

Vectra3D

Interfejs programu Vectra3D różni się od tego, który zobaczyliśmy w Swift 3D (rysunek 2.44).

Program ten nie pozwala tworzyć obiektów trójwymiarowych od podstaw ani nie oferuje gotowych schematów animacji, zatem praca w nim wygląda zgoła inaczej. Program wyświetla obiekt trójwymiarowy jako grupę punktów. Aby zobaczyć wypełnienia obiektu, musisz go wyrenderować.

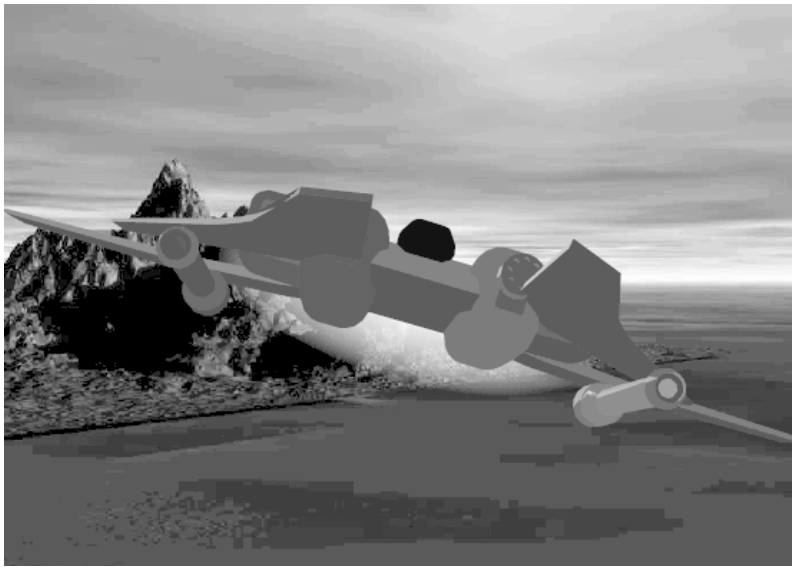


Rysunek 2.44. *Interfejs programu Vectra3D*

Studium 1. Myśliwiec

Moim pierwszym przykładem jest animacja futurystycznego myśliwca, który pikuje z nieba, wyrównuje poziom lotu, oddaje kilka strzałów, a następnie wykonuje zwrot w lewo i znika z pola widzenia. Animacja ta jest zwiastunem większego projektu, który nazwałem interfejsem Gwiezdne wrota.

Zobacz animację zapisaną w pliku *Spaceship_final.swf* na płycie CD-ROM (rysunek 2.45). Ukończony projekt możesz też zobaczyć pod adresem *www.sdflash.net/trailer.htm*.



Rysunek 2.45. *Animacja zapisana w pliku Spaceship_final.swf*

Animacja jest stosunkowo prosta i korzysta z niewielkiej liczby akcji. Jest to dowód na to, że dla uzyskania efektu trójwymiarowości we Flashu nie musisz tworzyć ton kodu.

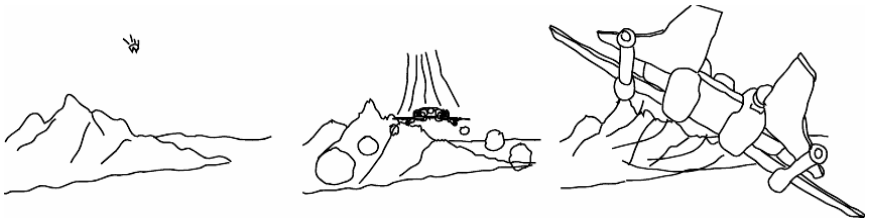
Planowanie animacji

Zanim przystąpisz do realizacji projektu, powinieneś naszkicować najważniejsze momenty trójwymiarowej animacji. Gdy zobaczysz ujęcia animacji na papierze, będziesz mógł dokładnie przeanalizować akcję i zdecydować, które elementy należy zrobić w programie renderującym, a które można zreazliować bezpośrednio we Flashu. Tym sposobem zaoszczędzisz mnóstwo czasu i frustracji na dalszych etapach produkcji.

Staraj się ograniczać liczbę ujęć renderowanych w zewnętrznej aplikacji. Pomoże ci to zminimalizować rozmiar wynikowego pliku Flasha. (Każde wyrenderowane ujęcie animacji to oddzielny rysunek, więc takie rozwiązanie wpływa na szybki wzrost rozmiaru pliku).

Rysunek 2.46 przedstawia scenariusz obrazkowy mojej animacji.

Rysunek 2.46. *Ręczny szkic najważniejszych ujęć animacji*



Ze scenariusza obrazkowego jasno wynika, że Flash miałby problemy z utworzeniem obracającego się myśliwca — gdy ten obraca się z niemal pionowej pozycji i przyjmuje pozycję frontalną, a następnie odbija w lewo.

Flash nie jest w stanie samodzielnie wyświetlić tych różnych widoków, ponieważ nie potrafi zasłaniać jednych i odsłaniać innych części obiektu. W związku z tym podzielimy zadania animacyjne pomiędzy Flasha i program Swift 3D w następujący sposób:

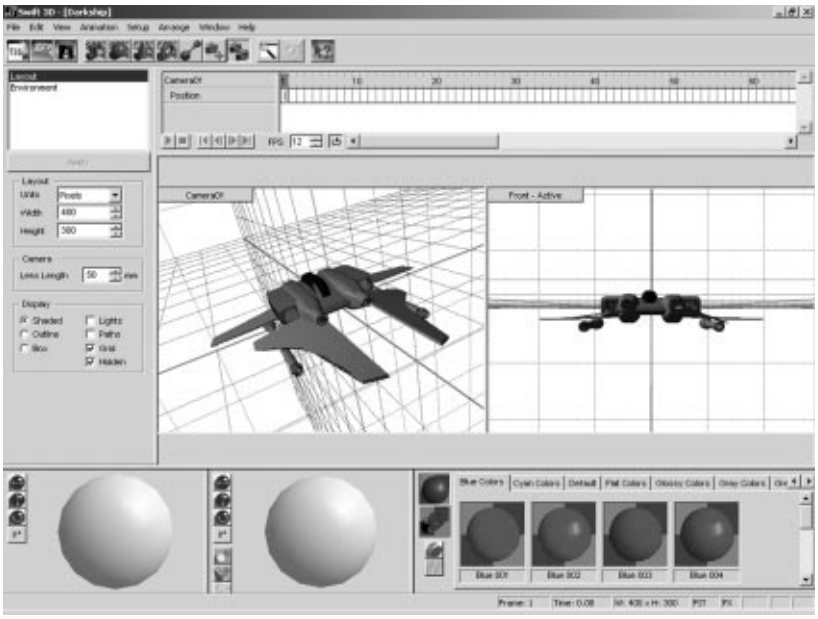
1. Myśliwiec zmienia pozycję z pionowej na poziomą i wyrównuje poziom lotu (Swift 3D).
2. Podczas wykonywania manewru obrotu myśliwiec schodzi w dół i zbliża się do obserwatora (Flash).
3. Myśliwiec strzela w stronę obserwatora (Flash).
4. Myśliwiec odbija w lewo (Swift 3D).
5. Podczas odbijania myśliwiec wznosi się i ucieka do prawego górnego narożnika ekranu, po czym znika (Flash).

Omówmy sekwencję tworzoną w programie Swift 3D.

Rozpoczynamy projekt w programie Swift 3D

W programie Swift 3D otwórz plik *Spaceship_Final.t3d*, zapisany na płycie CD-ROM (rysunek 2.47). Jest to plik tego programu; na płycie CD-ROM znajdziesz również próbną wersję programu Swift 3D.

Rysunek 2.47. *Gotowy obiekt programu Swift 3D, który wyeksportujemy do Flasha*



Kliknij ikonę *Play Animation* (symbol trójkąta) w lewym dolnym narożniku listwy czasowej, by sprawdzić przebieg animacji. Zobaczysz, jak myśliwiec z pozycji pionowej wyrównuje lot do poziomu, a następnie odbija w lewo. Dokładnie taki ruch myśliwca opisaliśmy na wcześniejszej liście.

Zwróć uwagę, że aktywna kamera nosi nazwę *camera01*. Gdy program Swift 3D importuje plik 3D Studio, dziedziczy pewne ustawienia tego programu, takie jak kąt widzenia kamery, światła i kolory. Jeśli jesteś użytkownikiem programu 3D Studio, jest to dla ciebie duże ułatwienie, ponieważ możesz przygotować kompletny obiekt przed przystąpieniem do pracy w programie Swift 3D.

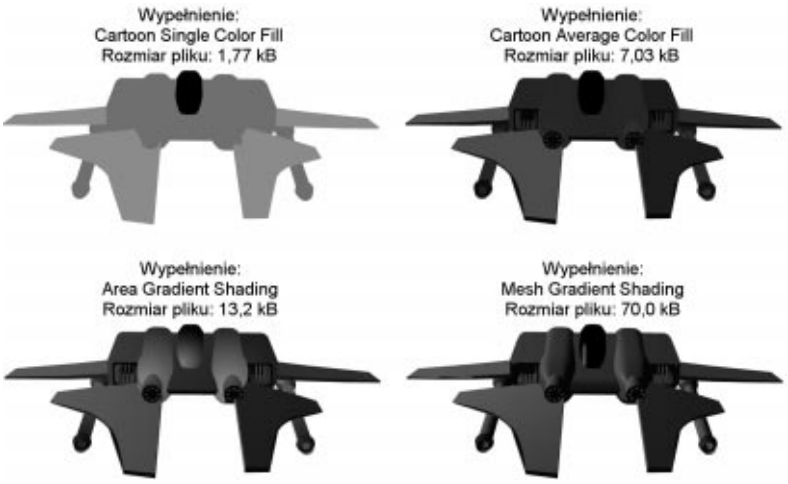
W Internecie możesz znaleźć mnóstwo darmowych siatek trójwymiarowych obiektów, przeznaczonych dla użytkowników, którzy nie mają czasu lub zdolności do przygotowywania własnych modeli. Sprawdź witryny *www.3dcafe.com* i *www.highend3d.com*. Używany przez nas model znalazłem na witrynie *www.3dcafe.com*.

Jak widać na rysunku 2.48, nasza animacja zawiera trzy ujęcia kluczowe. Operacje obrotu i pochylenia modelu w poszczególnych ujęciach kluczowych wykonałem za pomocą kulowego manipulatora obiektów (*Object Trackball*), znajdującego się w lewym dolnym narożniku interfejsu.

Rysunek 2.48. *Ułożenie myśliwca w trzech ujęciach kluczowych*

Gdy animacja jest już zdefiniowana, czas na wyeksportowanie pliku SWF. Przystępując do tego procesu, powinieneś rozważyć dwa czynniki — liczbę wielokątów używanych przez moduł renderujący do przedstawienia każdego ujęcia oraz typ wypełnienia. Im bardziej złożony typ wypełnienia wybierzesz, tym więcej szczegółów będzie zawierał tworzony obiekt i tym większy będzie rozmiar pliku dla każdej klatki[6]. Rysunek 2.49 dosadnie ilustruje ten problem. Poeksperymentuj, by znaleźć kompromis pomiędzy liczbą szczegółów a rozmiarem pliku.

Rysunek 2.49. *Ten sam obiekt wyeksportowany z różnymi ustawieniami wypełnienia*



Otwórz plik *Spaceship_rendered.swf* zapisany na płycie CD-ROM. Jest to animacja myśliwca wyeksportowana przez program Swift 3D.

Flash nie jest programem, w którym można umieszczać złożone obiekty trójwymiarowe. Zaimportuj obiekt złożony z 70 000 wielokątów, a zrozumiesz, co mam na myśli — oczywiście pod warunkiem, że w ogóle uda ci się wczytać taką animację! Znalezienie kompromisu pomiędzy rozmiarem pliku i jakością obrazu wymaga cierpliwości — czasem będziesz musiał wyrenderować obiekt kilka razy, zanim znajdziesz optymalne rozwiązanie.

[6] Aby ustawić typ wypełnienia w programie Swift 3D, przejdź na zakładkę *Preview and Export Editor*, w polu *Output Options* wybierz opcję *Fill Options*, a następnie z listy *Fill Type* wybierz odpowiednią metodę wypełnienia — *przyp. tłum.*

Finalizujemy projekt we Flashu

Po utworzeniu i wyrenderowaniu trójwymiarowego obiektu poza Flashem, możemy wzbogacić go we Flashu — łącząc realizm grafiki trójwymiarowej z wydajnymi rozwiązaniami grafiki wektorowej Flasha.

Otwórz plik *Spaceship_final.fla* zapisany na płycie CD-ROM (rysunek 2.50).

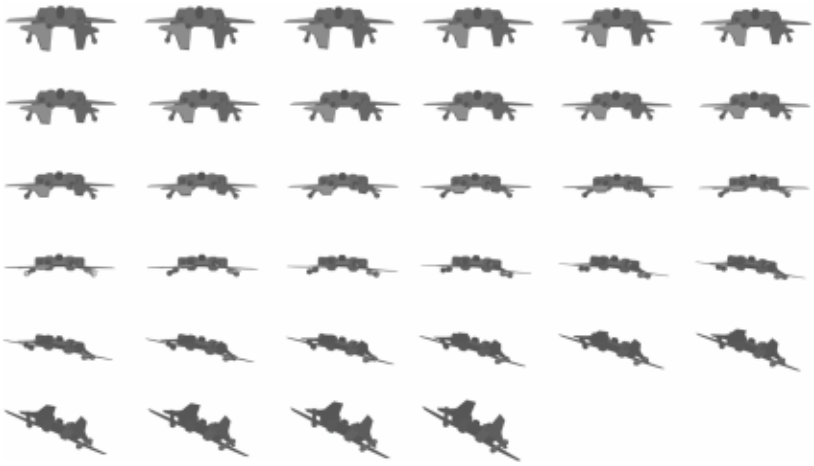
Rysunek 2.50. *Kontynuujemy prace nad projektem we Flashu*

Importujemy animację

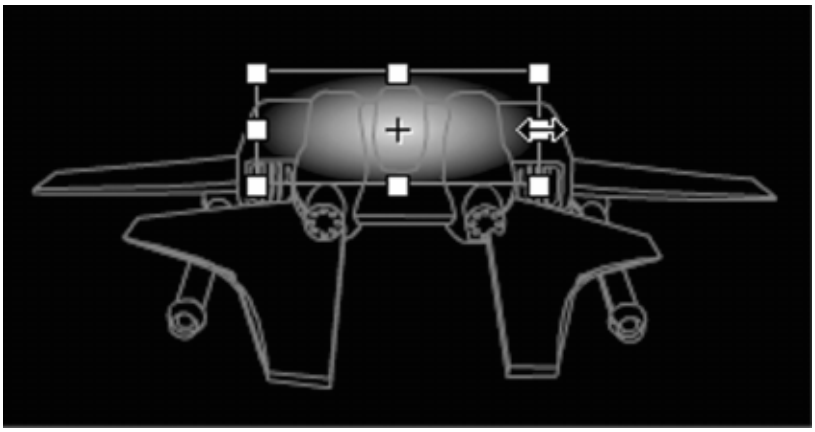
We Flashu rozpocznij pracę od utworzenia klipu filmowego o nazwie `ClipSpaceship`, w którym umieść wyrenderowany obiekt trójwymiarowy. Następnie zaimportuj animację SWF wygenerowaną w programie Swift 3D, umieszczając ją na domyślnej warstwie klipu. Warstwie tej nadaj nazwę `spaceship`. Po zaimportowaniu animacji warstwa ta powinna zawierać 34 ujęcia kluczowe (rysunek 2.51).

Płomień silników i strzały działka

Następnie utwórz symbol graficzny `Laser` zawierający koło wypełnione gradientem radialnym. Posłuży on do utworzenia efektów płomieni silników odrzutowych oraz strzałów działka. Umieść go na warstwie `engine thrust` klipu `ClipSpaceship`. Ważne jest, by warstwa ta znajdowała się pod warstwą `spaceship`, ponieważ płomień silników musi się znajdować za myśliwcem. Zmień skalę symbolu, tak aby wyglądał jak realistyczny żar emitowany z silników (rysunek 2.52). Stosując tę technikę, możesz nawet uzyskać efekt podobny do tego z *Gwiezdnych wojen* — myślę tu o biało-niebieskiej łunie wydobywającej się z silników myśliwca Millenium Falcon, którym latali Han Solo i Chewie.

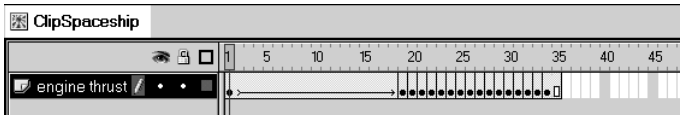


Rysunek 2.51. *Kolejne ujęcia kluczowe animacji myśliwca*



Rysunek 2.52. *Zmień skalę symbolu, aby dopasować łunę do wymiarów myśliwca*

W pierwszych dwudziestu klatkach klipu filmowego myśliwiec porusza się po linii prostej, zatem za pomocą zwykłej automatycznej animacji można sprawić, by płomień podążał za silnikami. W ostatnich 15 klatkach myśliwiec porusza się mniej regularnie i śledzenie go staje się bardziej kłopotliwe. W związku z tym w ostatnich 15 klatkach umieść ujęcia kluczowe, animując płomień silników metodą „klatka po klatce" (rysunek 2.53). Kolejne ujęcia kluczowe zawierają przeskalowane i obrócone klony symbolu `Laser`.



Rysunek 2.53. *Animacja płomieni stanowi kombinację automatycznej animacji ruchu i animacji „klatka po klatce"*

Następnie przejdź do tworzenia strzałów działek laserowych, kierowanych w stronę obserwatora. Rozpocznij od strzału prawego (patrząc od strony obserwatora) działka. Utwórz dla niego symbol klipu filmowego o nazwie `ClipLaserRight`, umieść na środku klipu klon symbolu `Laser` i zmień skalę klonu (wysokość = 13,6; szerokość = 13,6). W czwartej klatce utwórz nowe ujęcie kluczowe i ponownie zmień skalę klonu lasera

(wysokość = 272; szerokość = 272) oraz jego współrzędną *x* zmień na –128. Wreszcie w siódmej klatce utwórz trzecie ujęcie kluczowe, w którym ponownie zmodyfikuj laser (wysokość = 340; szerokość = 340; współrzędna *x* = –176). Modyfikując współrzędną *x*, zwiększysz realizm efektu w wynikowym filmie.

W ujęciu kluczowym w klatce numer 7 umieść następującą akcję:

```
gotoAndPlay(1);
```

Aby utworzyć efekt strzału z lewego działka, zduplikuj poprzednio utworzony strzał i nadaj duplikatowi nazwę `ClipLaserRight`. Następnie w duplikacie zmodyfikuj współrzędną *x* lasera w ujęciach 4. i 7., zmieniając w panelu *Info* wartości współrzędnej z ujemnych na dodatnie. Dzięki temu strzał lewego działka będzie się przemieszczał w przeciwną stronę.

Dźwięk

Ostatnim krokiem związanym ze strzałem działek jest dodanie dźwięku strzału. Zwróć uwagę, że dźwięk ten umieściłem tylko w jednym z dwóch klipów filmowych strzałów. Ponieważ działka strzelają równocześnie, nie ma potrzeby umieszczania dźwięku w obu klipach. Preferuję sterowanie dźwiękiem za pomocą skryptów, ponieważ obiekt *Sound* Flasha 5 oferuje znakomite możliwości w tej dziedzinie. (Więcej informacji na temat dźwięku we Flashu znajdziesz w rozdziale 4.).

Zwykle wszystkie używane w filmie dźwięki definiuję w pierwszym ujęciu głównej listwy czasowej. To ułatwia wyszukiwanie i usuwanie ewentualnych błędów w kodzie. Symbole dźwięków zawarte w bibliotece udostępniam skryptom, definiując dla każdego z nich identyfikator eksportu w oknie *Symbol Linkage Properties*. Zdefiniowany tam identyfikator funkcjonuje tak jak zewnętrzna nazwa klipu filmowego — za jego pomocą możesz odnosić się w swoich skryptach do symboli zawartych w bibliotece. Aby ułatwić sobie zapamiętanie identyfikatorów, nadaję im takie same nazwy, jak nazwy symboli wewnątrz biblioteki.

W pierwszym ujęciu klipu `ClipLaserRight` umieść następującą akcję:

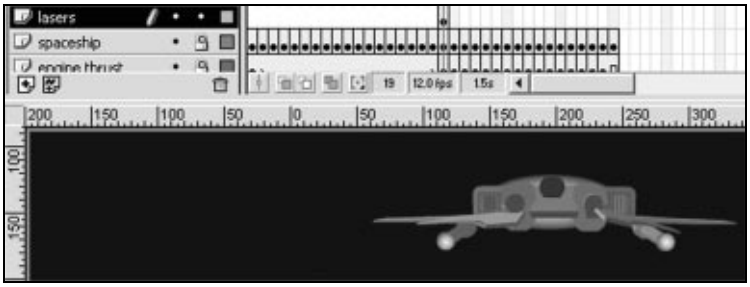
```
_root.blastSound.start();
```

Przygotowanie klipu myśliwca

Wróćmy jeszcze do prac nad klipem filmowym `ClipSpaceship`. Ponad warstwą `spaceship` utwórz nową warstwę o nazwie `lasers`. Działka powinny wystrzelić, gdy myśliwiec wyrówna swój lot, co oznacza, że strzały muszą się pojawić w 19. klatce klipu filmowego.

Utwórz w tej klatce puste ujęcie kluczowe. W nim umieść, dokładnie na wylocie działek, dwa klony klipów filmowych przedstawiających strzały lasera. Rysunek 2.54 przedstawia wygląd ujęcia kluczowego w 19. klatce klipu.

W ujęciach klipu filmowego myśliwca umieścimy kilka prostych akcji, które ułatwią sterowanie tym klipem w dalszej części projektu. W tym celu utwórz nową warstwę o nazwie



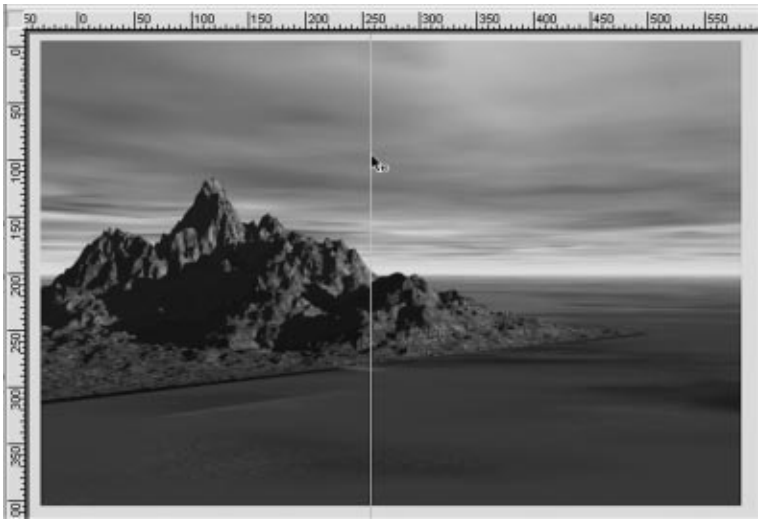
Rysunek 2.54. *Myśliwiec w chwili rozpoczęcia strzału*

`frame scripts` i na niej umieść dwa ujęcia kluczowe, odpowiednio w klatkach 19. i 35. W każdym z tych ujęć kluczowych umieść akcję `stop()`. (Na razie może się to wydawać niezrozumiałe, lecz wszystko się wyjaśni, gdy połączymy wszystkie elementy animacji).

Łączymy wszystkie elementy

Teraz, gdy klip filmowy myśliwca jest już gotowy, czas połączyć wszystkie elementy na głównym obrazie filmu. W scenie 1. utwórz nową warstwę nad warstwą `background` i nadaj jej nazwę `spaceship`. Aby ułatwić sobie wyrównywanie elementów, możesz posłużyć się miarką i liniami pomocniczymi Flasha 5. (Jeśli jesteś użytkownikiem programu FreeHand, pewnie nieraz już używałeś tych narzędzi).

Rysunek 2.55 przedstawia linię pomocniczą umieszczoną w centralnej części obrazu.



Rysunek 2.55. *Linia pomocnicza pomoże ci w animowaniu myśliwca*

Gdy jest aktywna warstwa `spaceship`, zaznacz w bibliotece klip filmowy `ClipSpaceship` i przeciągnij go na obraz. W panelu *Instance* nadaj jego klonowi nazwę `darkship_mc`.

Do nazw klonów klipów filmowych zawsze dodaję końcówkę `_mc` (*movie clip*). W ten sposób mogę łatwo poznać, czy odnoszę się do klipu filmowego, czy do zmiennej. Zwiększa to również czytelność kodu, co ma duże znaczenie podczas wyszukiwania ewentualnych błędów.

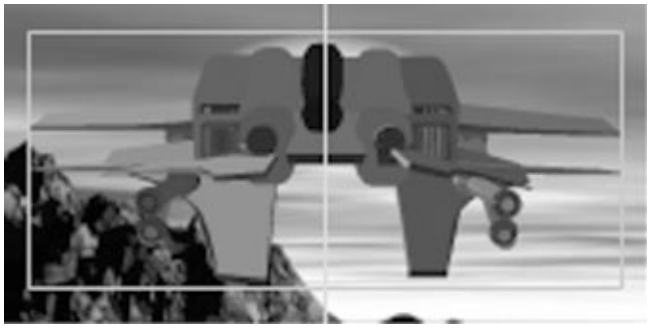
Zmień skalę klipu do 10 procent pierwotnej wielkości, a następnie umieść go w takim miejscu obrazu, by jego środek znalazł się dokładnie na linii pomocniczej. Myśliwiec powinien znajdować się w pobliżu górnej krawędzi obrazu, na przykład w punkcie o takich współrzędnych:

```
X: 237.1
Y: 0.1
```

W 20. klatce warstwy `spaceship` utwórz ujęcie kluczowe. W tym momencie myśliwiec zakończy nurkowanie i wyrówna lot.

Gdy umieścisz klip filmowy myśliwca na obrazie Flasha, w środowisku edycyjnym cały czas jest widoczne tylko pierwsze ujęcie klipu. To utrudnia właściwe ułożenie klipu, ponieważ musisz sobie wyobrazić ujęcie klipu, które będzie wyświetlane w danym momencie w wynikowym filmie. Możesz jednak posłużyć się sztuczką. Zaznacz klip `ClipSpaceship` i edytuj go. Utwórz nową warstwę na szczycie stosu warstw. Skopiuj myśliwiec z 19. klatki i umieść go w pierwszym ujęciu nowej warstwy.

Wróćmy jednak do edycji głównej listwy czasowej. Rysunek 2.56 przedstawia wygląd klipu w 20. klatce.



Rysunek 2.56. *Klip myśliwca podczas edycji w 20. klatce animacji*

W klatce numer 20 zwiększ skalę klipu do 65 procent i zmień jego współrzędne na następujące:

```
X: 128.2
Y: 54.0
```

W panelu *Frame* wybierz automatyczną animację ruchu (*Motion Tween*) i pozostaw domyślne ustawienia pozostałych opcji. Następnie utwórz kolejne ujęcie kluczowe w klatce numer 40. Tam zwiększ skalę myśliwca do 150 procent i przesuń go do współrzędnych:

```
X: -38.9
Y: 43.9
```

Ponownie wybierz automatyczną animację ruchu, pozostawiając jej domyślne parametry. W tej klatce musimy również wstawić akcję, która wznowi odtwarzanie klipu myśliwca po tym, jak zatrzyma się na akcji `stop()` w klatce numer 19 listwy czasowej klipu. Na głównej listwie czasowej utwórz nową warstwę i nadaj jej nazwę `frame scripts` (skrypty ujęć).

W ujęciu kluczowym w klatce numer 1 umieść poniższy skrypt, który tworzy obiekty *Sound* dla dźwięków używanych w filmie.

```
blastSound = new Sound();
blastSound.attachSound("blast");
xwingSound = new Sound();
xwingSound.attachSound("xwing");
```

Aby odtworzyć dźwięk nadlatującego myśliwca, utwórz nowe ujęcie kluczowe w klatce numer 6 i umieść w nim następującą akcję:

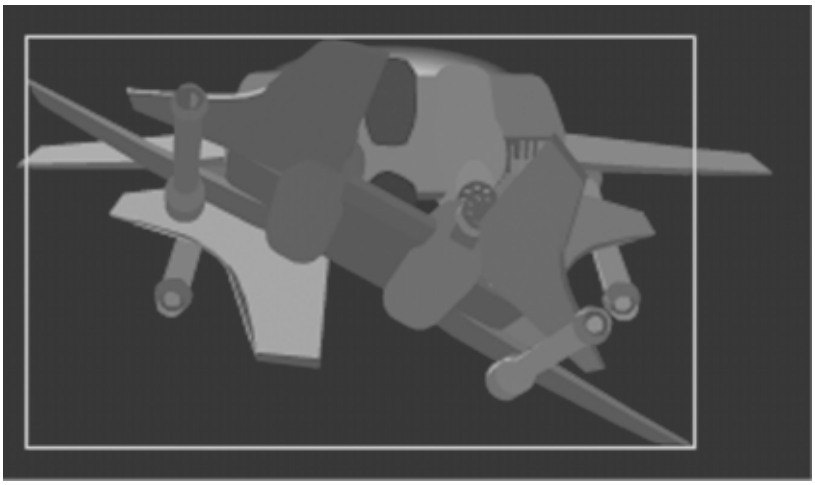
```
xwingSound.start();
```

Aby sterować klipem myśliwca, utwórz puste ujęcie kluczowe w 40. klatce warstwy `frame scripts` i umieść w nim taką akcję:

```
_root.darkship_mc.gotoAndPlay(20);
```

Tym sposobem wznowimy odtwarzanie klipu filmowego w miejscu, w którym myśliwiec odbija w lewo.

Aby ułatwić sobie prawidłowe ułożenie myśliwca w następnym ujęciu kluczowym (w 54. klatce), zastosuj tę samą sztuczkę co poprzednio — w klipie myśliwca skopiuj rysunek z 35. klatki i wklej go do pierwszej klatki klipu (rysunek 2.57). (Pamiętaj, by po zakończeniu edycji usunąć z klipu pomocnicze kopie myśliwca).



Rysunek 2.57. *Tymczasowo kopiując rysunek z odpowiedniej klatki klipu, ułatwisz sobie umieszczenie myśliwca w odpowiednim miejscu*

Na głównej listwie czasowej utwórz nowe ujęcie kluczowe w klatce numer 54 warstwy `spaceship`. Zwiększ skalę klipu do 300 procent i nadaj mu następujące współrzędne:

```
X: -252.8
Y: -347.0
```

Wreszcie, w 60. klatce warstwy `spaceship` utwórz ostatnie ujęcie kluczowe i przesuń klip do współrzędnych:

```
X: -9.8
Y: -499.0
```

I to już wszystko. Choć to studium nie jest skomplikowane z programistycznego punktu widzenia, zawiera kilka aspektów wartych przemyślenia.

W tym studium poznałeś podstawy korzystania z trójwymiarowych obiektów, lecz to jeszcze nie koniec. Pokażę teraz, jak można stosować trójwymiarowe obiekty w większym projekcie.

Studium 2. Interfejs Gwiezdne wrota

Fantastycznonaukowy film *Gwiezdne wrota* zainspirował mnie do utworzenia kolejnego projektu, który nazwałem interfejsem Gwiezdne wrota (*Stargate Interface*). Prezentowany tu interfejs to prototyp, lecz możesz w nim zobaczyć, co można osiągnąć za pomocą nieco większej ilości kodu ActionScript oraz mnóstwa wyobraźni. W czasie, gdy książka ta znajdzie się na półkach, projekt powinien być dostępny pod adresem *www.sdflash.net*.

Tematem przewodnim witryny są okrągłe wrota z ośmioma symbolami, gwiezdny nawigator (*Star Navigator*) ustawiający symbole dla poszczególnych sekwencji kodu oraz menu z sześcioma opcjami. Każda opcja menu jest powiązana z sekwencją kodu, złożoną z czterech symboli, które muszą być aktywowane przez wrota, by przejść do odpowiedniej podstrony. Symbole są wyróżniane przez gwiezdny nawigator, który wyszukuje je i blokuje je we wrotach. Po zablokowaniu wszystkich symboli w sekwencji kodu, film przechodzi do strony odpowiadającej wybranej opcji.

Otwórz plik *Stargate.swf* zapisany na płycie CD-ROM, by zobaczyć opisywany prototyp projektu.

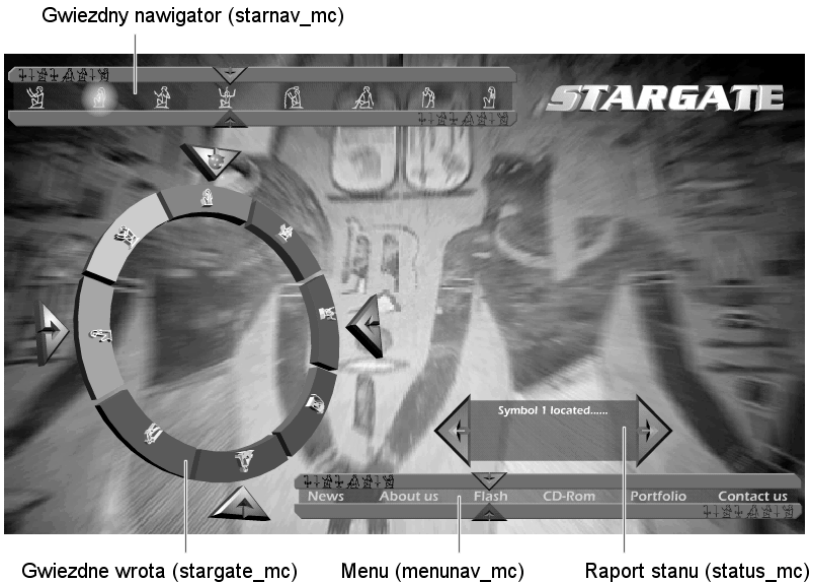
Wszystkie trójwymiarowe obiekty projektu powstały w programie Swift 3D. Same wrota zaprojektowałem jako płaski rysunek wektorowy w programie FreeHand, a następnie zaimportowałem go do programu Swift 3D. Wszystkie trójwymiarowe symbole wrót sprawiły, że powstał plik dosyć sporych rozmiarów, lecz zdecydowałem się kontynuować projekt ze względu na zadowalające rezultaty.

Interfejs składa się z czterech współpracujących ze sobą klipów filmowych (rysunek 2.58).

Inicjalizacja

Wszelkie czynności nawigacyjne rozpoczynają się w menu, gdzie użytkownik wybiera jedną z dostępnych opcji. Z każdą z opcji jest związana inna sekwencja kodu, złożona z czterech symboli. Poszczególne sekwencje są zdefiniowane w sześciu tablicach na początku filmu:

```
newsarray = new Array(5, 7, 4, 3);
aboutarray = new Array(3, 5, 6, 7);
flasharray = new Array(4, 1, 5, 3);
cdromarray = new Array(2, 5, 7, 3);
portfolioarray = new Array(2, 6, 1, 3);
contactarray = new Array(1, 5, 2, 6);
```



Rysunek 2.58. *Układ elementów nawigacyjnych*

Tablice te sterują całym schematem animacji. Ponadto na początku filmu zdefiniowałem pewne zmienne, które również mają kluczowe znaczenie w sterowaniu interfejsem:

```
// wartość pierwszego symbolu w sekwencji kodu
var symbol0 = 0;
// wartość drugiego symbolu w sekwencji kodu
var symbol1 = 0;
// wartość trzeciego symbolu w sekwencji kodu
var symbol2 = 0;
// wartość czwartego symbolu w sekwencji kodu
var symbol3 = 0;
// zmienna informująca wrota o tym, który z czterech symboli w sekwencji kodu należy
↳ustawić;
var step;
```

Wybór opcji z menu

Gdy użytkownik kliknie jeden z przycisków w menu, dwie strzałki ustawiają się na wskazanej pozycji. W chwili kliknięcia ustalana jest wartość pomocniczej zmiennej określającej miejsce, w którym strzałki się zatrzymają. Gdy strzałki dotrą do celu, wykonywany jest poniższy skrypt:

```
// sprawdzamy, czy zmienna posiada prawidłową wartość
if (this.menuitem == "news") {
   // przepisujemy wartości z odpowiedniej tablicy do zmiennych
   // odpowiadających czterem symbolom
   for (counter=0; counter<=3; counter++) {
      _root["symbol"+counter] = _root.newsarray[this.counter];
   }
   // niech wrota blokują pierwszy symbol w sekwencji kodu
   _root.step = 1;
   // niech gwiezdny nawigator odszuka pierwszy symbol sekwencji kodu
   _root.starnav_mc.counter = 0;
```

```
    // aktywuj gwiezdny nawigator
    _root.starnav_mc.play();
    // wyświetl odpowiedni raport o stanie interfejsu
    _root.status_mc.message = "Symbol tracking  initiated......";
    stop ();
}
```

Gwiezdny nawigator

Jak widać w powyższym kodzie, w tym momencie aktywowaliśmy gwiezdny nawigator. W nim zdefiniowałem funkcję, do której odwołuję się za każdym razem, gdy strzałki mijają symbol. Oto ta funkcja:

```
// argument lock określa symbol mijany przez strzałki
function funcStarlock (lock) {
// sprawdź, czy mijany jest właściwy symbol i czy zmienna step ma właściwą wartość
if (_root["symbol"+counter] == lock && _root.step==1) {
// wyświetl raport stanu
_root.status_mc.message = "Symbol "+_root.step+" located......";
//aktywuj klip gwiezdnych wrót
_root.stargate_mc.play();
// aktywuj migające światło na symbolu
setProperty (_root.starnav_mc["lock"+lock+"_mc"], _visible, true);
_root.locksound.start();
counter++;
stop ();
//ten fragment jest powtarzany dla każdego symbolu w sekwencji
} else if (_root["symbol"+counter] == lock && _root.step==2) {
_root.status_mc.message = _root.status_mc.message+"\nSymbol "+_root.step+"
↳located......";
_root.stargate_mc.play();
setProperty (_root.starnav_mc["lock"+lock+"_mc"], _visible, true);
_root.locksound.start();
counter++;
stop ();
} else if (_root["symbol"+counter] == lock && _root.step==3) {
_root.status_mc.message = _root.status_mc.message+"\nSymbol "+_root.step+"
↳located......";
_root.stargate_mc.play();
setProperty (_root.starnav_mc["lock"+lock+"_mc"], _visible, true);
_root.locksound.start();
counter++;
stop ();
} else if (_root["symbol"+counter] == lock && _root.step==4) {
_root.status_mc.message = "Symbol "+_root.step+" located......\nLaunch sequence
↳activated!!!!!";
_root.stargate_mc.play();
setProperty (_root.starnav_mc["lock"+lock+"_mc"], _visible, true);
_root.locksound.start();
counter++;
stop ();
}
}
```

Za każdym razem gdy strzałki docierają do symbolu, odwołuję się do tej funkcji, podając wartość symbolu jako argument, w taki sposób:

```
funcStarlock(1);
//1 jest wartością, która zostanie przypisana zmiennej lock
```

Po znalezieniu symbolu aktywuję klip filmowy gwiezdnych wrót (`stargate_mc`). Ponownie korzystam z nieocenionej akcji `function`, definiując dwie następne funkcje. Pierwsza z nich obsługuje pierwsze trzy symbole w sekwencji, zaś druga zajmuje się ostatnim, czwartym symbolem w sekwencji kodu. Podział ten wynika z faktu, że w przypadku ostatniego symbolu wykonywane są nieco inne akcje. Oto te dwie funkcje:

```
// funkcja dla pierwszych trzech symboli w sekwencji kodu
function funcStargate (symnum,symvalue,stepnum,clipname) {
if (_root["symbol"+symnum] == symvalue && _root.step == stepnum) {
// aktywuj odpowiednią strzałkę blokady w gwiezdnych wrotach
_root.stargate_mc[clipname+"_mc"].gotoAndPlay(10);
_root.gatelocksound.start();
// wznów aktywność gwiezdnego nawigatora
_root.starnav_mc.play();
_root.step++;
stop();
}
}
// funkcja dla ostatniego symbolu w sekwencji kodu
// zwróć uwagę na różnice w porównaniu z poprzednią funkcją
function funcStargate2 (symnum,symvalue,stepnum,clipname) {
if (_root["symbol"+symnum] == symvalue && _root.step == stepnum) {
_root.stargate_mc[clipname+"_mc"].gotoAndPlay(10);
_root.gatelocksound.start();
// skok do początku sekwencji podróży
_root.gotoAndPlay("travelstart");
_root.step++;
stop();
}
}
stop();
```

Te funkcje są wywoływane w klipie filmowym `stargate_mc` i uruchamiają odpowiednie elementy interfejsu. Wartości argumentów, którymi operujemy, zależą od położenia symboli na gwiezdnych wrotach oraz strzałek blokady, znajdujących się obok nich. Oto akcje zawarte w jednym z ujęć kluczowych:

```
// wywołanie funkcji dla pierwszych trzech symboli
funcStargate(0,1,1,"topnav");
funcStargate(1,7,2,"rightnav");
funcStargate(2,5,3,"bottomnav");
// wywołanie funkcji dla ostatniego, czwartego symbolu
funcStargate2(3,3,4,"leftnav");
```

W tym miejscu kończy się sekwencja kodu. Teraz czas na rozpoczęcie sekwencji podróży.

Sekwencja podróży

W sekwencji podróży gwiazda jest lokalizowana w podobny sposób, jak urządzenie GPS lokalizuje samochód. Wybierasz cel podróży, a komputer wybiera trasę. W interfejsie

zdefiniowałem sześć potencjalnych celów podróży i szesnaście „przystanków", które możesz minąć na drodze do celu. W klipie filmowym o nazwie `mapping_mc` utworzyłem sześć tras, po jednej dla każdego celu podróży (rysunek 2.59).

Rysunek 2.59. *Klip mapping_mc zawiera mapy tras prowadzących do poszczególnych celów podróży*

Funkcja `funcStargate2` zleca głównej listwie czasowej wykonanie skoku do etykiety o nazwie `travelstart`. W ujęciu oznaczonym tą etykietą rozpoczyna się sekwencja wyznaczania trasy podróży. Korzystamy przy tym z kolejnej funkcji, którą zdefiniowałem wcześniej w listwie czasowej, a która wyznacza trasę podróży. Oto postać tej funkcji:

```
var destination;
var label="route";
function funcTravelstart () {
// cel jest określany przez losową liczbę całkowitą z przedziału pomiędzy 1 i 6
destination = math.floor(math.random()*6)+1;
// wygeneruj nazwę etykiety, do której wykonamy skok
// (do tekstu zawartego w zmiennej label dodaj wylosowaną liczbę)
destination = label+destination;
}
```

Funkcja `funcTravelStart()` jest wywoływana w głównej listwie czasowej. Powoduje ona wykonanie skoku w klipie `mapping_mc` do wygenerowanej przez nią etykiety. Oto skrypt wykonujący ten skok:

```
funcTravelstart();
_root.mapping_mc.gotoAndPlay(destination);
```

Po ustaleniu trasy możemy rozpocząć sekwencję podróży. Taki przynajmniej jest mój zamiar — na razie nie zrealizowałem jeszcze tego fragmentu. Z tego względu nasze „zwiedzanie" prototypu interfejsu Gwiezdne wrota kończy się w tym miejscu. Mam nadzieję, że spodobał ci się przykład połączenia w jednym projekcie renderowanych obiektów trójwymiarowych ze skryptami ActionScript i być może stanie się on inspiracją do powstania innych tego typu projektów.

W czasie, gdy książka trafi na półki księgarń, powinien być już gotowy cały projekt. Odwiedź stronę *www.sdflash.net* i sprawdź, jaką trasę wśród gwiazd wybrałem.

Podsumowanie

Tym sposobem dotarliśmy do końca rozdziału. Mam nadzieję, że wiesz już, jak korzystać z renderowanej grafiki trójwymiarowej i zechcesz tworzyć ciekawe efekty wizualne, dzięki którym twoje filmy staną się bardziej realistyczne. Miłej pracy w trzech wymiarach!