WPROWADZENIE DO

fizyki w grach, animacjach i symulacjach Flash

DEV RAMTAL
ADRIAN DOBRE

Helion

Tytuł oryginału: The Essential Guide to Physics for Flash Games, Animation, and Simulations

Tłumaczenie: Julia Szajkowska

ISBN: 978-83-246-4473-5

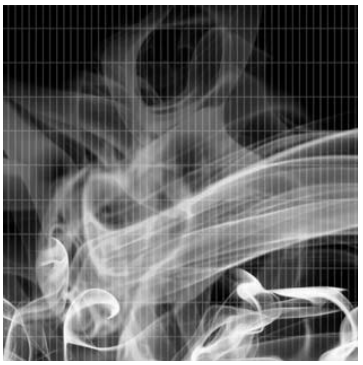
Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: http://helion.pl (księgarnia internetowa, katalog książek)

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

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

Printed in Poland.

Spis treści



Rozdział 8.

Siła sprężystości — drgania sprężyny

Sprężyny to jedno z najbardziej przydatnych narzędzi wykorzystywanych do wywoływania interesujących zjawisk fizycznych oraz ich modelowania. Okazuje się, że drgania sprężyste występują w bardzo wielu układach, przez co zachowanie ciał należących do tych układów daje się opisywać za pomocą równań związanych z ruchem ciała na sprężynie. Stąd prosty wniosek — czas poświęcony na poznanie ruchu drgającego nie będzie czasem straconym. Uważaj jednak, sprężyny uzależniają!

W tym rozdziale opiszemy następujące tematy:

- **Drgania swobodne** — ruch ciała, na które działa **siła sprężystości**.
- **Drgania tłumione** — tłumienie to wynik rozpraszania energii drgań, więc w efekcie doprowadza do ustania ruchu ciała.
- **Drgania wymuszone** — odpowiednia siła zewnętrzna działająca na układ może wprowadzić ciało w drgania i utrzymać te drgania mimo występowania w układzie sił tłumiących.
- **Układy oscylatorów** — tworzenie układów wielu sprężyn i ciał o różnych masach pozwala uzyskać bardzo interesujące efekty.

Sprężyny i oscylatory — podstawowe zjawiska

O oscylatorach pisaliśmy już w rozdziale 3., gdy wprowadziliśmy pojęcie funkcji sinusoidalnej oraz ideę łączenia ze sobą funkcji okresowych (szczegóły znajdziesz w podrozdziale „Funkcje trygonometryczne w animacjach"). **Drgania** to ruch powtarzalny i następujący względem określonego położenia równowagi. Najprostszymi przykładami ruchu drgającego są wahania wahadła czy też wychylenia huśtawki.

Drgania pojawiają się także w przyrodzie. Ruch drzew na wietrze to jeden z przykładów drgań, innym może być ruch ciał unoszących się i opadających na wodzie w wyniku jej falowania. Podobny ruch obserwujemy

w urządzeniach zawierających sprężyny, dlatego też **ruch drgający** nazywa się bardzo często **drganiami sprężystymi** — sami będziemy stosować te określenia wymiennie. Co więcej, podobieństwo zjawisk nie ogranicza się do synonimicznych nazw: układy drgające modeluje się właśnie za pomocą zestawów „sprężyn".

Co jest źródłem drgań i dlaczego są one tak często spotykane? Zanim zajmiemy się przygotowywaniem modeli, postaramy się odpowiedzieć na te pytania.

Ruch drgający

Wspominaliśmy już o ruchu drgającym, choć nie zrobiliśmy tego jawnie. Przypomnij sobie symulację piłki wypływającej z wody, którą przygotowałeś w czasie pracy z poprzednim rozdziałem — piłka oscylowała przez chwilę wokół położenia równowagi, nim ostatecznie określiła swoje położenie. Podobny ruch staje się udziałem wielu innych ciał — w ten sam sposób zachowuje się samochód wykorzystujący układ amortyzacji, podobny ruch wykonują drzewa gnące się na wietrze, ich gałęzie czy liście poruszane podmuchami powietrza. Inną grupę stanowią ciała, których ruch pozornie nie ma nic wspólnego z drganiami sprężystymi, a mimo to modeluje się go za pomocą układów wirtualnych sprężyn. Typowe przykłady to odkształcenia ciał, na przykład lin, tkaniny czy włosów. Domyślasz się już zapewne, że musi istnieć wspólny mianownik wszystkich tych układów, który sprawia, że ich zachowanie można przyrównać do zachowania odkształconej sprężyny. Jakie to cechy?

Siła sprężystości, tłumienie i wymuszanie

W układzie drgającym wyróżnia się:

- **punkt równowagi**, czyli takie położenie ciała, w którym to ciało pozostałoby na stałe, gdyby się nie poruszało;
- **siłę sprężystości**, czyli czynnik przesuwający ciało w stronę położenia równowagi, jeśli wcześniej zostanie ono z niego wychylone;
- **siłę tłumiącą**, czyli czynnik zmniejszający z czasem zakres drgań;
- **siłę wymuszającą**, czyli siłę wychylającą ciało z położenia równowagi.

Pierwsze dwa z wymienionych pojęć pojawiają się w opisie ruchu każdego oscylatora, natomiast dwa następne opisują czynniki, które mogą towarzyszyć drganiom, ale nie są konieczne do ich występowania. Złożone układy pozornie pozostają poza zasięgiem analizy za pomocą przedstawionych tu pojęć, ale okazuje się, że zazwyczaj można je rozłożyć na czynniki, dla których podane powyżej terminy mają już znaczenie (tak zwane modelowanie). Przykładem jest oddawanie natury liny za pomocą układu połączonych ze sobą sprężyn.

Do opisu zjawisk wynikających z istnienia siły sprężystości, tłumienia i wymuszania niezbędne jest zrozumienie pojęcia **amplitudy**. Amplituda drgań to termin, którym określa się maksymalne wychylenie z położenia równowagi. Amplituda to na przykład początkowe wychylenie huśtawki z położenia równowagi, do którego przeciągasz ją, by rozpocząć jej ruch.

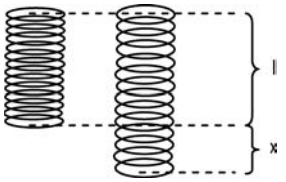
Siła sprężystości powoduje, że położenie ciała zmienia się z czasem, ale jej wartość nie zmienia amplitudy (maksymalnego przemieszczenia). Amplituda zależy wyłącznie od ilości energii, jaką dysponuje układ. Siła tłumiąca sprawia, że układ traci energię, co w efekcie prowadzi do zmniejszenia amplitudy drgań. To właśnie spotyka huśtawkę, którą wychylisz z położenia równowagi i której pozwolisz poruszać się pod wpływem tego jednokrotnego dostarczenia energii. Siła wymuszająca dostarcza energię do układu, zatem może zwięk-

szyć amplitudę drgań. Gdy w układzie działa zarówno siła tłumiąca, jak i siła wymuszająca, możliwe jest osiągnięcie stanu równowagi, w którym ilość traconej energii będzie natychmiast kompensowana dostarczaną energią. W takim przypadku amplituda drgań nie ulegnie zmianie tak długo, jak długo będzie działać siła wymuszająca. Innymi słowy, huśtawka będzie się huśtać tak długo, jak długo co jakiś czas pchniesz ją dodatkowo z odpowiednią siłą.

Prawo Hooke'a

Do opisu większości układów drgających można zastosować prawo Hooke'a (nazwane tak na cześć jego odkrywcy — Roberta Hooke'a).

Prawo Hooke'a jest stosunkowo proste. Wyjaśnimy je na przykładzie sprężyny, ponieważ takim układem posłużył się jego odkrywca. Spójrz na rysunek 8.1, gdzie przedstawiliśmy sprężynę w stanie nieodkształconym (o długości *l*) zamocowaną na jednym z końców oraz sprężynę, która została rozciągnięta o długość *x*, przez co osiągnęła wymiar *l*+*x*.



Rysunek 8.1. Na rozciągniętą sprężynę działa siła proporcjonalna do wydłużenia x

Prawo Hooke'a stwierdza, że sprężyna została rozciągnięta siłą o wartości *F* równą:

$$F = -k \cdot x.$$

Oznacza to, że siła rozciągająca jest proporcjonalna do wydłużenia *x*. Stałą proporcjonalności *k* pojawiającą się w tym równaniu określamy mianem **stałej sprężystości**. Jest ona miarą sztywności sprężyny. Im większa będzie wartość stałej *k*, tym większa siła będzie wiązać się z danym odkształceniem, zatem sprężyna będzie ściągana z większą siłą po tym, jak zostanie odkształcona.

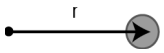
Znak minus sygnalizuje, że wektor siły jest zwrócony w przeciwnym kierunku niż wektor przemieszczenia. Zatem jeśli sprężyna zostanie ściśnięta, siła sprężystości będzie usiłowała doprowadzić do jej rozprężenia.

Wektorowo równanie to zapisuje się następująco, przyjmując **r** za wektor przemieszczenia swobodnego końca sprężyny:

$$\mathbf{F} = -k \cdot \mathbf{r}.$$

W większości zagadnień tego rozdziału nie będziemy poświęcać większej uwagi samej sprężynie (a i Ty zapewne szybko dojdziesz do wniosku, że jest ona mało interesującym obiektem obserwacji), skupimy się za to na opisie ruchu cząstki zaczepionej na końcu sprężyny. W jaki sposób będzie się ona poruszać? Wszyscy wiedzą, że zacznie drgać wokół punktu równowagi.

W zasadzie moglibyśmy pozbyć się z układu sprężyny i rozważać po prostu efekt działania siły sprężystości na cząstkę. Tak właśnie postąpimy w wielu przypadkach. Wtedy wektor **r** z prawa Hooke'a będzie po prostu wychyleniem z punktu równowagi, jakiego dozna drgająca cząstka zgodnie z tym, co przedstawiono na rysunku 8.2.



Rysunek 8.2. Cząstka drgająca wokół położenia równowagi

Zanim zakończymy ten podrozdział, chcielibyśmy podkreślić jeden istotny aspekt związany z prawem Hooke'a. Nie wolno zakładać, że podlegają mu wszystkie układy drgające; wiele z nich rządzi się innymi prawami. Na szczęście bardzo często prawo Hooke'a wypełnia swoją funkcję, nawet jeśli daje jedynie przybliżony opis zjawiska, dlatego przez pozostałą część rozdziału będziemy zajmować się właśnie nim.

Drgania swobodne

Zaczniemy od zbudowania modelu **drgań swobodnych**. Drgania tego rodzaju pojawiają się w układach drgających wyłącznie pod wpływem działania siły sprężystości. Oczywiście aby w układzie pojawił się tego rodzaju ruch, na ciało musi najpierw zadziałać pewna siła, która wychyli je z położenia równowagi. Nas będzie interesować to, co stanie się z ciałem po tym, jak początkowa siła przestanie na nie działać, gdy układ zostanie pozostawiony sam sobie.

Funkcja wyznaczająca siłę sprężystości

Zaczniemy od przygotowania nowej funkcji klasy `Forces`, której zadaniem będzie obliczanie siły sprężystości. Kod tej prostej funkcji — nazwijmy ją `spring()` — znajdziesz poniżej:

```
static public function spring(k:Number,r:Vector2D):Vector2D {
   return r.multiply(-k);
}
```

Funkcja `spring()` przyjmuje dwa argumenty — wartość stałej sprężystości `k` i wektor przesunięcia `r` (to wektor, który pojawił się w poprzednim wzorze). Wynikiem jej działania jest wektor siły sprężystości, $\mathbf{F} = -k \cdot \mathbf{r}$.

Tak przygotowaną funkcję wykorzystamy do napisania prostego oscylatora.

Przygotowanie prostego oscylatora

Wiesz już, co będzie dalej, prawda? Poniższy kod tworzy ciało, które postaramy się wprawić w drgania. Będzie to obiekt `object` klasy `Ball`. Nie będziemy wprowadzać do animacji sprężyny, natomiast umieścimy w niej jeszcze jedną instancję klasy `Ball`, tak zwany `attractor`, która będzie się znajdować w położeniu równowagi. Potem przekażemy dane obiektów `object` i `attractor` do konstruktora obiektu `BasicOscillator` odpowiedzialnego za wywoływanie drgań. Zarówno `object`, jak i `attractor` mają zerowe prędkości początkowe i są oddalone od siebie o pewną odległość, dzięki czemu współrzędne ich wektorów położenia, podane jako dane typu `pos2D`, są inne.

```
package{
   import flash.display.Sprite;
   import com.physicscodes.objects.Ball;
   import com.physicscodes.math.Vector2D;
   import com.physicscodes.objects.Particle;

   public class BasicOscillations extends Sprite{
      public function BasicOscillations():void{
```

```
            init();
        }
        private function init():void{
            // Tworzy ciało.
            var object:Ball;
            object = new Ball(15,0x0000cc,1);
            object.pos2D = new Vector2D(100,50);
            object.velo2D=new Vector2D(0,0);
            addChild(object);

            // Tworzy sprężynę.
            var attractor:Ball;
            attractor=new Ball(2,0x000000);
            attractor.pos2D=new Vector2D(275,200);
            attractor.velo2D=new Vector2D(0,0);
            addChild(attractor)

            // Sprawia, że układ ciało – sprężyna zaczyna się poruszać.
            var oscillator:BasicOscillator=new BasicOscillator(object,attractor);
            oscillator.startTime(10);
        }
    }
}
```

A oto kod obiektu `BasicOscillator`:

```
package {
    import com.physicscodes.motion.Forcer;
    import com.physicscodes.motion.Forces;
    import com.physicscodes.objects.Ball;
    import com.physicscodes.math.Vector2D;

    public class BasicOscillator extends Forcer{
        private var _object:Ball;
        private var _center:Vector2D;
        private var _displ:Vector2D;
        private var _kSpring:Number=1;

        public function BasicOscillator(pobject:Ball,pattractor:Ball):void{
            _object = pobject;
            _center = pattractor.pos2D;
            super(_object);
        }
        override protected function calcForce():void{
            _displ=_object.pos2D.subtract(_center);
            force = Forces.spring(_kSpring,_displ);
        }
    }
}
```

Żadne z wprowadzonych tu rozwiązań nie powinno być Ci obce. Jak zwykle nadpisujemy metodę `calcForce()` tak, by obliczała wektor położenia ciała względem atraktora, a następnie na tej podstawie wyznaczyła wektor siły sprężystości, z jaką atraktor działa na ciało. Zmienna `_kSpring` to oczywiście stała sprężystości *k*.

Uruchom skrypt, a przekonasz się, że ciało oscyluje wokół punktu równowagi tak, jak się tego spodziewaliśmy. Zmień teraz stałą sprężystości z 1 na 10 — w ten sposób usztywnisz sprężynę. Gdy teraz uruchomisz

kod, zobaczysz, że kulka oscyluje szybciej. Jeśli chcesz, by ciało poruszało się z inną amplitudą, zmień jego położenie początkowe w pliku *BasicOscillations.as*.

Teraz zmień znów wartość stałej *k* na 1, a następnie przypisz ciału prędkość początkową (200, 0):

```
object.velo2D=new Vector2D(200,0);
```

Po uruchomieniu skryptu powinieneś zobaczyć ciało poruszające się po owalnej orbicie wokół dawnego punktu równowagi. Przypomina to nieco układ grawitacyjny znany Ci z rozdziału 6. Siła sprężystości, tak samo jak grawitacja, zawsze działa w kierunku określonego punktu. Różnica polega na tym, że siła grawitacji **maleje** wraz ze wzrostem odległości, natomiast siła sprężystości **rośnie** z odległością. Dzieje się tak, ponieważ grawitacja jest proporcjonalna do $1/r^2$, a siła sprężystości jest wprost proporcjonalna do *r*, gdzie *r* jest odległością od źródła przyciągania.

Prosty ruch harmoniczny

Rodzaj drgań, który właśnie uzyskaliśmy w symulacji, to tak zwane **proste drgania harmoniczne**. Ciało porusza się w ten sposób, jeśli działa na nie jedynie siła sprężystości, a tak właśnie było w przytoczonym przykładzie.

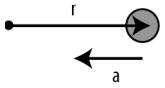
Ponieważ w układzie drgającym prostym ruchem harmonicznym jedyną działającą siłą jest siła sprężystości, $\mathbf{F} = -k\cdot\mathbf{r}$, to zgodnie z drugą zasadą dynamiki Newtona musi ona być siłą wypadkową $\mathbf{F} = m\cdot\mathbf{a}$. Zatem możemy zapisać:

$$m\cdot\mathbf{a} = -k\cdot\mathbf{r}.$$

Po wykonaniu obustronnego dzielenia przez m otrzymamy:

$$\mathbf{a} = -\frac{k}{m}\cdot\mathbf{r}\ .$$

W powyższym równaniu *m* jest masą drgającej cząstki, *k* jest stałą sprężystości, zatem wyrażenie *k*/*m* ma wartość stałą. Oznacza to, że przyśpieszenie drgającej cząstki jest proporcjonalne do wektora przemieszczenia tej cząstki względem punktu równowagi. Stała równowagi ma wartość ujemną, zatem wektor przyśpieszenia jest zwrócony zawsze przeciwnie do wektora przesunięcia (zawsze wskazuje na punkt równowagi, ponieważ wektor przemieszczenia jest zwrócony zawsze w stronę cząstki). Sytuację tę ilustruje rysunek 8.3.



Rysunek 8.3. Wektor przyśpieszenia oscylatora harmonicznego jest zawsze zwrócony przeciwnie do wektora przemieszczenia ciała

Pamiętasz zagadnienia dotyczące pochodnych, które wprowadziliśmy w rozdziale 3.? Wspominaliśmy wtedy, że przyśpieszenie jest drugą pochodną przemieszczenia po czasie, co wyraża się następująco:

$$\mathbf{a} = \frac{d^2\mathbf{r}}{dt^2}\ .$$

Zatem możemy napisać równanie oscylatora harmonicznego $\mathbf{a} = -\frac{k}{m}\cdot\mathbf{r}$ w postaci:

$$\frac{d^2\mathbf{r}}{dt^2} = -\frac{k}{m}\cdot\mathbf{r}\ .$$

Wykazaliśmy właśnie, że równanie oscylatora harmonicznego jest równaniem różniczkowym drugiego rzędu, takim, jakie opisywaliśmy w rozdziale 5. Zatem obiekt klasy `BasicOscillator` rozwiązuje równania różniczkowe drugiego rzędu za pomocą metody Eulera (opisanej także w rozdziale 5.). Przypominamy, że klasa `BasicOscillator` rozszerza klasę `Forcer`, która z kolei jest rozszerzeniem klasy `Mover`. Klasa `Forcer` przeprowadzała całkowanie przyśpieszenia, by uzyskać wartość prędkości (odpowiedzialna za to była metoda `updateVelo()`), a klasa `Mover` całkowała prędkość, by w ten sposób określić przemieszczenie (metoda `move` ↪`Object()`). W obydwu przypadkach zastosowaliśmy numeryczną metodę całkowania zaproponowaną przez Eulera.

Jednocześnie podane wcześniej równanie ma rozwiązanie analityczne opisujące wzorem przemieszczenie w funkcji czasu. Rozwiązanie tego rodzaju zadania wymaga znajomości rachunku różniczkowo-całkowego na poziomie akademickim, więc tu po prostu podamy Ci gotową funkcję:

$$\mathbf{r} = \mathbf{A}\cdot\cos(\omega\cdot t)+\mathbf{B}\cdot\sin(\omega\cdot t),$$

gdzie **A** i **B** są stałymi wektorami odpowiadającymi warunkom początkowym zadanym w problemie, a ω jest częstością kątową ruchu drgającego (rozdział 3.).

Ostatecznie doszliśmy do wniosku, że prosty ruch harmoniczny jest sumą dwóch funkcji — sinus i cosinus. Nie powinno stanowić to dla Ciebie zaskoczenia, ponieważ ruch harmoniczny jest podstawowym ruchem drgającym, a — co wiesz z rozdziału 3. — sinus i cosinus to funkcje opisujące drgania. Wartość ω określa częstotliwość drgań, a zatem również ich okres, natomiast wartości *A* i *B* definiują amplitudę drgań (maksymalne wychylenie z położenia równowagi).

Dla każdego, kto zna choćby podstawy rachunku różniczkowego, zapisanie wektora prędkości ciała zgodnie z definicją $\mathbf{v} = d\mathbf{r}/dt$ nie będzie stanowić najmniejszego problemu:

$$\mathbf{v} = -\omega\cdot\mathbf{A}\cdot\sin(\omega\cdot t)+\omega\cdot\mathbf{B}\cdot\cos(\omega\cdot t).$$

Jakie wartości przyjmują *A*, *B* i ω? Wektory *A* i *B* są zależne od warunków początkowych, jakie wprowadzisz w problemie — zależą od początkowego przemieszczenia i prędkości początkowej ciała (to wektory **r** i **v** w chwili $t = 0$). Wprowadźmy odpowiednie oznaczenia: $\mathbf{r}_0$ na początkowe położenie ciała i $\mathbf{v}_0$ na jego prędkość początkową. Przy założeniu, że $t = 0$, z podanych przed chwilą równań otrzymamy:

$$\mathbf{r}_0 = \mathbf{A}$$

oraz

$$\mathbf{v}_0 = \omega\cdot\mathbf{B},$$

ponieważ $\cos(0) = 1$, a $\sin(0) = 0$.

Stąd natychmiast wyznacza się $\mathbf{A} = \mathbf{r}_0$ i $\mathbf{B} = \mathbf{v}_0/\omega$.

Gdybyś rozwiązał to równanie różniczkowe, otrzymałbyś także wzór opisujący częstość kątową ciała ω:

$$\omega = \sqrt{\frac{k}{m}}\ .$$

Wielkość tę nazywa się **częstością własną** układu. Gdy do układu drgającego wprowadzimy początkowe zaburzenie (wychylające ciało z położenia równowagi), a potem pozostawimy układ własnemu losowi, będzie on drgać właśnie z tą częstotliwością. Jej zmiana jest możliwa wyłącznie pod wpływem dodatkowej siły, na przykład tłumienia.

Przypomnij sobie (znów rozdział 3.), że $\omega = 2 \cdot \pi \cdot f$ i $f = 1/T$, gdzie f jest częstotliwością (liczbą drgań wykonywanych w jednej sekundzie), a T okresem drgań (czasem potrzebnym na wykonanie pełnego cyklu drgań). Wiedząc to, możesz zapisać wzory wyrażające częstotliwość drgań i okres drgań w zależności od wartości parametrów układu k i m:

$$f = \frac{1}{2 \cdot \pi} \cdot \sqrt{\frac{k}{m}},$$

$$T = 2 \cdot \pi \cdot \sqrt{\frac{m}{k}}.$$

Z powyższych wzorów wynika, że zwiększając sztywność sprężyny, k, zwiększysz też częstotliwość drgań, przez co spadnie ich okres (ciało będzie drgać szybciej). Zwiększając masę cząstki, wywołasz odwrotny efekt — ciało będzie drgać wolniej. Wprowadź odpowiednie zmiany w kodzie i przekonaj się o tym sam!

Częstotliwość drgań i okres zależą wyłącznie od tych dwóch parametrów — ani położenie początkowe, ani prędkość ciała nie mają wpływu na drgania. Mogłoby się wydawać, że ciało wychylone z położenia równowagi na większą odległość będzie potrzebowało więcej czasu na wykonanie pełnego cyklu drgań, ale w przypadku ruchu harmonicznego prostego jest inaczej. Im dalej znajdzie się ono początkowo, tym większego będzie doświadczać przyśpieszenia, zatem jego średnia prędkość również będzie wyższa, co skompensuje większą odległość. Nie wierzysz? Spróbuj zmierzyć czas drgań stoperem lub odmierz czas w animacji.

W zasadzie możemy zrobić coś jeszcze lepszego — przygotujemy odpowiedni wykres.

Drgania a dokładność obliczeń numerycznych

Zmienimy nieco kod zawarty w pliku *BasicOscillations.as*: przesuniemy drgające ciało w nowe położenie.

```
object.pos2D = new Vector2D(100,50);
```

Zmodyfikujemy też wiersz:

```
var oscillator:BasicOscillator=new BasicOscillator(object,attractor);
```

w następujący sposób:

```
var oscillator:FreeOscillator=new FreeOscillator(object,attractor);
```

Zmienimy też nazwę klasy i konstruktora z `BasicOscillations` na `FreeOscillations`, a potem zapiszemy w nowym pliku *FreeOscillations.as*.

Pora przygotować plik *FreeOscillator.as*. W tym celu zmodyfikujemy zawartość pliku *BasicOscillator.as*. Chcielibyśmy, by zmieniony skrypt przygotowywał wykres zależności przemieszczenia ciała od czasu ruchu.

Oto pełny kod skryptu *FreeOscillations.as* ze zmianami zaznaczonymi pogrubieniem:

```
package {
   import com.physicscodes.motion.Forcer;
   import com.physicscodes.motion.Forces;
```

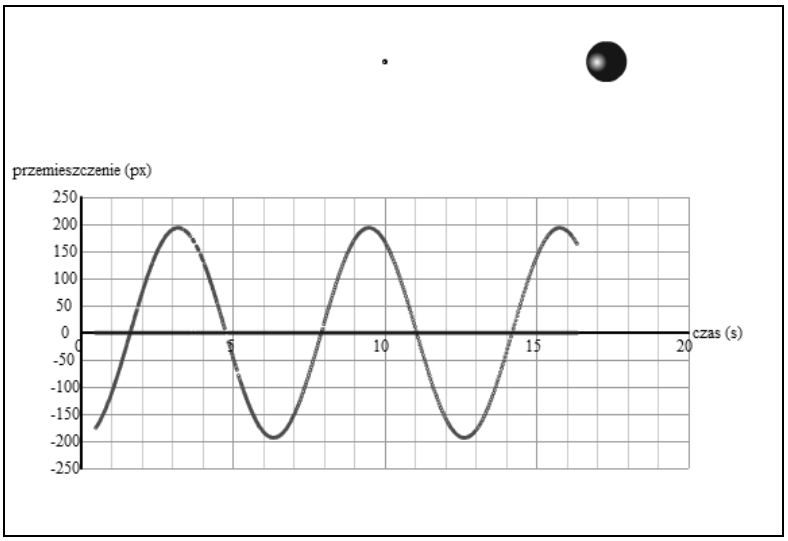
```
import com.physicscodes.objects.Ball;
import com.physicscodes.math.Vector2D;
import com.physicscodes.math.Graph;

public class FreeOscillator extends Forcer{
   private var _object:Ball;
   private var _center:Vector2D;
   private var _displ:Vector2D;
   private var _kSpring:Number=1;
   private var _graph:Graph;

   public function FreeOscillator(pobject:Ball,pattractor:Ball):void{
      _object = pobject;
      _center = pattractor.pos2D;
      setupGraph();
      super(_object);
   }

   override protected function moveObject():void{
      super.moveObject();
      plotGraph();
   }

   override protected function calcForce():void{
      _displ=_object.pos2D.subtract(_center);
      force = Forces.spring(_kSpring,_displ);
   }

   // Tworzy wykres zależności przemieszczenia od czasu.
   private function setupGraph():void {
      _graph= new Graph(0,20,-250,250,50,250,450,200);
      _graph.drawgrid(5,1,50,50);
      _graph.drawaxes('czas (s)','przemieszczenie (px)');
      _object.stage.addChild(_graph);
   }
   private function plotGraph():void{
      _graph.plot([time], [_displ.x], 0xff0000, false, true);
      _graph.plot([time], [_displ.y], 0x0000ff, false, true);
   }
}
}
```

Dodaliśmy do skryptu instrukcje importujące klasę `Graph`, zadeklarowaliśmy nową zmienną prywatną `_graph` jako obiekt klasy `Graph` i wewnątrz konstruktora wywołaliśmy nową prywatną metodę `setupGraph()`. Na koniec nadpisaliśmy metodę `moveObject()` tak, by w każdym kroku wywoływała nową prywatną metodę `plotGraph()`.

Metoda `setupGraph()` jest odpowiedzialna za utworzenie obiektu `_graph` klasy `Graph` i umieszczenie go na scenie.

Wywoływana w każdym kroku obliczeń metoda `plotGraph()` wywołuje publiczną metodę klasy `Graph` — `plot()` — której zadaniem jest umieszczenie na wykresie przemieszczenia współrzędnych ciała mierzonych względem punktu równowagi.

Symulacja powinna dać taki efekt, jaki przedstawiliśmy na rysunku 8.4. Przemieszczenie ciała w poziomie z pewnością jest opisane funkcją sinus, co zgadza się z wnioskami, do jakich doszliśmy w poprzednim podrozdziale. Przemieszczenie pionowe jest równe zero, ponieważ taki warunek początkowy znalazł się w kodzie. Łatwo zmienić ten stan rzeczy — wystarczy przypisać prędkości ciała również pionową składową lub podać inne położenie początkowe w pionie. Gdy to zrobisz, przekonasz się, że przemieszczenie w pionie również opisane jest funkcją sinus. Możesz też spróbować zmienić wartości k i m oraz początkowe położenie ciała, by sprawdzić, czy informacje dotyczące częstotliwości drgań i okresu, które podaliśmy w poprzednim podrozdziale, są prawdziwe — czy zmieniają się wraz z tymi parametrami, czy nie.



Rysunek 8.4. Przemieszczenie drgającego ciała jako funkcja czasu

Sprawdzimy teraz, na ile dokładne są wyniki tak przeprowadzonych obliczeń. Porównamy je z wartościami przemieszczenia drgającego ciała wyznaczanymi na podstawie rozwiązania analitycznego równania ruchu, które podaliśmy w poprzednim podrozdziale.

W tym celu na podstawie skryptu *FreeOscillations.as* przygotujemy skrypt *FreeOscillations2.as*, w którym zmienimy prędkość początkową drgającego ciała:

```
object.velo2D=new Vector2D(0,50);
```

Potem zmodyfikujemy wiersz:

```
var oscillator:FreeOscillator=new FreeOscillator(object,attractor);
```

w następujący sposób:

```
var oscillator:FreeOscillator2=new FreeOscillator2(object,attractor);
```

Klasa `FreeOscillator2` jest po prostu zmienioną wersją klasy `FreeOscillator`. Dodaliśmy do niej kod, który na podstawie znanych Ci już zależności — $\omega = \sqrt{k/m}$, $\mathbf{A} = \mathbf{r}_0$ i $\mathbf{B} = \mathbf{v}_0/\omega$ — oblicza wartości wektorów **A** i **B** oraz stałej ω (zdefiniowanych odpowiednio jako zmienne prywatne `_A`, `_B` i `_omega`). Obliczenia te wykonywane są w konstruktorze klasy `FreeOscillator2`. Odpowiedzialne za to są wprowadzone instrukcje:

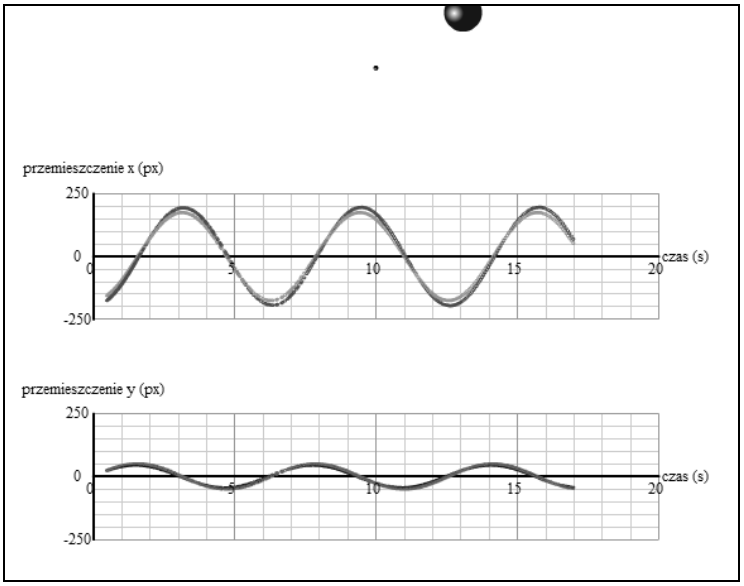
```
_omega = Math.sqrt(_kSpring/_object.mass);
_A = _object.pos2D.subtract(_center);
_B = _object.velo2D.multiply(1/_omega);
```

Metoda `plotGraph()` została zmodyfikowana w taki sposób, by umieszczała na wykresach także wyniki rozwiązań analitycznych równania $\mathbf{v} = -\omega \cdot \mathbf{A} \cdot \sin(\omega \cdot t) + \omega \cdot \mathbf{B} \cdot \cos(\omega \cdot t)$. Zadanie to realizuje kod:

```
var r:Vector2D =
  _A.multiply(Math.cos(_omega*time)).add(_B.multiply(Math.sin(_omega*time)));
_graphX.plot([time], [r.x], 0x00ff00, false, true);
_graphY.plot([time], [r.y], 0xff00ff, false, true);
```

Kolejną istotną zmianą w stosunku do wykorzystywanej poprzednio klasy `FreeOscillator` jest pojawienie się drugiego wykresu — obiektu klasy `Graph`. W tej wersji w kodzie znajdują się deklaracje dwóch obiektów tej klasy: `_graphX` oraz `_graphY` (tworzone przez odpowiednio zmodyfikowaną metodę `setupGraph()`). Na pierwszym z nich umieszczamy zmiany współrzędnej *x* położenia ciała w czasie, na drugim — analogiczne wartości współrzędnej *y*. Przygotowanie wykresów leży w gestii funkcji `plotGraph()`.

Po uruchomieniu skryptu na scenie pojawi się ciało okrążające położenie równowagi po wydłużonej orbicie, natomiast na dwóch widocznych poniżej wykresach zaczną odkładać się wartości przemieszczenia ciała w pionie i w poziomie. Na każdym wykresie pojawiają się dwie funkcje, z których jedna odpowiada rozwiązaniu numerycznemu, druga zaś — analitycznemu. Jak widzisz, punkty w obydwu przypadkach leżą dość blisko siebie, czasami wręcz na siebie zachodzą (rysunek 8.5). W tym przypadku całkowanie metodą Eulera sprawdza się całkiem nieźle.



Rysunek 8.5. Porównanie rozwiązań numerycznego i analitycznego równania ruchu oscylatora harmonicznego

Jeśli jednak zmienisz wartość zmiennej `_kSpring` na przykład na `10` (sprężyna stanie się wtedy sztywniejsza, a ciało zacznie poruszać się szybciej), przekonasz się, że różnica pomiędzy wynikami uzyskiwanymi numerycznie i analitycznie znacznie rośnie — to znak, że metoda Eulera przestaje się sprawdzać. Gdy zmienisz

wartość zmiennej `_kSpring` na `100`, metoda Eulera da zupełnie bzdurne wyniki, przez co ciało zacznie pojawiać się w nieodpowiednich miejscach w niewłaściwym czasie. Teraz chyba rozumiesz już, dlaczego staranne dobranie metody całkującej jest tak istotne, szczególnie w przypadku ruchu drgającego — niepoprawnie wybrany schemat obliczeniowy może zniszczyć całą symulację. W rozdziale 14. przedstawimy rozwiązanie tego problemu, a na razie postaramy się unikać zbyt dużych wartości stałej sprężystości.

Drgania tłumione

Poprzednio przedstawiliśmy model drgań, w którym ruch trwał w nieskończoność. W rzeczywistości nie zdarza się to nazbyt często. Większość układów drgających doznaje **tłumienia**. Oznacza to, że amplituda drgań stopniowo spada, aż ruch ustanie całkowicie w wyniku wyprowadzenia energii z układu. Mamy tu do czynienia ze zjawiskiem analogicznym do powstawania oporu, który rozpraszał energię kinetyczną poruszającego się ciała i w efekcie doprowadzał do jego zatrzymania. Aby wprowadzić do symulacji tłumienie, musimy zdefiniować odpowiednią siłę zmniejszającą amplitudę drgań.

Siła tłumiąca

Siłę tłumiącą definiuje się zazwyczaj jako czynnik proporcjonalny do prędkości poruszającego się ciała. Oznacza to, że w każdej chwili trwania ruchu jest ona opisana następującym równaniem:

$$\mathbf{F} = -c \cdot \mathbf{v},$$

gdzie c jest stałą tłumienia, a znak minus wskazuje na przeciwny w stosunku do prędkości zwrot wektora siły.

Zauważ, że siła ta ma dokładnie taki sam charakter jak siła oporu aerodynamicznego dla niewielkich prędkości. Oznacza to, że tłumienie drgań oscylatora można uzyskać, korzystając z definicji funkcji `linearDrag()`. Jednak w fizyce opór powietrza, choć również może powodować tłumienie drgań, nie jest jedynym rodzajem siły rozpraszającej energię oscylatora. Opór to siła, z jaką płyn przeciwstawia się ruchowi zanurzonego w nim ciała, natomiast tłumienie drgań może być wywoływane nie tylko przez czynniki zewnętrzne (na przykład opór ośrodka czy tarcie zewnętrzne), lecz także przez czynniki wewnętrzne (tak zwane tarcie wewnętrzne związane z właściwościami cząsteczkowymi materiału, z którego wykonano sprężynę). Co więcej, tłumienie może pojawić się w wyniku występowania sił wewnętrznych i jednoczesnego oporu ośrodka (oczywiście każda z sił byłaby wtedy opisana innym współczynnikiem). Przykładem mogą być drgania masy zawieszonej na sprężynie odbywające się w powietrzu lub w wodzie — ruch będzie tłumiony zarówno w wyniku działania sił wytwarzających się wewnątrz sprężyny, jak i w wyniku oporu ośrodka. Dlatego wolimy utworzyć osobną funkcję, która będzie odpowiedzialna za obliczanie tłumienia. Jak wynika z poniższego listingu, to odpowiednik funkcji `linearDrag()`:

```
static public function damping(c:Number,vel:Vector2D):Vector2D {
   var force:Vector2D;
   var velMag:Number=vel.length;
   if (velMag>0) {
      force=vel.multiply(-c);
   }else {
      force=new Vector2D(0,0);
   }
   return force;
}
```

Skutek tłumienia drgań

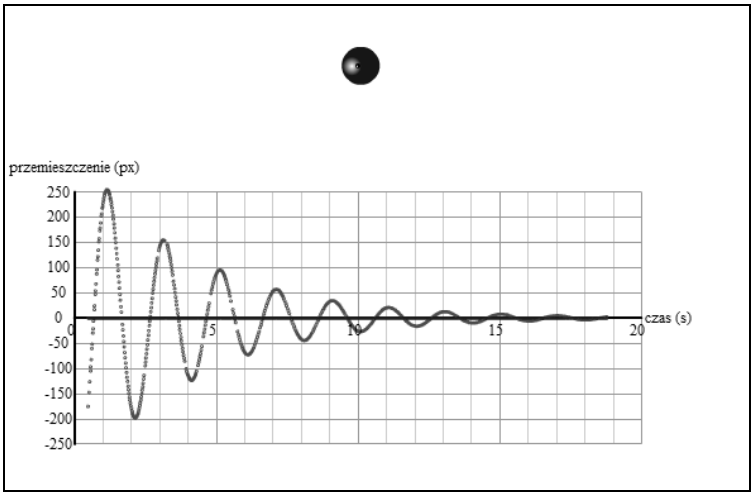
Zmienimy teraz pliki *FreeOscillations.as* i *FreeOscillator.as* tak, by umieścić w nich klasy odpowiedzialne za wprowadzenie do układu siły tłumiącej. Nowe pliki będą nosić nazwy *DampedOscillations.as* i *DampedOscillator.as*. Skrypt z pliku *DampedOscillations.as* różni się od zawartości pliku *FreeOscillations.as* jedynie nazwą klasy oraz konstruktora odpowiedzialnego za utworzenie instancji klasy *DampedOscillator*. Skrypt *DampedOscillator.as* wprowadza nową siłę — tłumienie — zatem główną różnicą pomiędzy nim a *FreeOscillator.as* jest deklaracja współczynnika tłumienia `_cDamping` i przypisanie mu wartości:

```
private var _cDamping:Number=0.5;
```

Drugą istotną zmianą jest wprowadzenie siły tłumiącej do metody `calcForce()` i dodanie jej do siły sprężystości, by w ten sposób wyznaczyć ich wypadkową.

```
override protected function calcForce():void{
   _displ=_object.pos2D.subtract(_center);
   var restoring:Vector2D = Forces.spring(_kSpring,_displ);
   var damping:Vector2D = Forces.damping(_cDamping,_object.velo2D);
   force = Forces.add([restoring, damping]);
}
```

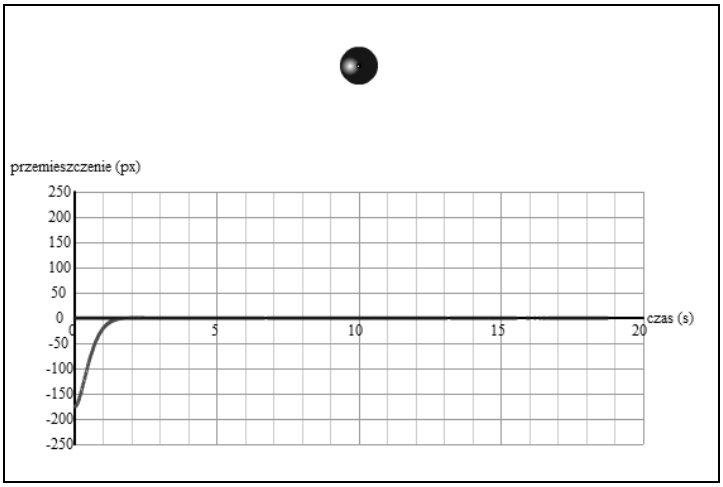
To już wszystko. Po uruchomieniu skryptu z parametrami `_kSpring = 10` i `_cDamping = 0.5` powinieneś otrzymać wynik zbliżony do przedstawionego na rysunku 8.6. Z upływem czasu amplituda drgań ciągle spada, aż w końcu ruch zamiera zupełnie i ciało zatrzymuje się w położeniu równowagi.



Rysunek 8.6. Drgania tłumione

W tym momencie warto zatrzymać się nieco nad przedstawionym przykładem i przeprowadzić kilka testów dla różnych wartości stałej tłumienia `_cDamping`. Gdy przypiszesz jej mniejszą wartość, okaże się, że drgania trwają dłużej, natomiast większa wartość wytłumi je szybciej. Można by przypuszczać, że większe wartości stałej *c* spowodują szybsze tłumienie drgań, tymczasem okazuje się, że istnieje pewna jej krytyczna wartość, dla której drgania ustają w najkrótszym czasie — w zasadzie trudno wtedy mówić o drganiach, bo ciało po prostu dociera do punktu równowagi i nie przemieszcza się dalej. To tak zwane **tłumienie**

krytyczne. Wykres zależności przemieszczenia od czasu dla takiej wartości stałej *c* przedstawia rysunek 8.7. Dla sprężyny opisanej stałą sprężystości `_kSpring = 10` wartość krytyczna stałej tłumienia `_cDamping` wynosi około 5,5. Ciało osiąga wtedy położenie równowagi po około jednej sekundzie. Jeśli zwiększysz stałą tłumienia powyżej tej wartości — na przykład do 10 czy 20 — przekonasz się, że ruch ciała trwa dłużej. Wynika to stąd, że większe tłumienie wydłuża jednocześnie sam ruch i choć ciało nie przekracza punktu równowagi, potrzebuje więcej czasu, by do niego dotrzeć. Zjawisko tłumienia krytycznego znalazło praktyczne zastosowanie w mechanizmach montowanych w drzwiach wahadłowych.



Rysunek 8.7. Tłumienie krytyczne

Analityczne rozwiązanie równania ruchu drgającego z tłumieniem

Ten krótki podrozdział powstał z myślą o wszystkich entuzjastach. Jeśli nie interesują Cię wzory, możesz spokojnie przejść dalej. Chcemy przedstawić tu rozwiązanie analityczne, by pokazać, jak zmieniają się wyniki uzyskane poprzednio dla oscylatora harmonicznego po wprowadzeniu do układu drgań. Sprawdzimy też, jak symulacja radzi sobie z opisem rzeczywistości. W efekcie ta część rozdziału ma charakter przede wszystkim edukacyjny.

Druga zasada dynamiki Newtona dla oscylatora, na który działa siła sprężystości wyrażona jako $\mathbf{F} = -k\cdot\mathbf{r}$ oraz siła tłumiąca $\mathbf{F} = -c\cdot\mathbf{v}$, wyraża się jako:

$$m\cdot\mathbf{a} = -k\cdot\mathbf{r} - c\cdot\mathbf{v},$$

co w postaci różniczkowej przedstawia się jako:

$$\frac{d^2\mathbf{r}}{dt^2} = -\frac{k}{m}\cdot\mathbf{r} - \frac{c}{m}\cdot\frac{d\mathbf{r}}{dt}.$$

Rozwiązaniem analitycznym tego równania jest wektor wyrażony jako:

$$\mathbf{r} = \left[\mathbf{A}\cdot\cos\left(\omega_d\cdot t\right)+\mathbf{B}\cdot\sin\left(\omega_d\cdot t\right)\right]\cdot\exp\left(-\gamma\cdot\omega_0\cdot t\right),$$

gdzie stałe γ, ω_0, ω_d, **A** i **B** to kolejno:

$$\gamma = \frac{c}{2\cdot\sqrt{m\cdot k}},$$

$$\omega_0 = \sqrt{\frac{k}{m}},$$

$$\omega_d = \omega_0\cdot\sqrt{1-\gamma^2},$$

$$\mathbf{A} = \mathbf{r}_0,$$

$$\mathbf{B} = \frac{\left(\mathbf{v}_0+\gamma\cdot\omega_0\cdot\mathbf{r}_0\right)}{\omega_d}.$$

W rozwiązaniu pojawił się niewystępujący poprzednio czynnik wykładniczy odpowiedzialny za tłumienie drgań sinusoidalnych (więcej w rozdziale 3.). Parametr γ to czynnik związany z tłumieniem. Gdy nie ma tłumienia, współczynnik $c = 0$, więc i $\gamma = 0$, a rozwiązanie redukuje się do znanej Ci już postaci. Zmieniła się też częstość kątowa drgań, która teraz przyjmuje wartość ω_d (pojawia się w obydwu funkcjach trygonometrycznych) mniejszą od częstości własnej układu ω_0, z jaką mieliśmy do czynienia w przypadku oscylatora harmonicznego prostego (wtedy oznaczaliśmy ją symbolem ω).

Wprowadziliśmy te równania do zmienionej wersji pliku *FreeOscillator2.as*, którą nazwaliśmy *DampedOscillator2.as*. Jeśli chcesz, zapoznaj się z zawartym tam kodem skryptu. Gdy uruchomisz powiązany z nim skrypt *DampedOscillations2.as*, przekonasz się, że pomiędzy wynikami numerycznymi i analitycznymi pojawia się pewna rozbieżność. Co więcej, prawdopodobnie uznasz, że symulacja przebiega zupełnie nieprawdopodobnie, dając wyniki niemające wiele wspólnego z rzeczywistością. Wynika to stąd, że metoda Eulera zupełnie nie sprawdza się podczas rozwiązywania tego rodzaju problemów, z wyjątkiem może najprostszych ich wariantów. W rozdziale 14. przedstawimy alternatywy, które pozwolą przeprowadzić poprawną symulację.

Drgania wymuszone

Drgania w układzie tłumionym z czasem ustają same, dlatego aby je podtrzymać, potrzebna jest dodatkowa siła. Siłę tę nazywamy **wymuszającą**.

Siła wymuszająca

Niemal każda znana Ci siła może stać się siłą wymuszającą, nie będziemy więc wprowadzać specjalnego oznaczenia, a po prostu opiszemy je wszystkie ogólnym symbolem **f**(t). W ten sposób zaznaczamy jedynie, że siła jest funkcją czasu, bez określania, z jakim prawem się wiąże.

Wymuszeniem może być na przykład działająca okresowo siła opisana wzorem **F** = **A**·cos(ω·t)+ **B**·sin(ω·t), w którym **A** i **B** są stałymi wektorami wyznaczającymi amplitudę siły (jej wartość maksymalną) i kierunek jej działania, natomiast ω jest częstością kątową wymuszania. Tego rodzaju model opisuje ruch drgający wywoływany przez oddziaływanie typu okresowego, na przykład pojawiające się co chwila podmuchy wiatru, które wprawiają w drgania most wiszący.

Zacznijmy od przykładu.

Przykład — okresowa siła wymuszająca

Podstawą do dalszej pracy będzie kod z pliku *DampedOscillator2.as*. Nie interesują nas na razie rozwiązania analityczne, więc pozbędziemy się części kodu zawartego w metodzie `plotGraph()`, ale zachowamy wiersze obliczające wartości zmiennych `_gamma`, `_omega0` i `_omegad` występujące w konstruktorze. Będziemy przechowywać w nich wartości stałych γ, ω_0 i ω_d. Po usunięciu zbędnych deklaracji zmiennych zmienimy nazwę klasy i konstruktora na `ForcedOscillator` i zapiszemy efekt pracy w pliku *ForcedOscillator.as*. Utworzymy też plik *ForcedOscillations.as* odpowiedzialny za uruchomienie animacji.

Pamiętaj, że zmienna `_omega0` (ω_0) to częstość własna nietłumionego układu drgającego, natomiast `_omegad` (ω_d) jest częstością kątową układu tłumionego. Zaczniemy od umieszczenia w metodzie `calcForce()` następujących wierszy:

```
var forcing:Vector2D =
  new Vector2D(200*Math.cos(2*_omegad*time)+200*Math.sin(2*_omegad*time), 0);
force = Forces.add([restoring, damping, forcing]);
```

Kod ten dodaje wektor siły opisanej wzorem $\mathbf{F} = \mathbf{A}\cdot\cos(\omega\cdot t) + \mathbf{B}\cdot\sin(\omega\cdot t)$, gdzie **A** i **B** mają wartość 200 w kierunku *x* (musimy używać dość dużych wartości, by wpływ wymuszenia na drgania był widoczny), a $\omega = 2\cdot\omega_d$. Oznacza to, że drgania w układzie będą wymuszane siłą zmieniającą się w czasie zgodnie z funkcją sinus, której częstość kątowa jest dwukrotnie większa od częstości układu tłumionego.

Uruchom skrypt, by przekonać się, że drgania w układzie również tym razem zaczynają zamierać, ale po pewnej chwili ciało zaczyna znów oscylować wokół punktu równowagi z częstością niemal dwukrotnie większą niż poprzednio i ze znacznie mniejszą amplitudą. Te drgania będą trwać w nieskończoność. Skutkiem działania sinusoidalnej siły wymuszającej jest wprowadzenie układu w drgania z częstością wymuszenia (kosztem amplitudy), mimo że układ „wolałby" drgać z częstością własną. Możesz przeprowadzić kilka testów, zmieniając częstość wymuszania. Jeśli na przykład użyjesz częstości o wartości $\omega_d/2$ (równej połowie częstości tłumienia), ciało będzie ostatecznie drgać z częstością równą połowie częstości własnej. Wypływa stąd wniosek, że ciało może drgać z częstością inną niż własna układu, ale odbywa się to kosztem amplitudy drgań.

Sprawdź jeszcze jeden szczególny przypadek wymuszania drgań, w którym częstość wymuszająca jest równa ω_d:

```
var forcing:Vector2D =
  new Vector2D(200*Math.cos(_omegad*time)+200*Math.sin(_omegad*time), 0);
```

Uruchom kod i przekonaj się, że drgania bardzo szybko stabilizują się na pierwotnej częstości ω_d, utrzymując jednocześnie dużą, stałą wartość amplitudy. Układ jest „zadowolony", ponieważ zewnętrzna siła wymusza na nim drgania z częstością, którą sam preferuje. Dzięki temu bardzo szybko podporządkowuje się działaniu tej siły i drga z dużą amplitudą. Choć tłumienie wyprowadza energię z układu, wymuszanie wprowadza odpowiednią jej porcję, podtrzymując tym samym drgania.

Gdy częstość wymuszania drgań jest równa częstości własnej układu, mamy do czynienia ze zjawiskiem zwanym **rezonansem**. Rezonans pojawia się w wielu układach — możemy mówić o nim w przypadku ruchu huśtawki oraz w układach elektronicznych. Jego skutki nie zawsze są pożądane; czasami, na przykład w przy-

padku wprawiania mostu w drgania rezonansowe przez regularnie pojawiające się podmuchy wiatru, może dojść do tragedii — w 1940 roku most Tacoma Narrows Bridge w Waszyngtonie zawalił się właśnie z tego powodu.

Przykład — losowa siła wymuszająca

Teraz sprawdzimy, jaki wpływ na ruch drgający ma siła wymuszająca, której wektor zmienia się losowo w czasie. Aby przeprowadzić symulację takiego zjawiska, wystarczy zmienić jedną linię kodu. Sprawdzimy najpierw tak dobrane parametry:

```
var forcing:Vector2D = new Vector2D(1000*Math.random(),0);
```

Instrukcja ta sprawia, że na ciało zadziała siła o pewnej wartości składowej wzdłuż osi X losowo wybranej z przedziału od zera do tysiąca jednostek (siła zawsze skierowana w prawo). Gdyby było to jedyne oddziaływanie w układzie, ciało ruszyłoby w prawą stronę ekranu i już nigdy nie powróciło na scenę, ale nie wolno nam zapominać o obecności siły sprężystości, która przyciągnie ciało z powrotem w kierunku punktu równowagi. Ponieważ siła sprężystości jest proporcjonalna do wychylenia ciała z położenia równowagi, im dalej od niego znajdzie się ciało, tym mocniej będzie przyciągane. Efekt jest wyjątkowo interesujący i nie wątpimy, że z chęcią przeprowadzisz kilka eksperymentów. Ponieważ siła wymuszająca jest zwrócona w prawo, ciało będzie spędzać większość czasu po prawej stronie punktu równowagi, choć czasami trafi też na lewą stronę.

Aby drgania były bardziej symetryczne, wprowadzimy do nowej linii kodu pewną modyfikację:

```
var forcing:Vector2D = new Vector2D(1000*(Math.random()-0.5),0);
```

Dzięki temu na ciało działa wektor siły o wartości wybranej losowo z przedziału [−500, 500], zwrócony w prawą bądź lewą stronę osi X. Uruchom skrypt, a przekonasz się, jak działa oscylator nieokresowy (czyli taki, w którym drgania pojawiają się nieregularnie).

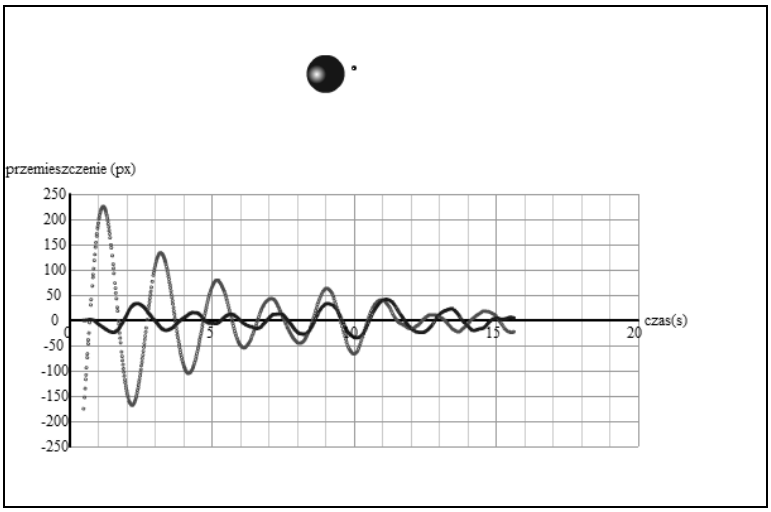
Na koniec zajmiemy się przygotowaniem oscylatora, w którym drgania będą wymuszane przez losowo działającą siłę o dwóch współrzędnych niezerowych. W tym celu znów zmodyfikujemy nieco znaną już linię kodu:

```
var forcing:Vector2D = new Vector2D(1000*(Math.random()-0.5),1000*(Math.random()-0.5));
```

Uruchom teraz skrypt — przekonasz się, że układ bardzo szybko przechodzi w stan, w którym drgania w kierunkach X i Y mają podobną wielkość, przez co ciało oscyluje wokół atraktora (rysunek 8.8). Nie może się oddalić od punktu równowagi, niezależnie od tego, jak duża zadziała nań siła, ponieważ im bardziej oddali się od atraktora, tym silniej zostanie przyciągnięte z powrotem. To ciekawy efekt. Przypomina nieco jakiś szalony ruch po orbicie czy tor, po jakim pszczoła porusza się wokół kwiatowego kielicha. Poeksperymentuj nieco z parametrami!

Grawitacja jako siła wymuszająca — skoki na bungee

Teraz zajmiemy się czymś nieco innym, mianowicie rozważymy przypadek skoku na *bungee*. Sport ten polega na oddaniu skoku z mostu lub innej wysokiej budowli na elastycznej linie, której jeden koniec jest przymocowany do budowli, a na drugim znajduje się skoczek. Lina, czy raczej potężna guma, nieobciążona niczym poza własnym ciężarem, ma pewną długość, nazwijmy ją `cordLength`. Zatem dopóki odległość skoczka



Rysunek 8.8. Drgania wymuszane losowym wektorem siły

od punktu zaczepienia liny nie przekroczy wartości `cordLength`, będzie działać na niego wyłącznie siła ciężkości oraz opór powietrza, jednak z chwilą przekroczenia granicy `cordLength` pojawi się także oddziaływanie sprężyste, które zacznie ściągać skoczka z powrotem w górę.

W tym przypadku grawitacja gra rolę siły wymuszającej, zatem $\mathbf{f}(t) = m \cdot \mathbf{g}$, gdzie m jest masą skoczka. Tym razem siła wymuszająca ma stałą wartość — takiego przypadku jeszcze nie omawialiśmy. Tłumienie zachodzi głównie za sprawą siły oporu powietrza, więc można zaimplementować je za pomocą metody `Forces.drag()`. Musimy też pamiętać, że siła sprężystości działa na skoczka wyłącznie wtedy, gdy jego odległość od punktu zamocowania liny jest większa niż `cordLength`. Odległość od punktu mocowania jest mniejsza od długości liny (przed oddaniem skoku i w początkowej fazie lotu, ale też za każdym razem, gdy siła sprężystości wyniesie skoczka ponad punkt wyznaczony długością liny).

Ponieważ ten przykład różni się znacznie od wcześniejszych, przedstawimy Ci najpierw cały kod, a potem omówimy najważniejsze jego fragmenty. Zaczniemy od przygotowania sceny, czyli skryptu z pliku *Bungee.as*.

```
package{
   import flash.display.Sprite;
   import com.physicscodes.objects.Ball;
   import com.physicscodes.math.Vector2D;
   import com.physicscodes.objects.Particle;

   public class Bungee extends Sprite{
      public function Bungee():void{
         init();
      }
      private function init():void{
         // Tworzy sprężynę.
         var spring:Sprite=new Sprite();
         addChild(spring);

         // Tworzy punkt zawieszenia.
         var fixedPoint:Ball;
```

```
            fixedPoint=new Ball(2,0x000000);
            fixedPoint.pos2D=new Vector2D(275,50);
            addChild(fixedPoint);

            // Tworzy skoczka.
            var jumper = new Jumper();
            jumper.pos2D = new Vector2D(275,50);
            jumper.mass = 90;
            addChild(jumper);

            // Wprawia układ w ruch.
            var oscillator:BungeeOscillator = new BungeeOscillator
              (jumper,fixedPoint, spring);
            oscillator.startTime();
        }
    }
}
```

W kodzie tworzymy instancje trzech obiektów: sprężyny, punktu zaczepienia i skoczka. Funkcję sprężyny pełni pusty obiekt `Sprite` — w klasie `BungeeOscillator` narysujemy w nim coś, co sprawi, że układ będzie poprawnie działać. Skoczek jest instancją klasy `Jumper`, rysunku klasy `MovieClip`, z dodaną instancją klasy `Particle`. Wszystkie trzy obiekty trafiają do konstruktora klasy `BungeeOscillator`. Oto jej kod:

```
package {
   import com.physicscodes.motion.Forcer;
   import com.physicscodes.motion.Forces;
   import com.physicscodes.objects.Particle;
   import com.physicscodes.objects.Ball;
   import com.physicscodes.math.Vector2D;
   import flash.display.Sprite;

   public class BungeeOscillator extends Forcer{
      private var _object:Particle;
      private var _center:Vector2D;
      private var _displ:Vector2D;
      private var _g:Number=20;
      private var _kDamping=0.1;
      private var _kSpring=25;
      private var _cordLength:Number=100;
      private var _spring:Sprite;

      public function BungeeOscillator(pobject:Particle,pfixedSupport:Ball,
        pspring:Sprite):void{
         _object = pobject;
         _center = pfixedSupport.pos2D;
         _spring = pspring;
         super(_object);
      }
      override protected function moveObject():void{
         super.moveObject();
         drawSpring();
      }
      private function drawSpring():void{
         with (_spring.graphics){
            clear();
            if (_displ.length > _cordLength){
```

```
                lineStyle(2,0x999999);
            }else{
                lineStyle(1,0xcccccc);
            }
            moveTo(_center.x,_center.y);
            lineTo(_object.x,_object.y)
        }
    }
    override protected function calcForce():void{
        _displ=_object.pos2D.subtract(_center);
        var gravity:Vector2D = Forces.constantGravity(_object.mass,_g);
        var damping:Vector2D = Forces.drag(_kDamping,_object.velo2D);
        var extension:Vector2D = _displ.subtract(_displ.unit().multiply(_cordLength));
        var restoring:Vector2D;
        if (_displ.length > _cordLength) {
            restoring = Forces.spring(_kSpring,extension);
        }else{
            restoring = new Vector2D(0,0);
        }
        force = Forces.add([gravity, damping, restoring]);
    }
  }
}
```

Przede wszystkim zwróć uwagę na wartość stałej sprężystości `_kSpring`, w kodzie zdefiniowaną jako 25. Zdecydowaliśmy się na tak dużą wartość, ponieważ masa skoczka w tym przykładzie to aż 90, a nie jak dotychczas — 1. Wypływa stąd prosty wniosek — jeśli chcesz, by dość masywne ciało doznało działania siły sprężystości, „sprężyna" musi być naprawdę mocna!

Spójrz teraz na ciało metody `calcForce()`: znajdziesz w nim wywołania funkcji obliczających ciężar, tłumienie i sprężystość. Dwie pierwsze zostały zaimplementowane dokładnie tak jak dotychczas, natomiast obliczanie siły sprężystości odbiega od metody, jaką posługiwaliśmy się dotąd. Przede wszystkim wektor przemieszczenia określa przesunięcie końca liny (jej wydłużenie), a nie przemieszczenie ciała względem punktu zaczepienia (ten problem wyjaśnialiśmy na początku podrozdziału). Wynika stąd, że najpierw musimy obliczyć wydłużenie liny:

```
var extension:Vector2D = _displ.subtract(_displ.unit().multiply(_cordLength));
```

by potem móc przekazać je do funkcji `Forces.spring()` jako jej drugi argument.

Po drugie, siła sprężystości ma niezerową wartość jedynie w tych przypadkach, w których ciało znajduje się poniżej granicy wyznaczonej długością nieobciążonej liny. Warunek ten przyjmuje postać `_displ.length > _cordLength`. Teraz zawartość instrukcji `if` nie powinna stanowić zagadki.

Nowością jest także metoda `drawSpring()` wywoływana w nadpisanej metodzie `moveObject()`. Jest ona odpowiedzialna za umieszczenie w animacji kreski symbolizującej linę. W każdym kroku obliczeń rysowana jest nowa kreska odpowiedniej długości. Gdy lina nie jest rozciągnięta, kreska jest cieńsza i bardziej blada. Skoczek rozpoczyna ruch dokładnie w punkcie zaczepienia liny; to sprawia, że porusza się on po prostej (drgania odbywają się w jednym wymiarze). Jeśli chcesz przenieść je w dwa wymiary, po prostu zmień współrzędną *x* położenia początkowego skoczka w pliku *Bungee.as* na przykład na:

```
object.pos2D = new Vector2D(300,50);
```

Na rysunku 8.9 przedstawiliśmy jedną klatkę animacji.

Rysunek 8.9. Symulacja skoku na bungee

Przykład — siła wymuszająca sterowana przez użytkownika

Rolę siły wymuszającej może pełnić także działanie podejmowane przez użytkownika. W tym przykładzie przygotujemy symulację, w której użytkownik będzie mógł złapać ciało kliknięciem, odciągnąć je na żądaną odległość i zwolnić. W ten sposób wprowadzi zaburzenie do układu i wywoła w nim drgania.

Aby zrealizować ten plan, musimy zmodyfikować nieco kod skryptu skoków na *bungee*. Wywołamy nowe klasy `DraggingOscillations` i `DraggingOscillator`. Klasa z pliku *DraggingOscillations.as* nie różni się zbytnio od skryptu z pliku *Bungee.as*, więc nie będziemy omawiać jej tu dokładnie. Wystarczy wspomnieć, że zmieniliśmy masę ciała na 1. Główną zmianą w klasie `DraggingOscillator` jest wprowadzenie kodu umożliwiającego przemieszczanie ciała za pomocą kursora myszy i odpowiednie zmodyfikowanie metody `calcForce()`. Oto kod klasy:

```
package {
   import com.physicscodes.motion.Forcer;
   import com.physicscodes.motion.Forces;
   import com.physicscodes.objects.Ball;
   import com.physicscodes.math.Vector2D;
   import flash.display.Sprite;
   import flash.events.MouseEvent;

   public class DraggingOscillator extends Forcer{
      private var _object:Ball;
      private var _center:Vector2D;
      private var _displ:Vector2D;
      private var _g:Number=20;
      private var _kDamping=0.5;
      private var _kSpring=1;
      private var _springLength:Number=200;
      private var _spring:Sprite;

      public function DraggingOscillator(pobject:Ball, pfixedSupport:Ball,
        pspring:Sprite):void{
         _object = pobject;
         _center = pfixedSupport.pos2D;
```

```
            _spring = pspring;
            _object.addEventListener(MouseEvent.MOUSE_DOWN,onDown);
            super(_object);
        }
        override protected function moveObject():void{
            super.moveObject();
            drawSpring();
        }
        private function drawSpring():void{
            with (_spring.graphics){
                clear();
                lineStyle(2,0x999999);
                moveTo(_center.x,_center.y);
                lineTo(_object.x,_object.y)
            }
        }
        override protected function calcForce():void{
            _displ=_object.pos2D.subtract(_center);
            var gravity:Vector2D = Forces.constantGravity(_object.mass,_g);
            var damping:Vector2D = Forces.damping(_kDamping,_object.velo2D);
            var extension:Vector2D = _displ.subtract(_displ.unit().multiply
              (_springLength));
            var restoring:Vector2D = Forces.spring(_kSpring,extension);
            force = Forces.add([gravity, damping, restoring]);
        }
        private function onDown(e:MouseEvent):void{
            _object.velo2D = new Vector2D(0,0);
            _object.stage.addEventListener(MouseEvent.MOUSE_UP,onUp);
            _object.startDrag(true);
            _spring.graphics.clear();
            stopTime();
        }
        private function onUp(e:MouseEvent):void{
            _object.pos2D = new Vector2D(_object.x,_object.y);
            _object.stage.removeEventListener(MouseEvent.MOUSE_UP,onUp);
            _object.stopDrag();
            startTime();
        }
    }
}
```

W funkcji `calcForce()` znajdziesz, jak poprzednio, definicję grawitacji, natomiast tłumienie jest wyznaczane za pomocą funkcji `Forces.damping()`. Później obliczamy wydłużenie sprężyny i przekazujemy je jako argument do funkcji siły sprężystości. Tym razem mamy do czynienia z klasyczną sprężyną, a nie rozciągliwą liną, więc siła sprężystości działa także na skróconą sprężynę. Aby uzyskać odpowiedni efekt, usunęliśmy blok instrukcji `if`.

W konstruktorze znajduje się także funkcja wychwytująca wystąpienie zdarzenia `MOUSE_DOWN`, której zadaniem jest wywołanie metody `onDown()` obsługującej to zdarzenie. Metoda `onDown()` nadaje ciału zerową prędkość, inicjuje funkcję wychwytującą zdarzenie `MOUSE_UP`, umożliwia przeciągnięcie ciała w inne miejsce ekranu, usuwa sprężynę i zatrzymuje zegar animacji. Funkcja `onUp()` obsługująca zdarzenie jest wywoływana po zwolnieniu ciała. To ona aktualizuje wektor położenia ciała, usuwa funkcje wychwytujące zdarzenia, wyłącza opcję przeciągania ciała i ponownie uruchamia zegar.

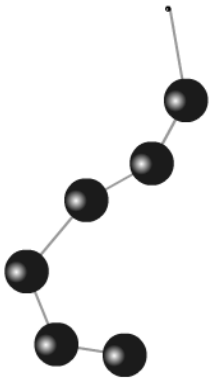
Kod symulacji jest dość prosty. Uruchom go, by zobaczyć, w jaki sposób przebiegają drgania wymuszane przez siłę grawitacji i jak maleje ich amplituda w wyniku tłumienia. Gdy przesuniesz ciało w inne miejsce, wymusisz kolejną porcję drgań.

Układy oscylatorów — wiele ciał na sprężynach

Jak dotąd zajmowaliśmy się wyłącznie układami złożonymi z jednej sprężyny i zawieszonego na niej ciała. Gdy połączy się ze sobą kilka ciał różnymi sprężynami, można obserwować bardzo interesujące zjawiska. W ten sposób można modelować naprawdę złożone układy. W następnym podrozdziale pokażemy Ci, jak przygotować taki układ. Do tematu wrócimy jeszcze w rozdziale 13., gdy przedstawimy bardziej złożone przykłady z ciałami ulegającymi deformacji.

Przykład — łańcuch mas połączonych sprężynami

Rysunek 8.10 przedstawia układ, który zaraz przygotujesz. Będzie to model złożony z kilku mas połączonych ze sobą sprężynami; całość będzie zawieszona w punkcie mocującym. Ruch punktu mocującego będzie wywoływać drgania i przemieszczenia mas zawieszonych na sprężynach.



Rysunek 8.10. Układ mas połączonych sprężynami

W symulacji uwzględnimy wpływ grawitacji, tłumienia i siły sprężystości. Dla każdej ze sprężyn zdefiniujemy jej naturalną długość, jak miało to miejsce w poprzednich przykładach. Na każdą z mas będzie działać siła grawitacji, siła tłumiąca drgania oraz siła sprężystości — jedna ze strony sprężyny umieszczonej nad masą, druga ze strony sprężyny znajdującej się pod nią (wyjątkiem jest ostatnia kulka). Same sprężyny traktujemy jako nieważkie.

Poniżej znajdziesz kod z pliku *CoupledOscillations.as*.

```
package{
   import flash.display.Sprite;
   import com.physicscodes.objects.Ball;
   import com.physicscodes.math.Vector2D;
   import com.physicscodes.objects.Particle;

   public class CoupledOscillations extends Sprite{
```

```
        public function CoupledOscillations():void{
            init();
        }
        private function init():void{
            // Tworzy sprężynę.
            var spring:Sprite=new Sprite();
            addChild(spring);

            // Tworzy punkt zaczepienia.
            var fixedPoint:Ball;
            fixedPoint=new Ball(2,0x000000);
            fixedPoint.pos2D=new Vector2D(275,50);
            addChild(fixedPoint);

            // Tworzy drgające ciała.
            var objects:Array = new Array();
            var numObjects:Number = 6;
            for (var i:uint=0; i<numObjects; i++){
                var object:Ball= new Ball(15,0x0000ff,1);
                // object.pos2D = new Vector2D(275+60*i,100+60*i);
                object.pos2D = new Vector2D(275,100+60*i);
                addChild(object);
                objects.push(object);
            }

            // Wprawia układ w ruch.
            var oscillator:CoupledOscillator = new CoupledOscillator
              (objects,fixedPoint,spring);
               oscillator.startTime();
        }
    }
}
```

Jak w poprzednich dwóch przykładach, tak i tutaj zaczynamy od utworzenia sprężyny na podstawie klasy `Sprite`, w której potem będą rysowane sprężyny. Następnie tworzymy obiekt klasy `Ball`, który będzie pełnić funkcję mocowania (mocowanie będzie ruchome). Potem definiujemy kilka dodatkowych obiektów klasy `Ball` i umieszczamy je w tablicy o nazwie `objects`. Wreszcie tworzymy instancję klasy `CoupledOscillator`, przekazujemy do niej odpowiednie obiekty i uruchamiamy metodę `startTime()`.

Zanim jednak przyjrzymy się zawartości klasy *CoupledOscillator.as*, będziemy musieli zmienić nieco klasę `MultiForcer`. Wywołamy odpowiednio przygotowaną wersję `MultiForcer2`, której kod znajdziesz poniżej.

```
package com.physicscodes.motion {
    import com.physicscodes.motion.Mover;
    import com.physicscodes.objects.Particle;
    import com.physicscodes.math.Vector2D;

    public class MultiForcer2 extends Mover{
        private var _particles:Array;
        private var _force:Vector2D;
        private var _acc:Vector2D;
        public function MultiForcer2(pparticles:Array):void{
            _particles = pparticles;
            super(null);
        }
        public function get force():Vector2D {
```

```
            return _force;
        }
        public function set force(pforce):void {
            _force = pforce;
        }
        override protected function moveObject():void{
            for (var i:uint=0; i<_particles.length; i++){
                var particle:Particle = _particles[i];
                particle.pos2D = particle.pos2D.addScaled(particle.velo2D,dt);
                calcForce(particle,i);
                _acc = _force.multiply(1/particle.mass);
                particle.velo2D = particle.velo2D.addScaled(_acc,dt);
            }
        }
        protected function calcForce(pparticle:Particle,pnum:uint):void{
            _force = Forces.zeroForce();
        }
    }
}
```

Jeśli porównasz ten kod z kodem klasy `MultiForcer`, przekonasz się, że jedyne różnice między nimi to dwa pogrubione wiersze. Zmieniliśmy metodę `calcForce()` w taki sposób, by móc wprowadzić do niej drugi argument typu `uint`. Metoda `calcForce()` zostaje wywołana w funkcji `moveObject()`, wtedy też wprowadzamy dodatkowy parametr `i` będący numerem cząstki w tablicy. Dlaczego stosujemy takie rozwiązanie? Na każdą z kulek działają siły sprężystości wywierane przez sprężynę znajdującą się ponad kulką i pod nią. Wygodniej jest rozwiązywać takie zadania, gdy wie się dokładnie, dla której kulki w danej chwili prowadzi się obliczenia. Dzięki nowemu parametrowi możemy to kontrolować.

Klasa `CoupledOscillator`, rozszerzająca przecież klasę `MultiForcer2`, natychmiast przejmuje jej możliwości, dzięki czemu możemy przeprowadzić obliczenia dla najbliższych sąsiadów kulki i wyznaczyć siłę sprężystości działającą na każdy element łańcucha. Oto kod skryptu z pliku *CoupledOscillator.as*.

```
package {
    import com.physicscodes.motion.MultiForcer2;
    import com.physicscodes.motion.Forces;
    import com.physicscodes.objects.Ball;
    import com.physicscodes.objects.Particle;
    import com.physicscodes.math.Vector2D;
    import flash.display.Sprite;

    public class CoupledOscillator extends MultiForcer2{
        private var _objects:Array;
        private var _support:Ball;
        private var _center:Vector2D;
        private var _displ:Vector2D;
        private var _g:Number=20;
        private var _kDamping=0.5;
        private var _kSpring=10;
        private var _springLength:Number=50;
        private var _spring:Sprite;

        public function CoupledOscillator(pobjects:Array,psupport:Ball,
          pspring:Sprite):void{
            _objects = pobjects;
            _support = psupport;
```

```
            _center = _support.pos2D;
            _spring = pspring;
            super(_objects);
        }
        override protected function moveObject():void{
            super.moveObject();
            drawSpring();
        }
        override protected function calcForce(pparticle:Particle,pnum:uint):void{
            var centerPrev:Vector2D;
            var centerNext:Vector2D;
            if (pnum > 0){
               centerPrev = _objects[pnum-1].pos2D;
            }else{
               centerPrev = _center;
            }
            if (pnum < _objects.length-1){
               centerNext = _objects[pnum+1].pos2D;
            }else{
               centerNext = pparticle.pos2D;
            }
            var displPrev:Vector2D = pparticle.pos2D.subtract(centerPrev);
            var displNext:Vector2D = pparticle.pos2D.subtract(centerNext);
            var extensionPrev:Vector2D = displPrev.subtract(displPrev.unit().
              multiply(_springLength));
            var extensionNext:Vector2D = displNext.subtract(displNext.unit().
              multiply(_springLength));
            var gravity:Vector2D = Forces.constantGravity(pparticle.mass,_g);
            var damping:Vector2D = Forces.damping(_kDamping,pparticle.velo2D);
            var restoringPrev:Vector2D = Forces.spring(_kSpring,extensionPrev);
            var restoringNext:Vector2D = Forces.spring(_kSpring,extensionNext);
            force = Forces.add([gravity, damping, restoringPrev, restoringNext]);
        }
        private function drawSpring():void{
            with (_spring.graphics){
               clear();
               lineStyle(2,0x999999);
               moveTo(_center.x,_center.y);
               for (var i:uint=0; i<_objects.length; i++){
                  var X:Number = _objects[i].xpos;
                  var Y:Number = _objects[i].ypos;
                  lineTo(X,Y);
               }
            }
        }
    }
}
```

Kod w większości powinien być Ci znany. Pojawia się w nim kilka nowych instrukcji związanych z tym, że w układzie znajduje się teraz kilka poruszających się ciał i sprężyn, które oddziałują ze swoimi najbliższymi sąsiadami. Problem ten rozwiązaliśmy, dodając w funkcji `calcForce()` zmienne `centerPrev` i `centerNext`, które wyznaczają położenie kulek sąsiadujących z kulką, dla której prowadzimy właśnie obliczenia. Wartości zmiennych `centerPrev` i `centerNext` są dostępne poprzez podanie indeksu `pnum` tablicy `objects`, który jest jednym z argumentów funkcji `calcForce()`. Parametr `centerPrev` dla pierwszej kulki wskazuje na położenie

punktu zawieszenia, `centerNext` ostatniej kulki to współrzędne położenia jej własnego środka. Wartości zmiennych `centerPrev` i `centerNext` pozwalają obliczać wektory przemieszczenia kulki względem jej najbliższych sąsiadów w sposób, z jakiego korzystaliśmy w poprzednich rozwiązaniach. Przemieszczenie końca sprężyny względem punktu wyznaczającego jej naturalną długość `_springLength` musi być wyznaczane osobno dla każdej sprężyny, ale metoda nie zmieniła się i nadal przebiega to tak, jak w poprzednich dwóch przykładach. Siła sprężystości działająca na ciało ze strony wybranej sprężyny jest wyznaczana za pomocą funkcji `Forces.spring()` i sumowana z wektorami sił grawitacji i tłumienia.

Linia rozpoczynająca się w punkcie zaczepienia i łącząca wszystkie kulki jest rysowana za pomocą metody `drawSpring()` wywoływanej wewnątrz funkcji `moveObject()`.

Początkowe położenia ciał zostają zadane w skrypcie pliku *CoupledOscillations.as*. Spróbuj uruchomić skrypt kilkakrotnie, za każdym razem z innym położeniem początkowym kulek, i przekonaj się, w jaki sposób zmienia się ich ruch pod wpływem sił grawitacji i sprężystości oraz jak starają się one osiągnąć położenie równowagi.

Nie wahaj się eksperymentować. Podany poniżej kod po wstawieniu do funkcji `moveObject()` sprawi, że punkt zawieszenia zacznie poruszać się ruchem sinusoidalnym, ciągnąc za sobą układ kulek.

```
_support.xpos = 100*Math.sin(1.0*time)+275;
_center = _support.pos2D;
```

W ten sposób uzyskasz efekt zbliżony do przedstawionego na rysunku 8.10.

Spróbuj też zmieniać masy ciał i stałe sprężystości sprężyn. Uruchom możliwość przeciągania kulek w inne miejsce sceny za pomocą myszy albo zamknij łańcuch. Wykorzystaj wszystkie pomysły, jakie przyjdą Ci do głowy.

Podsumowanie

Mamy nadzieję, że przykłady zawarte w tym rozdziale dały Ci dużo radości. Jak wspomnieliśmy na początku, sprężyny to nie tylko źródło dobrej zabawy, ale także bardzo potężne narzędzie modelowania zjawisk fizycznych. Można powiązać je z prędkością, dzięki czemu ciało będzie łagodnie przyśpieszać i zwalniać, by w końcu zyskać pewną prędkość „równowagi". Nie wątpimy, że znajdziesz dla sprężyn niejedno zastosowanie. Jedynym ograniczeniem będzie Twoja wyobraźnia. My tymczasem chwilowo zostawiamy sprężyny, ale wrócimy do nich jeszcze.

Skorowidz

A

G

H

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

V

W

Z

Odwzoruj prawdziwy świat w najdrobniejszych szczegółach!

Dobre odwzorowanie praw fizyki w grach zazwyczaj sprawia, że stają się one bardziej atrakcyjne. Dlaczego tak uważamy? Pewnie ma to związek z naszą intuicją — lubimy, gdy przedmioty w grze zachowują się zgodnie z oczekiwaniami. Realistyczne zderzenia, grawitacja, odbicia to tylko część elementów, których zastosowanie zwiększy Twoją szansę na sukces!

Ta książka porusza wszystkie aspekty związane z wykorzystaniem praw fizyki w grach, animacjach i symulacjach tworzonych we Flashu. W trakcie lektury zostaniesz stopniowo i bezboleśnie wprowadzony w świat obliczeń numerycznych — od najprostszych, pozwalających nadać ruch odbijającej się piłce, do najbardziej skomplikowanych, odwzorowujących na przykład prawdziwy ruch planet w Układzie Słonecznym. Ponadto dowiesz się, jaki wpływ na ruch ma tarcie oraz jak zaprezentować siłę wyporu. Książka ta jest idealną pozycją dla każdego programisty chcącego tworzyć gry i animacje jak najbliższe rzeczywistemu światu.

Zdobądź tę książkę i opanuj wiedzę na temat:

- wykorzystania praw fizyki w projektowaniu gier i animacji
- symulowania uderzeń, odbić i zderzeń
- wpływu siły grawitacji i tarcia na ruch obiektów

Helion

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

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

ISBN 978-83-246-4473-5

9 788324 644735

Cena: 79,00 zł