helion.pl

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

Flash MX. Programowanie



Autor: Robert Penner
Tłumaczenie: Marek Binkowski, Rafał Jońca
ISBN: 83-7361-085-5
Tytuł oryginału: Robert Penner's Programming Macromedia Flash MX
Format: B5, stron: 400
Przykłady na ftp: 545 kB

Robert Penner – ekspert i innowator w dziedzinie języka ActionScript – dzieli się swoją wiedzą na temat programowania Macromedia Flash. Penner przybliża tajniki łączenia złożonej matematyki i fizyki z atrakcyjną i estetyczną grafiką Flasha. Dzięki książce „Flash MX. Programowanie" poznasz podstawowe zagadnienia związane z przestrzenią, matematyką, kolorem, ruchem i kształtem, a następnie dowiesz się, jak wykorzystać tę teorię do generowania nieprzeciętnych animacji. Przekonasz się, jak wspaniale możesz za pomocą ActionScriptu symulować zjawiska atmosferyczne takie jak tornado, śnieżyca czy zorza polarna.

Książka omawia szeroki zakres zagadnień – od projektowania i mechaniki ruchu po tajniki pisania poprawnego, zorientowanego obiektowo kodu; służy także pomocą w tworzeniu kolorów, dźwięku, ruchu i interakcji.

- Wyjaśnienie podstaw matematyki w kontekście programu Flash
- Tworzenie skomplikowanych animacji oraz prostego ruchu i efektów typu „rollover"
- Prezentacja możliwości obiektowych języka ActionScript
- Tworzenie łatwych do modyfikacji interfejsów wielokrotnego użytku
- Tworzenie obiektów trójwymiarowych
- Dokładne wyjaśnienie procesu tworzenia kilku przykładowych animacji: śnieżycy, fraktalnego tancerza i zorzy polarnej
- Projektowanie i generowanie dynamicznej animacji za pomocą języka ActionScript
- Tworzenie grafiki – od prostych schematów po złożone filmy.

Połączenie odwiecznych praw fizyki i matematyki z nowoczesną technologią Flasha MX sprawi, że będziesz w stanie tworzyć niezwykłe animacje. Poznaj granice własnej wyobraźni z książką „Flash MX. Programowanie"!

Robert Penner to niezależny twórca filmów w programie Macromedia Flash, konsultant, pisarz i wykładowca. Na całym świecie znany jest z innowacyjnych pomysłów i animacji odtwarzających piękno natury. Jego biografię artystyczną i efekty eksperymentów można znaleźć na witrynie http://robertpenner.com

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

Spis treści

O Autorze ... 17

Wprowadzenie ... 19

Co omawia ta książka? (opisy części i rozdziałów) ... 19
Jak czytać tę książkę? ... 21
Pliki z przykładami ... 21

Część I Początek procesu ... 23

Rozdział 1. Jak poznałem Flasha? ... 25

Zaplecze osobiste ... 25
Lata studenckie ... 27
Studium techniczne ... 28
Odkrycie Flasha ... 28
Istota dyscypliny ... 30
Co to są dyscypliny? ... 30
Nawyki ... 31
Moje dyscypliny ... 32
Samodzielna nauka ... 32
Rozwój przez ćwiczenie ... 33
Przykład: nauka skrótów klawiszowych ... 34
Społeczność ... 35
Uczenie się przez nauczanie innych ... 36
Udostępnianie kodu źródłowego ... 37
Proces iteracyjny ... 38
Nauka jest okrągła ... 38
Ustalanie priorytetów ... 39
Tworzenie z myślą o przyszłości ... 40
Układ książki ... 41

Rozdział 2. Programowanie zorientowane obiektowo ... 43

Istota programowania ... 43
Pamięć i zmienne ... 44
Umiejętności i funkcje ... 44
Obiekty: pamięć i umiejętności ... 45
Właściwości i metody ... 45

Klasy ... 47
Wbudowane klasy i obiekty Flasha ... 47
Konstruktory klas ... 49
Klonowanie klas wizualnych ... 51
Właściwość prototype ... 51
Dodawanie metod do klas ... 53
Zastępowanie wbudowanych metod ... 54
Rozszerzanie możliwości obiektów statycznych ... 54
Klasyczne pojęcia z dziedziny OOP ... 55
Abstrakcja ... 55
Separacja ... 56
Polimorfizm ... 59
Tworzenie własnej klasy ... 59
Analiza ... 60
Lista zadań ... 60
Scenariusze poszczególnych zadań ... 61
Projektowanie ... 62
Dobór właściwości ... 63
Dobór metod ... 63
Relacje między obiektami ... 64
Diagramy klas ... 65
Tworzenie klas w języku ActionScript ... 66
Konstruktor klasy PhotoAlbum ... 66
Metoda showPhotoAt() ... 68
Metoda next() ... 69
Metoda prev() ... 70
Klasa PhotoAlbum w akcji ... 71
Dziedziczenie klas ... 72
Dziedziczenie właściwości a dziedziczenie metod ... 72
Dziedziczenie właściwości za pomocą funkcji super() ... 73
Dziedziczenie metod ... 74
Obiekt super ... 74
Dziedziczenie metod za pomocą słowa kluczowego new ... 75
Dziedziczenie metod przy użyciu właściwości __proto__ ... 76
Rozwiązanie korzystające z właściwości __constructor__ ... 76
Metoda superCon() ... 78
Wnioski ... 79

Część II Podstawowe pojęcia ... 81

Rozdział 3. Matematyka część I: trygonometria i układy współrzędnych ... 83
Trygonometria ... 84
Trójkąt prostokątny ... 84
Twierdzenie Pitagorasa ... 84
Odległość między dwoma punktami ... 85
Kąty w trójkątach prostokątnych ... 86
Sinus ... 88
Kosinus ... 91
Tangens ... 93
Arcus tangens ... 94

Arcus kosinus ... 96
Arcus sinus ... 97
Układy współrzędnych ... 98
Kartezjański układ współrzędnych ... 98
Układ współrzędnych Flasha ... 99
Współrzędne biegunowe ... 102
Wnioski ... 107

Rozdział 4. Matematyka część II: wektory w dwóch wymiarach ... 109

Wektory ... 109
Klasa Vector ... 110
Konstruktor klasy Vector ... 111
Metoda Vector.toString() ... 111
Metoda Vector.reset() ... 112
Metoda Vector.getClone() ... 112
Metoda Vector.equals() ... 113
Dodawanie wektorów ... 114
Metoda Vector.plus() ... 115
Metoda Vector.plusNew() ... 115
Odejmowanie wektorów ... 116
Metoda Vector.minus() ... 116
Metoda Vector.minusNew() ... 116
Odwracanie wektora ... 117
Metoda Vector.negate() ... 117
Metoda Vector.negateNew() ... 118
Skalowanie wektorów ... 118
Metoda Vector.scale() ... 118
Metoda Vector.scaleNew() ... 119
Długość wektora ... 119
Metoda Vector.getLength() ... 120
Metoda Vector.setLength() ... 120
Orientacja wektora ... 121
Metoda Vector.getAngle() ... 121
Metoda Vector.setAngle() ... 122
Obracanie wektora ... 122
Metoda Vector.rotate() ... 123
Metoda Vector.rotateNew() ... 123
Iloczyn skalarny ... 124
Interpretacja iloczynu skalarnego ... 124
Metoda Vector.dot() ... 124
Wektory prostopadłe ... 125
Poszukiwanie normalnej wektora ... 125
Metoda Vector.getNormal() ... 125
Sprawdzanie prostopadłości wektorów ... 126
Metoda Vector.isNormalTo() ... 126
Obliczanie kąta między dwoma wektorami ... 127
Wyprowadzenie wzoru na kąt między wektorami ... 127
Metoda Vector.angleBetween() ... 128
Traktowanie punktów jako wektory ... 128
Wnioski ... 130

Rozdział 5. Matematyka część III: wektory w trzech wymiarach 131
Osie X, Y i Z 131
Klasa Vector3d 132
Konstruktor klasy Vector3d 132
Metoda Vector3d.toString() 132
Metoda Vector3d.reset() 133
Metoda Vector3d.getClone() 133
Metoda Vector3d.equals() 134
Podstawowe operacje z udziałem trójwymiarowych wektorów 134
Metoda Vector3d.plus() 134
Metoda Vector3d.plusNew() 135
Metoda Vector3d.minus() 135
Metoda Vector3d.minusNew() 136
Metoda Vector3d.negate() 136
Metoda Vector3d.negateNew() 137
Metoda Vector3d.scale() 137
Metoda Vector3d.scaleNew() 137
Metoda Vector3d.getLength() 138
Metoda Vector3d.setLength() 138
Iloczyny wektorów 139
Iloczyn skalarny 139
Metoda Vector3d.dot() 140
Iloczyn wektorowy 140
Metoda Vector3d.cross() 142
Kąt między dwoma wektorami 142
Równanie pozwalające na wyznaczenie kąta 143
Metoda Vector3d.angleBetween() 143
Rzutowanie wektora na płaszczyznę ekranu 144
Metoda Vector3d.getPerspective() 144
Metoda Vector3d.persProject() 145
Metoda Vector3d.persProjectNew() 146
Obroty w trzech wymiarach 146
Obrót wokół osi X 146
Metoda Vector3d.rotateX() 146
Metoda Vector3d.rotateXTrig() 147
Obrót wokół osi Y 148
Metoda Vector3d.rotateY() 148
Metoda Vector3d.rotateYTrig() 149
Obrót wokół osi Z 149
Metoda Vector3d.rotateZ() 150
Metoda Vector3d.rotateZTrig() 150
Metoda Vector3d.rotateXY() 151
Metoda Vector3d.rotateXYTrig() 151
Metoda Vector3d.rotateXYZ() 152
Metoda Vector3d.rotateXYZTrig() 152
Rysowanie cząsteczek w trzech wymiarach 153
Klasa Particle3d 153
Metoda Particle3d.attachGraphic() 155
Metoda Particle3d.render() 155

Przykład: ściana cząsteczek ... 156
Przygotowania ... 157
Funkcja getWallPoints() ... 157
Inicjalizacja ściany ... 158
Funkcja arrayRotateXY() ... 159
Animowanie z udziałem zdarzenia onEnterFrame ... 159
Wnioski ... 160

Rozdział 6. Programowanie zdarzeń ... 161

Model zdarzeń we Flashu 5 ... 161
Model obsługi zdarzeń we Flashu MX ... 162
Detektory zdarzeń przycisków i klipów filmowych we Flashu MX ... 163
Przykład: Poślizg MX ... 164
Sondy ... 167
Programowanie bazujące na czasie ... 167
Programowanie bazujące na zdarzeniach ... 168
Nasłuchiwanie zdarzeń obiektów Flasha MX ... 169
Przykład: nasłuchiwanie zdarzeń pola tekstowego ... 170
Wysyłanie zdarzeń do wielu obiektów ... 171
Wbudowane źródła zdarzeń ... 172
Zajrzyjmy głębiej ... 172
Możliwości źródeł zdarzeń ... 173
Obiekt ASBroadcaster ... 174
Inicjalizacja źródła zdarzeń ... 174
Rozsyłanie zdarzeń ... 175
Źródło zdarzeń NewsFeed ... 176
Konstruktor obiektu NewsFeed ... 177
Metoda NewsFeed.toString() ... 177
Inicjalizacja źródła zdarzeń za pomocą metody ASBroadcaster ... 177
Modyfikacja konstruktora ... 178
Metoda NewsFeed.sendNews() ... 178
Przygotowywanie systemu ... 179
Tworzenie obiektu będącego źródłem zdarzeń ... 180
Tworzenie obiektu, dla którego zostanie założona sonda ... 180
Definiowanie detektorów zdarzeń ... 180
Zakładanie sond ... 181
Rozsyłanie zdarzeń ... 181
Wnioski ... 182

Część III Dynamiczna grafika ... 183

Rozdział 7. Ruch, klatki pośrednie, przyspieszanie i zwalnianie ... 185

Zasady ruchu ... 185
Dodatkowe informacje o położeniu ... 186
Położenie jako funkcja czasu ... 187
Ruch jako wykres ... 187
Statyczne klatki pośrednie we Flashu ... 188

Dynamiczne klatki pośrednie w ActionScript ... 189
Przejście wykładnicze ... 189
Podstawowe komponenty przejścia ... 191
Funkcje przejść ... 191
Przejście liniowe ... 192
Wykres ... 192
Funkcja w ActionScript ... 194
Implementacja przejścia za pomocą funkcji ... 194
Estetyka ruchu liniowego ... 196
Wygładzanie ruchu ... 197
Estetyka ruchu przyspieszonego ... 198
Przyspieszenie ... 198
Spowolnienie ... 199
Przyspieszenie i spowolnienie w jednym ... 199
Odmiany przejść z przyspieszeniem ... 200
Przyspieszenie kwadratowe ... 200
Przyspieszenie stopnia trzeciego ... 201
Przyspieszenie stopnia czwartego ... 202
Przyspieszenie stopnia piątego ... 203
Przyspieszenie sinusoidalne ... 204
Przyspieszenie wykładnicze ... 205
Przyspieszenie kołowe ... 206
Wprowadzenie do klasy Tween ... 207
Klasa Motion ... 207
Konstruktor Motion ... 207
Metody publiczne ... 209
Metody ustawiania i pobierania ... 213
Metody prywatne ... 217
Właściwości pobierania i ustawiania ... 218
Zakańczanie ... 219
Klasa Tween ... 219
Konstruktor Tween ... 220
Metody publiczne ... 221
Metody pobierania i ustawiania ... 223
Właściwości pobierania i ustawiania ... 224
Zakańczanie ... 225
Wnioski ... 225

Rozdział 8. Fizyka ... 227

Kinematyka ... 227
Położenie ... 227
Przemieszczenie ... 228
Odległość ... 229
Prędkość ... 229
Szybkość ... 231
Przyspieszenie ... 231
Siła ... 233
Pierwsze prawo dynamiki Newtona ... 233
Siła niezrównoważona (siła wypadkowa) ... 234

Drugie prawo dynamiki Newtona ... 235
Ruch we Flashu sterowany siłą ... 236
Tarcie ... 237
Tarcie kinetyczne ... 238
Tarcie statyczne ... 239
Tarcie dla cieczy i gazów ... 240
Grawitacja w kosmosie ... 241
Grawitacja w pobliżu powierzchni ... 244
Elastyczność (sprężystość) ... 245
Stan spoczynkowy ... 246
Prawo Hooke'a ... 246
Stała sprężystości ... 246
Kierunek siły sprężystości ... 247
Implementacja w ActionScript ... 247
Ruch falowy ... 248
Amplituda ... 249
Częstotliwość ... 249
Okres ... 250
Przesunięcie czasu ... 251
Offset ... 251
Równanie fali ... 252
Klasa WaveMotion ... 252
Konstruktor WaveMotion ... 252
WaveMotion.getPosition() ... 253
Metody ustawiania i pobierania ... 253
Właściwości ustawiania i pobierania ... 255
Używanie WaveMotion ... 255
Wnioski ... 256

Rozdział 9. Kolorowanie w ActionScript ... 257

Ustawianie koloru ... 257
Color.setRGB() ... 257
MovieClip.setRGB() ... 258
Color.getRGB() ... 259
MovieClip.getRGB() ... 259
Color.setRGBStr() ... 259
MovieClip.setRGBStr() ... 260
Color.getRGBStr() ... 260
MovieClip.getRGBStr() ... 261
Color.setRGB2() ... 262
MovieClip.setRGB2() ... 262
Color.getRGB2() ... 263
MovieClip.getRGB2() ... 264
Przekształcenia kolorów ... 264
Color.setTransform() ... 264
MovieClip.setColorTransform() ... 265
Color.getTransform() ... 265
MovieClip.getColorTransform() ... 265

Powrót do oryginalnego koloru ... 265
Color.reset() ... 265
MovieClip.resetColor() ... 266
Sterowanie jasnością ... 266
Color.setBrightness() ... 267
MovieClip.setBrightness() ... 267
Color.getBrightness() ... 268
MovieClip.getBrightness() ... 268
Przesunięcie jasności ... 268
Color.setBrightOffset() ... 269
MovieClip.setBrightOffset() ... 269
Color.getBrightOffset() ... 270
MovieClip.getBrightOffset() ... 270
Zabarwienie ... 270
Color.setTint() ... 270
MovieClip.setTint() ... 271
Color.getTint() ... 271
MovieClip.getTint() ... 271
Offset zabarwienia ... 272
Color.setTintOffset() ... 272
MovieClip.setTintOffset() ... 272
Color.getTintOffset() ... 272
MovieClip.getTintOffset() ... 273
Odwrócenie kolorów ... 273
Color.invert() ... 273
MovieClip.invertColor() ... 274
Color.setNegative() ... 274
MovieClip.setNegativeColor() ... 274
Color.getNegative() ... 274
MovieClip.getNegativeColor() ... 275
Zmiany poszczególnych kolorów ... 275
setRed() ... 275
setGreen() ... 275
setBlue() ... 276
getRed() ... 276
getGreen() ... 276
getBlue() ... 276
Dodawanie właściwości koloru do klipu filmowego ... 277
MovieClip._red, _green i _blue ... 277
MovieClip._rgb ... 279
MovieClip._brightness ... 279
MovieClip._brightOffset ... 280
Wnioski ... 280

Rozdział 10. Rysowanie w ActionScript ... 281

Interfejs programistyczny rysowania kształtów ... 281
MovieClip.moveTo() ... 282
MovieClip.lineTo() ... 282

MovieClip.lineStyle() ... 282
MovieClip.curveTo() ... 283
MovieClip.beginFill() ... 283
MovieClip.beginGradientFill() ... 284
MovieClip.endFill() ... 284
MovieClip.clear() ... 285
Animacja i dynamiczne rysowanie ... 285
Animacja klipu filmowego ... 285
Animacja kształtu ... 286
Rysowanie typowych kształtów ... 287
Linie ... 287
Trójkąty ... 288
Czworobok ... 290
Prostokąty ... 290
Kwadraty ... 293
Kropki ... 294
Wieloboki ... 295
Wieloboki foremne ... 296
Elipsy ... 297
Okręgi ... 298
Znajdowanie położenia kursora rysowania ... 298
Właściwości MovieClip._xpen i _ypen ... 298
Właściwości MovieClip._xpenStart i _ypenStart ... 299
Inicjalizacja właściwości ... 300
Krzywe Beziera stopnia trzeciego ... 300
Krzywe Beziera stopnia drugiego a trzeciego ... 301
Metody rysowania krzywych Beziera stopnia trzeciego ... 303
Wnioski ... 305

Część IV Przykłady ... 307

Rozdział 11. Nocna aura ... 309
Rozwój pomysłu ... 309
Klasa PhysicsParticle ... 310
Konstruktor ... 310
Metody publiczne ... 312
Metody pobierania i ustawiania ... 313
Metody prywatne ... 315
Właściwości pobierania i ustawiania ... 320
Wykańczanie ... 320
Klasa Force ... 320
Konstruktor ... 321
Metody pobierania i ustawiania ... 321
Pozostałe metody ... 323
Właściwości pobierania i ustawiania ... 324
Wykańczanie ... 325
Klasa ElasticForce ... 325
Konstruktor ... 325
Metody ... 326

Właściwości pobierania i ustawiania ... 328
Zakańczanie ... 328
Prosty przykład ... 328
Kod FLA przykładu Aurora Borealis ... 329
Kod głównej listwy czasowej ... 329
Komponent aurora ... 330
Klip filmowy cząsteczki ... 330
Wnioski ... 334

Rozdział 12. Śnieżyca ... 335

Klasa Snowflake ... 335
Funkcje pomocnicze ... 337
Konstruktor Snowflake ... 338
Metody ustawiania i pobierania ... 339
Metody prywatne ... 341
Klasa Snowstorm ... 346
Konstruktor Snowstorm ... 346
Metody publiczne ... 347
Kod przykładu Snowstorm ... 349
Kod głównej listwy czasowej ... 349
Komponent snowstorm ... 349
Metody komponentu ... 350
Wnioski ... 352

Rozdział 13. Fraktalny tancerz ... 353

Komponent FractalTree ... 354
Parametry komponentu ... 354
Metody ... 355
Klasa FractalBranch ... 357
Konstruktor FractalBranch ... 357
Metody ... 358
Klasa MotionCam ... 362
Konstruktor MotionCam ... 362
Metody publiczne ... 363
Metody pobierania i ustawiania ... 365
Metody prywatne ... 367
Wnioski ... 368

Rozdział 14. Cyklon ... 369

Wymyślanie cyklonu ... 370
Rozbijamy zagadnienie ... 373
Cząsteczka ... 373
Ścieżka ... 373
Path.onEnterFrame() ... 373
Path.init() ... 374
Oval ... 375
Oval.init() ... 375
Oval.sidewind() ... 376

Komponent Cyclone ... 376
Cyclone.init() ... 377
Cyclone.makeParticle() ... 377
Cyclone.grow() ... 378
Cyclone.sidewind() ... 379
Cyclone.startSidewind() i stopSidewind() ... 380
Parametry komponentu ... 380
Dragger ... 381
Akcje w klatkach ... 381
Dragger.appear() ... 381
Dragger.onEnterFrame() ... 381
Akcje przycisku ... 382
Wnioski ... 382

Dodatki ... 383

Skorowidz ... 385

Rozdział 8.
Fizyka

Zrozumienie fizyki to zrozumienie działania otaczającego nas świata. W tym rozdziale omówimy podstawy mechaniki — prędkość, przyspieszenie i siłę — oraz związki między nimi. Poznamy pewne szczególne rodzaje sił, jak tarcie, sprężystość, grawitacja i sposoby ich obliczania oraz dołączania do dynamicznych animacji. Na końcu zajmiemy się fizyką fali i własną klasą `WaveMotion`, która umożliwi generowanie oscylacji.

Kinematyka

Mechanika to dział fizyki zajmujący się działaniem sił na obiekty fizyczne i wynikającym z tego ruchem. Istnieją trzy główne obszary, którymi zajmuje się mechanika: statyka, kinetyka i kinematyka. Statyka zajmuje się siłami w równowadze, a kinetyka — przyspieszeniem powodowanym przez brak równoważenia się sił. Tymczasem kinematyka to czysty ruch, który nie przejmuje się siłami i masą. W tym podrozdziale przyjrzymy się kinematyce, a w dalszej części rozdziału przejdziemy do kinetyki (sił).

Położenie

W tym rozdziale skupimy się na kinematyce dwuwymiarowej, czyli z dwoma parametrami: x i y. Z tego powodu *położenie* w naszych rozważaniach będzie dwuwymiarowym wektorem zawierającym komponenty x i y. Skorzystamy z klasy `Vector` omówionej w rozdziale 4. Obiekt zawierający dwuwymiarowe położenie wykreujemy, tworząc kopię `Vector` za pomocą poniższego kodu.

```
// położenie jako dwuwymiarowy wektor
pos = new Vector (2, 5);
```

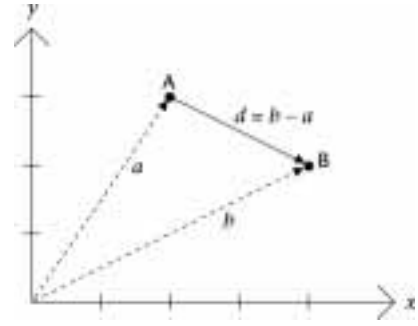
W dalszej części rozdziału będziemy bardzo intensywnie wykorzystywać tę podstawową składnię do tworzenia wektorów dotyczących różnych aspektów fizyki, na przykład przemieszczenia, prędkości, przyspieszenia i siły.



Kinematykę można rozszerzyć do trzech lub więcej wymiarów bez zbytnich problemów. Jeśli nie będziemy mieli kłopotów z implementacją kinematyki w ActionScript za pomocą klasy `Vector`, nie powinniśmy też mieć większych problemów z przejściem na klasę `Vector3d` (patrz rozdział 5.). Specjalnie korzystałem z polimorfizmu (patrz rozdział 2.) przy projektowaniu obydwu klas wektorów, aby ich interfejsy były jak najbardziej podobne.

Przemieszczenie

Przemieszczenie to *zmiana położenia*. Innymi słowy, przemieszczenie to różnica między dwoma położeniami. Gdy położenia to wektory, przemieszczenie także jest wektorem. Wektor przemieszczenia między dwoma położeniami znajdziemy, odejmując jeden wektor położenia od drugiego. Rysunek 8.1 przedstawia wektory położeń i wektor przemieszczenia między nimi.



Rysunek 8.1. *Znajdowanie wektora przemieszczenia d między punktami A i B*

Mamy dwa punkty: *A* w (2, 3) i *B* w (4, 2). Mamy też dwa odpowiadające im wektory: ***a*** = [2, 3] i ***b*** = [4, 2]. Wektor przemieszczenia ***d*** obliczamy jako różnicę ***b*** i ***a*** (odejmujemy ***b*** od ***a***). Rysunek 8.1 przedstawia sytuację graficznie, ale poniżej przedstawiam sposób analityczny.

$$a = [2,3]$$
$$b = [4,2]$$

Niech ***d*** będzie przemieszczeniem między ***b*** i ***a***.

$$d = b - a$$
$$d = [4-2, 2-3]$$
$$d = [2,1]$$

Klasa `Vector` powoduje, że przeniesienie tego procesu do ActionScript jest bardzo proste, ponieważ określiliśmy już metodę odejmowania. Poniższy kod przedstawia sposób znajdowania wektora przemieszczenia dla dwóch określonych wektorów położenia.

```
// znajdowanie wektora przemieszczenia
punktA = new Vector (2, 3);
punktB = new Vector (4, 2);
przem = punktB.minusNew (punktA);
trace (przem);   // wyświetli: [2, -1]
```

Najpierw tworzymy dwa wektory położenia, `punktA` i `punktB`. Aby znaleźć przemieszczenie z `punktA` do `punktB`, odejmujemy pierwszy wektor od drugiego. Korzystając z możliwości klasy `Vector`, wywołujemy metodę `minusNew()` dla punktu `B`, by odjąć punkt `A` i zwrócić różnicę jako nowy wektor `przem`.

Odległość

Choć *odległość* i *przemieszczenie* są blisko związane, to jednak oznaczają dwie różne rzeczy. Odległość to wartość jednowymiarowa, a przemieszczenie może mieć dwa lub więcej wymiarów. Inaczej mówiąc, odległość to skalar, a przemieszczenie to wektor. W powyższym kodzie przemieszczenie między `punktA` i `punktB` wynosi [2, –1]. Gdy policzymy odległość, okaże się, że jest to pierwiastek kwadratowy z 5, czyli w przybliżeniu 2,236. Możemy to sprawdzić za pomocą twierdzenia Pitagorasa, na przykład przy użyciu funkcji `Math.distance` z rozdziału 3. Możemy też za pomocą poniższego kodu obliczyć *długość* wektora przemieszczenia. Uzyskamy tę samą wartość.

```
// kontynuacja poprzedniego przykładu
trace (disp.getLength());  // wyświetli: 2.23606797749979
```

To nie przypadek. Odległość między dwoma punktami jest zawsze równa *długości wektora przemieszczenia* między tymi punktami. Jeśli więc mamy wektor przemieszczenia między dwoma punktami i chcemy poznać odległość, wywołujemy dla wektora metodę `getLength()`.

```
// znajdowanie odległości z przemieszczenia
przem = new Vector (3, 4);
odleglosc = przem.getLength();
trace (odleglosc);         // wyświetli: 2.23606797749979
```

Prędkość

Prędkość to *zmiana położenia w czasie*. Można powiedzieć, że jest to *stosunek przemieszczenia i czasu*. Prędkość to wartość wektorowa, w naszym przypadku będzie dwuwymiarowa.

Gdy tworzymy animację za pomocą skryptu, rzadko chcemy *obliczać* prędkość poruszania się obiektu, śledząc jego położenia. Chcemy raczej *zdefiniować* na początku prędkość i poruszać obiekt zgodnie z nią. Jeśli na przykład gramy w grę *Asteroids*, nie sterujemy bezpośrednio położeniem statku. Mamy tylko wpływ na prędkość, która modyfikuje położenie. A oto inny przykład: pomyślmy o różnicy między myszą a joystickiem. Mysz bezpośrednio steruje *położeniem* kursora, ale joystick steruje raczej jego *prędkością*. W obiektach z rzeczywistego świata znacznie łatwiej sterować prędkością niż położeniem, więc większość urządzeń zdalnego sterowania korzysta z joysticków.

Ruch ze stałą prędkością

Gdy obiekt posiada stałą prędkość w pionie i w poziomie, porusza się po linii prostej. Poniższy pod przedstawia przykład ruchu klipu filmowego ze stałą prędkością.

```
// ruch ze stałą prędkością
pos = new Vector (0, 0);
vel = new Vector (1, 2);
this.onEnterFrame = function () {
  pos.plus (vel);
```

```
  this._x = pos.x;
  this._y = pos.y;
};
```

Najpierw deklarujemy dwa obiekty `Vector`, `pos` i `vel`, które reprezentują położenie i prędkość. Początkowe położenie to (0, 0), a początkowa prędkość to w dół i na prawo. Następnie kreujemy pętlę dla klatek przy użyciu procedury obsługi zdarzenia `onEnterFrame()`. W każdej klatce dodajemy do położenia wektor prędkości, korzystając z metody `Vector.plus()`. Na końcu zmodyfikowane komponenty `x` i `y` przepisujemy do właściwości `_x` i `_y` klipu filmowego.



Powyższy przykład używa układu współrzędnych Flasha. Jeśli chcemy korzystać z układu kartezjańskiego, wystarczy zastąpić `this._y = pos.y` konstrukcją `this._y = -pos.y`. Odwracamy wartość komponentu `y` przed przypisaniem go do `_y`.

Ruch z losową prędkością

Po omówieniu stałej prędkości przejdźmy do przykładu ruchu, w którym prędkość nie jest stała.

```
// ruch z losową prędkością
pos = new Vector (200, 200);
vel = new Vector (0, 0);
this.onEnterFrame = function () {
  vel.reset (random(9)-4, random(9)-4);
  pos.plus (vel);
  this._x = pos.x;
  this._y = pos.y;
};
```

Struktura jest mniej więcej taka sama jak w poprzednim przykładzie. Najpierw definiujemy wektory `pos` i `vel` (obiekty `Vector`). Następnie w procedurze obsługi zdarzenia `onEnterFrame()` dodajemy `vel` do `pos` i renderujemy aktualne położenie na ekranie. Modyfikacja polega tylko na jednym dodatkowym wierszu kodu.

```
vel.reset (random(9)-4, random(9)-4);
```

W każdej klatce ustawiamy nową prędkość, którą później wykorzystujemy do przesunięcia obiektu. Wartości `x` i `y` z `vel` to wartości losowe z przedziału od –4 do 4. Ruch obiektu staje się chaotyczny.

Ruch z prędkością sterowaną myszą

Nasz następny przykład obrazuje sposób interaktywnego sterowania prędkością klipu za pomocą myszy.

```
// prędkość sterowana myszą
pos = new Vector (200, 200);
vel = new Vector (0, 0);
this.onEnterFrame = function () {
  vel.reset (this._xmouse / 10, this._ymouse / 10);
  pos.plus (vel);
```

```
  this._x = pos.x;
  this._y = pos.y;
};
```

Przykład jest identyczny do poprzedniego poza wierszem:

```
vel.reset (this._xmouse / 10, this._ymouse / 10);
```

Jako wartości nowej prędkości stosujemy współrzędne myszy zamiast liczb losowych. Ponieważ użyte właściwości `_xmouse` i `_ymouse` są liczone względem obiektu, więc porusza się on w kierunku myszy.

Szybkość

Szybkość i prędkość łączy podobny związek jak odległość i przemieszczenie. Prędkość to wektor (zwykle wielowymiarowy), a szybkość to skalar (zawsze jeden wymiar). Co więcej, szybkość to *długość* wektora prędkości; nazywana jest też czasem *wartością bezwzględną* prędkości. Pamiętajmy o tym, że wektor posiada wartość, jak i kierunek. Gdy odrzucimy kierunek, pozostanie tylko wartość — szybkość.

Poniżej przedstawiam sposób znajdowania szybkości z wektora prędkości.

```
v = new Vector (3, 4);
sz = v.getLength();
trace (sz);   // wyświetli: 5
```

Podobnie jak wcześniej wywoływaliśmy metodę `getLength()` do znalezienia odległości z wektora przemieszczenia, tak teraz używamy tej samej metody do znalezienia modułu wektora prędkości — szybkości.

Przyspieszenie

Przyspieszenie to *stopień zmian prędkości*. Ponieważ prędkość to stopień zmian położenia w czasie, możemy powiedzieć, że przyspieszenie to *podwójny stopień zmian położenia*. Prędkościomierz samochodu podaje szybkość w danym momencie czasu. Gdy samochód przyspiesza, wskaźnik prędkościomierza przechodzi na inną wartość. Im szybciej porusza się strzałka, tym większe przyspieszenie. Przyspieszenie to wartość wektorowa, więc posiada moduł i kierunek.

Przyjrzyjmy się trzem skryptom, które obrazują wpływ przyspieszenia na dynamiczny ruch. W sposób podobny do trzech wcześniejszych przykładów dotyczących prędkości, sprawdzimy stałe, losowe i sterowane myszą przyspieszenie.

Ruch ze stałym przyspieszeniem

Poniższy kod wykonuje ruch ze stałym przyspieszeniem.

```
// ruch ze stałym przyspieszeniem
pos = new Vector (0, 0);
vel = new Vector (0, 0);
```

```
accel = new Vector (1, 0);
this.onEnterFrame = function () {
  vel.plus (accel);
  pos.plus (vel);
  this._x = pos.x;
  this._y = pos.y;
};
```

Najpierw tworzymy trzy obiekty `Vector` reprezentujące położenie, prędkość i przyspieszenie. W procedurze obsługi `onEnterFrame()` dodajemy przyspieszenie do prędkości, a prędkość — do położenia. Następnie renderujemy nową prędkość. To wszystko.

Obiekt początkowo nie porusza się, ale później coraz szybciej pędzi na prawo. Możemy zmodyfikować wartości początkowe wektorów `pos`, `vel` i `accel`, aby uzyskać inną animację. Nic nie stoi na przeszkodzie, aby zdefiniować prędkość wskazującą w lewo, jak w poniższym kodzie.

```
vel = new Vector (-12, 0);
```

Spowoduje to początkowy ruch obiektu w lewo, ale po chwili obiekt zatrzyma się i zacznie poruszać się ze stałym przyspieszeniem w prawo.

Ruch z losowym przyspieszeniem

Nasz następny przykład wykonuje losowe przyspieszenie.

```
// ruch z losowym przyspieszeniem
pos = new Vector (200, 200);
vel = new Vector (0, 0);
accel= new Vector (0, 0);
this.onEnterFrame = function () {
  accel.reset (random(5)-2, random(5)-2);
  vel.plus (accel);
  pos.plus (vel);
  this._x = pos.x;
  this._y = pos.y;
};
```

Kod jest podobny do przykładu z losową prędkością, ale teraz zmieniamy wartości `accel.x` i `accel.y` na losowe liczby z przedziału od –2 do 2. Skrypt tworzy chaotyczny ruch, ale wygląda on inaczej niż chaotyczny ruch dla prędkości.



Przykład ten jest w zasadzie przykładem ruchów Browna, w którym to obiekt doświadcza losowych zmian sił (a co za tym idzie przyspieszenia) z różnych kierunków. Ruchom Browna przyjrzymy się dokładniej w rozdziale 11.

Ruch z przyspieszeniem sterowanym myszą

Pozostał jeszcze skrypt modyfikujący przyspieszenie na podstawie położenia myszy.

```
// przyspieszenie sterowane myszą
pos = new Vector (200, 200);
vel = new Vector (0, 0);
```

```
accel = new Vector (0, 0);
this.onEnterFrame = function () {
  accel.reset (this._xmouse / 100, this._ymouse / 100);
  vel.plus (accel);
  pos.plus (vel);
  this._x = pos.x;
  this._y = pos.y;
};
```

Siła

Siła to po prostu ciągnięcie lub pchanie w określonym kierunku. Bez siły nic w fizycznym świecie nie może się zmienić, więc cały świat byłby się do bólu statyczny. Z pewnej perspektywy możemy traktować życie jako proces ciągłej zmiany materii.

Pewnie każdy słyszał o tym, że materia w zasadzie składa się pustej przestrzeni. Atom to jądro i orbitujące wokół niego elektrony. Elektrony od jądra dzieli znaczna odległość, porównywalna z odległościami występującymi w naszym układzie planetarnym. Jak więc obiekty będące w 99,9999% puste, powodują przesunięcie innych obiektów? A jak lina utrzymuje mój ciężar, skoro jej atomy nawet się nie dotykają? W rzeczywistości między elektronami atomów występuje *siła elektromagnetyczna*, dając wrażenie „solidności", do której jesteśmy przyzwyczajeni. Poza grawitacją wszystkie siły doznawane w naszym mikroskopijnym świecie, czyli tarcie, sprężystość, magnetyzm, elektrostatyczność, to wynik siły elektromagnetycznej.



Pozostałymi dwiema podstawowymi siłami w fizyce, poza grawitacją i elektromagnetyzmem, są słabe i silne siły wiązań atomowych. Ponieważ zachodzą one na poziomie atomów, nie odczuwamy ich bezpośrednio. Siły te są jednak bardzo ważne; gdyby ich nie było, uleglibyśmy natychmiastowej dematerializacji.

Siła to wartość wektorowa z modułem i kierunkiem. Wpływa ona na ruch, zmieniając przyspieszenie. Odkrył to w XVII wieku Newton. Newton jest najbardziej znany z odkrycia grawitacji, ale jego trzy zasady dynamiki także zasługują na ogromne uznanie.

Pierwsze prawo dynamiki Newtona

Jedną z podstawowych sił mechaniki jest przyspieszenie — zmiana prędkości — które nie powstaje samo z siebie. Jeśli zmienia się prędkość obiektu, musi pojawić się siła powodująca tę zmianę. O tym dokładnie mówi *pierwsze prawo dynamiki Newtona*.

> *Jeżeli siły działające na punkt materialny równoważą się, to w inercyjnym układzie odniesienia ciało porusza się ruchem jednostajnym lub spoczywa.*

Pierwsze prawo dynamiki nazywane jest też *prawem inercji*. Inercja to opór, jaki stawia obiekt w przypadku chęci zmiany jego prędkości. Skała stoi w miejscu, dopóki nie zadziała na nią niezrównoważona siła. Podobnie dzieje się, jeśli skała porusza się, nie zwolni ani nie przyspieszy, gdy nie działa na nią tarcie czy grawitacja.

W kosmosie ciała prawie wcale nie doznają tarcia, więc mogą przebywać znaczne odległości bez większej utraty prędkości. W ten sposób możemy wysyłać sondy do systemu słonecznego, by szpiegować inne planety. Bez inercji podróż mierzona milionami kilometrów nie byłaby jeszcze możliwa (wymagałaby ogromnej ilości paliwa).

Siła niezrównoważona (siła wypadkowa)

Pierwsze prawo dynamiki Newtona mówi o równowadze sił, a zmiana ruchu następuje tylko przy jej braku. Na ciało w tym samym czasie może działać wiele sił z różnych kierunków. Na przykład w zabawie z przeciąganiem liny dwie drużyny ciągną linę w przeciwnych kierunkach. Jeśli obydwie drużyny stosują tę samą siłę, lina nie porusza się, ponieważ przyłożone do niej siły wzajemnie się znoszą. Gdy siła jest niezrównoważona, lina przesunie się w kierunku silniejszej drużyny. Różnica między dwiema siłami to siła wypadkowa.

W systemie, w którym kilka sił działa na ciało, obliczamy siłę wypadkową, sumując wszystkie wektory sił. Jeśli siła wypadkowa wynosi zero, obiekt nie zmienia swojej prędkości. Gdy siła wypadkowa jest różna od zera, obiekt przyspiesza w kierunku wskazywanym przez siłę wypadkową.

W poniższym przykładzie zdefiniowaliśmy trzy siły, a następnie dodaliśmy je do siebie, by obliczyć siłę wypadkową.

```
// deklaracja wektorów sił
silaA = new Vector (2, 0);
silaB = new Vector (-3, 2);
silaC = new Vector (0, 1);
// suma sił
silaWypadkowa = new Vector (0, 0);
silaWypadkowa.plus (silaA);
silaWypadkowa.plus (silaB);
silaWypadkowa.plus (silaC);
trace (silaWypadkowa);   // [-1, 3]
```

W tym podejściu tworzymy zerowy wektor siły wypadkowej i dodajemy do niego po kolei trzy wektory sił składowych. Łączna wartość komponentu x wynosi –1 (2 – 3 + 0), a komponentu y — 3 (0 + 2 +1).

Możemy skrócić zapis, używając metody `Vector.plusNew()`, która w jednym kroku dodaje dwa wektory i zwraca sumę jako osobny obiekt.

```
// suma sił (skrót)
silaWypadkowa = silaA.plusNew (silaB);
silaWypadkowa.plus (silaC);
trace (silaWypadkowa);   // [-1, 3]
```

Wektor `silaA` dodajemy do `silaB`, a wynik przypisujemy do `silaWypadkowa`. Aby zakończyć proces sumowania, dodajemy `silaC` do `silaWypadkowa`.



W przypadku gdy musimy dodać do siebie wiele sił, powiedzmy pięć i więcej, bardziej wydajne wydaje się zastosowanie własnej procedury dodawania zamiast `Vector.plus()`. Procedura ta może przyjmować tablicę wektorów, przechodzić przez nią w pętli i dodawać komponenty x oraz y do osobnych zmiennych. Na końcu wystarczy wpisać komponenty do nowego wektora i zwrócić go.

Drugie prawo dynamiki Newtona

Z pierwszego prawa dynamiki wnioskujemy, że siła wypadkowa powoduje przyspieszenie ruchu obiektu. Wiemy też, jak policzyć siłę wypadkową dla obiektu, dodając siły składowe. Nie wiemy jednak, jakie przyspieszenie spowoduje konkretna siła — jaki jest między tymi wektorami związek? Doświadczenie podpowiada, że popchniecie lekkiego i ciężkiego ciała z tą samą siłą powoduje szybsze poruszanie się lżejszego obiektu. Wygląda na to, że masa ciała ma wpływ na przyspieszenie.

Drugie prawo dynamiki Newtona opisuje związek między siłą, masą i przyspieszeniem.

> *Przyspieszenie powodowane przez siłę wypadkową jest wprost proporcjonalne do modułu siły wypadkowej, a odwrotnie proporcjonalne do masy ciała. Kierunek przyspieszenia i siły jest taki sam.*

Możemy powiedzieć, że im większa siła wypadkowa, tym większe przyspieszenie, a im większa masa, tym mniejsze przyspieszenie.

Drugie prawo dynamiki można wyrazić za pomocą poniższego wzoru matematycznego.

$$f = ma$$

Tutaj a to przyspieszenie obiektu, f to działająca niego siła, a m to masa obiektu. Równanie to jest często przekształcane do postaci:

$$a = f/m$$

Dzięki tej postaci potrafimy obliczyć przyspieszenie, gdy znamy przyłożoną siłę oraz masę obiektu. Wystarczy tylko podzielić siłę przez masę. Przypuśćmy, że siła wypadkowa wynosi [8, 0] (moduł równy osiem i kierunek w prawo), a masa równa jest 2. Przyspieszenie policzymy w następujący sposób.

$$f = [8,0]$$
$$m = 2$$
$$a = f/m$$
$$a = [8,0]/2$$
$$a = [4,0]$$

Uzyskaliśmy wektor przyspieszenia o wartości [4, 0] — jego kierunek jest jak przypuszczaliśmy taki sam jak działającej siły.

Załóżmy teraz, że stosujemy tę samą siłę [8, 0] dla cięższego obiektu, na przykład o masie wynoszącej 4. Przyspieszenie będzie wynosiło:

$$f = [8,0]$$
$$m = 4$$
$$a = f/m$$

$$a = [8,0]/4$$
$$a = [2,0]$$

Przyspieszenie cięższego obiektu wyniesie [2, 0] — połowę przyspieszenia lżejszego obiektu. Teraz bardzo łatwo zauważyć związek między masą i przyspieszeniem. Gdy masa wzrasta dwa razy, przyspieszenie zmniejsza się dwukrotnie.

Ruch we Flashu sterowany siłą

Aby uzyskać we Flashu wiarygodny, dynamiczny ruch bazujący na siłach, musimy mieć siłę i na jej podstawie automatycznie obliczać nowe położenie obiektu. By wykonać to zadanie, skorzystamy z poznanych do tej pory wiadomości na temat prędkości, przyspieszenia itp. Oto krótkie przypomnienie. Siły działające na obiekt sumujemy, by uzyskać siłę wypadkową. Z siły wypadkowej obliczamy przyspieszenie. Przyspieszenie zmienia prędkość poruszania się obiektu. Prędkość wpływa na położenie. Kolejność jest następująca: siły → siła wypadkowa → przyspieszenie → prędkość → położenie.

W ActionScript implementacja tego procesu to sześć oddzielnych kroków.

1. Obliczamy siły działające na obiekt.

   Wektory sił możemy policzyć dowolną metodą. W dalszej części rozdziału dowiemy się, jak obliczać siły, takie jak grawitacja czy sprężystość. Na razie w tym kroku po prostu je zdefiniujemy.

   ```
   // definicje sił
   silaA = new Vector (1, 2);
   silaB = new Vector (3, 4);
   ```

2. Dodajemy siły, by obliczyć siłę wypadkową.

   We wcześniejszej części rozdziału przedstawiłem skrótowy sposób obliczania siły wypadkowej.

   ```
   // znajdowanie siły wypadkowej
   silaWypadkowa = silaA.plusNew (silaB);
   ```

3. Znajdujemy przyspieszenie dla siły.

   Pamiętamy równanie na przyspieszenie:

   $$a = f/m$$

   W ActionScript dzielimy wektor siły wypadkowej przez masę obiektu.

   ```
   // znajdujemy przyspieszenie
   // masa została zdefiniowana wcześniej
   przysp = silaWypadkowa.scaleNew (1 / masa);
   ```

   Czasem masa poruszającego się obiektu nie ma wpływu na animację. Możemy więc zignorować masę i założyć, że ma ona wartość jednostkową. W takim przypadku równanie na przyspieszenie upraszcza się.

   $$a = f/m$$
   $$m = 1$$

$$a = f/1$$

$$a = f$$

Gdy masa wynosi 1, przyspieszenie jest równe sile wypadkowej. Z tego powodu w ActionScript możemy napisać następujące wiersze.

```
// znajdujemy przyspieszenie
przysp = silaWypadkowa;
```

W większości tworzonych animacji masa nie jest istotna, więc zakładam, że ma wartość 1 i pomijam krok 3.

4. Dodajemy przyspieszenie do prędkości.

   Wektor przyspieszenia modyfikuje wektor prędkości.

```
// znajdujemy nową prędkość
// pr została zdefiniowana wcześniej
pr.plus (przysp);
```

5. Dodajemy prędkość do położenia.

   Ten krok jest podobny do poprzedniego, ale tym razem modyfikujemy położenie.

```
// znajdujemy nowe położenie
// pol została zdefiniowana wcześniej
pol.plus (pr);
```

6. Umieszczamy obiekt w nowym położeniu.

   Ogólnie oznacza to ustawienie właściwości _x i _y klipu filmowego na komponenty *x* i *y* wektora położenia.

```
// aktualizacja położenia klipu filmowego
mc._x = pol.x;
mc._y = pol.y;
```

   Wspomniałem już w rozdziale 3., że czasem preferuję tradycyjne współrzędne kartezjańskie, a nie współrzędne ekranu Flasha. Obracam więc oś *y*, mnożąc komponent dla tej osi przez –1.

```
// użycie współrzędnych kartezjańskich
mc._x = pol.x;
mc._y = -pol.y;
```

Powyższe 6. kroków to ogólna procedura generowania dynamicznego ruchu na podstawie sił. Do tej pory poruszaliśmy się w idealnym wszechświecie — nie istniał żaden opór. Aby urealnić ruch, musimy wziąć pod uwagę tarcie.

Tarcie

Tarcie to reakcja na ruch — siła działająca w odwrotnym kierunku niż ruch obiektu. Gdy próbujemy przesunąć sofę po podłodze, dywan rywalizuje z nami za pomocą siły tarcia. Gdy nasza siła przewyższy tarcie, uzyskamy siłę wypadkową zdolną przesunąć sofę.

Pamiętajmy o tym, że tarcie to siła *przeciwdziałająca*. Nie powoduje spontanicznego przesuwania się obiektów — tarcie pojawia się tylko w odpowiedzi na ruch. Gdy sofa stoi w miejscu, nie działa na nią żadne tarcie. Gdy próbujemy ją poruszyć, siła tarcia przeciwdziała ruchowi. Kiedy dwa ciała stałe są ze sobą złączone, pojawiają się dwa główne rodzaje tarcia: statyczne i kinetyczne.

Tarcie kinetyczne

Tarcie kinetyczne nazywane jest także tarciem ślizgowym. Gdy krążek hokejowy pędzi po lodzie, siła tarcia cały czas go zwalnia. Stopień tej siły jest zwykle stały. Oznacza to, że spowalnianie krążka także jest stałe. Wykres spowolnienia przypomina spowolnienie kwadratowe.

Wartość tarcia zależy od tego, jaka jest siła prostopadła (nazywana też *siłą normalną*) do powierzchni, po której się przesuwamy. Na tym samym lodzie cięższy obiekt będzie miał większe tarcie niż obiekt lekki, ponieważ działa na niego większa siła grawitacji.

Oto wzór na tarcie kinetyczne w jego typowej postaci.

$$f_k = \mu_k N$$

W tym równaniu f_k to siła tarcia kinetycznego, μ_k to współczynnik tarcia, a N to moduł z siły normalnej łączącej obydwa ciała. Wartość μ_k (μ wymawiamy jak „mi") znajduje się typowo w przedziale od 0 do 1. Im wyższa wartość współczynnika, tym większe tarcie dla danej siły dociskającej.

Jeśli chcemy używać tarcia kinetycznego w animacji Flasha, która nie jest symulacją fizyczną, lepiej od razu przypisać wartość tarcia do f_k i nie przejmować się współczynnikiem ani siłą dociskającą (normalną). Można nawet od razu wyliczyć *przyspieszenie* tarcia, a nie *siłę* tarcia. Poprawny, ale dłuższy sposób polega na znalezieniu wartości siły tarcia, a następnie — na wyliczeniu przyspieszenia ze wzoru $f = ma$. Oto przykładowe obliczenia dla tej metody.

$$f_k = 10$$
$$a = f/m$$
$$m = 5$$
$$a = 10/5$$
$$a = 2$$

Możemy założyć, że przyspieszenie tarcia będzie stałe (nie będzie zależeć od masy obiektu), więc przypiszemy wartość przyspieszenia na stałe.

$$a = 2$$

Gdy mamy przyspieszenie powodowane przez tarcie kinetyczne, proces jego implementacji nie należy do najprostszych (przynajmniej w porównaniu z uproszczonym tarciem płynów, którym zajmiemy się za chwilę). Poniższy kod ActionScript wykonuje obliczenia dla tarcia kinetycznego.

```
// obliczanie tarcia kinetycznego w jednym wymiarze
prX = 0;
przyspTarcia = .8;

this.onEnterFrame = function () {
  prX +=przyspX;
  przyspX = 0;
  if (prX > 0) {
    prX = Math.max (0, prX - przyspTarcia);
  } else if (prX < 0) {
    prX = Math.min (0, prX + przyspTarcia);
  }
  this._x += prX;
};

this.onMouseDown = function () {
  przyspX = 15;
};
```

Gdy uruchomimy ten przykład, obiekt stoi w miejscu, dopóki nie naciśniemy przycisku myszy generującego przyspieszenie. Obiekt natychmiast zaczyna się poruszać, ale później stopniowo zwalnia z powodu tarcia kinetycznego.

Logika kodu jest następująca. Jeśli prędkość w poziomie jest większa od zera (czyli obiekt porusza się w prawo), tarcie kinetyczne działa w druga stronę — w lewo. Z tego powodu zmniejszamy aktualną prędkość o wartość tarcia (`prX - przyspTarcia`).

Tarcie nie może zmienić kierunku prędkości. Innymi słowy, gdy prędkość była dodatnia, odejmowanie przyspieszenia tarcia nie może spowodować, że stanie się ujemna. Tarcie spowalnia prędkość obiektu do zera, ale nie powoduje ruchu w przeciwnym kierunku. Korzystam z funkcji `Math.min()` i `Math.max()`, aby upewnić się, że odejmowanie nie przekroczy zera.

Tarcie statyczne

Gdy obiekt stoi na powierzchni, możliwe jest tarcie statyczne. Tarcie statyczne jest najczęściej silniejsze od tarcia kinetycznego, co wyjaśnia, dlaczego trudniej wprawić coś w ruch, niż tylko kontynuować ruch.

Obiekt nie rozpocznie ruchu, dopóki nie popchniemy go odpowiednio mocno. Gdy przekroczymy próg siły, obiekt nagle zaczyna się poruszać. Sanki na stoku potrafią stać niewzruszone. Działa na nie siła grawitacji, ale siła tarcia między sankami a śniegiem jest taka sama. Ponieważ siły mają identyczną wartość, ale przeciwne kierunki, wzajemnie się znoszą, więc sanki stoją w miejscu. Gdy jednak lekko popchniemy sanki, dodatkowa siła przezwycięża siłę tarcia statycznego i sanki zaczynają się poruszać.

Tarcie statyczne zachodzi tylko wtedy, gdy dwa obiekty, których ono dotyczy, nie poruszają się względem siebie. Gdy istnieje ruch, mamy już do czynienia z tarciem kinetycznym, które zwykle ma mniejszą wartość niż tarcie statyczne.

Oto wzór na tarcie statyczne.

$$f_s = \mu_s N$$

W tym równaniu f_s to siła tarcia statycznego, μ_s to współczynnik tarcia, a N to moduł z siły normalnej łączącej obydwa ciała. Wartość μ_s znajduje się typowo w przedziale od 0 do 1. Im wyższa wartość współczynnika, tym większe tarcie dla danej siły dociskającej.

Do tej pory nie musiałem w żadnej animacji korzystać z tarcia statycznego, ale należy teraz wspomnieć o takiej możliwości. Sposób implementacji jest prosty.

1. Sprawdzamy, czy obiekt nie porusza się (czyli jego prędkość wynosi zero).
2. Obliczamy siłę wypadkową działającą na obiekt.
3. Obliczamy siłę tarcia statycznego działającego na obiekt.
4. Jeśli siła wypadkowa jest mniejsza do siły tarcia, nie robimy nic.
5. W przeciwnym przypadku zaczynamy przyspieszać obiekt.

Tarcie dla cieczy i gazów

Do tej pory mówiliśmy o tarciu między obiektami stałymi. Z tarciem płynów spotykamy się na co dzień, gdy pływamy się w wodzie. Im szybciej porusza się obiekt, tym większe odczuwa tarcie. Gdy na przykład prowadzimy samochód, czujemy większy opór powietrza, gdy jedziemy szybciej.

Wykonanie dokładnej symulacji tarcia w płynach wymaga złożonych równań. Ponieważ nie zależy nam aż tak na dokładności w dynamicznych animacjach Flasha, skorzystamy z przybliżenia, które w większości przypadków sprawdza się doskonale. Musimy tylko podporządkować się głównej zasadzie: *większa prędkość oznacza większą utratę prędkości spowodowaną tarciem*. Innymi słowy, utrata prędkości jest *wprost proporcjonalna* do samej prędkości.

Proporcje często wyrażamy w procentach. Na przykład podatek VAT wyrażamy w procentach. Im więcej płacimy za produkt, tym większy odprowadzamy od niego podatek. Tarcie w płynach to jakby podatek od prędkości, który możemy zdefiniować w *procentach utraty prędkości na jednostkę czasu*. We Flashu podstawową jednostką czasu jest klatka, więc ze względów praktycznych tarcie w płynach określimy jako *procent utraty prędkości na klatkę*.

Na przykład siła tarcia może powodować zmniejszanie prędkości ciała o 10% w każdej klatce. Jeśli obiekt rozpoczyna ruch z prędkością 100, to w następnej klatce jego prędkość spadnie do 90 (100 – 10). W kolejnej klatce prędkość spadnie o 9 (10% z 90), czyli do wartości 81. Następny spadek wyniesie 8,1 do 72,9 itd.

Poniższy kod obrazuje sposób uzyskania tego efektu w ActionScript.

```
// obliczanie tarcia w płynach w jednym wymiarze
prX = 0;
tarciePlynu = 0.07;
this.onEnterFrame = function () {
  prX +=przyspX;
  przyspX = 0;
  prX *= (1 - tarciePlynu);
  this._x += prX;
};
```

```
this.onMouseDown = function () {
  przyspX = 15;
};
```

Przykład ten jest bardzo podobny do wcześniejszego przykładu z tarciem kinetycznym. Różnice są tylko dwie. Pierwsza to deklaracja zmiennej tarcia.

```
tarciePlynu = 0.07;
```

Tarcie płynu definiujemy w procentach; w tym przypadku strata wynosi 7% wartości prędkości w każdej klatce. Druga różnica to wiersz, w którym tarcie wpływa na prędkość.

```
prX *= (1 - tarciePlynu);
```

W ten sposób redukujemy `prX` o wartość określoną w `tarciePlynu`. W naszym przykładzie `1 - tarciePlynu` daje wynik 0,93. Pomnożenie prędkości przez tę wartość zmniejsza ją o 7%.

Jak możemy zauważyć, kod tarcia w płynach jest szybszy i prostszy niż w przypadku tarcia kinetycznego. W większości animacji używam tarcia w płynach. Główny wyjątek stanowi witryna Axis Interactive v2 (*www.axis-media.com*), gdzie użyłem tarcia kinetycznego dla pływającego menu.

Jeśli głównym naszym zmartwieniem jest szybkość działania, korzystajmy z tarcia w płynach. Jeśli poszukujemy większej dokładności, używamy tarcia kinetycznego, gdzie obiekty toczą się lub ślizgają, i tarcia w płynach dla cieczy, gazów i przestrzeni kosmicznej (która także zawiera niewielkie ilości gazu). Gdy porównamy tarcia do przejść, okaże się, że tarcie kinetyczne to w zasadzie spowolnienie kwadratowe, a tarcie w płynach to spowolnienie wykładnicze (szczegóły znajdziemy w rozdziale 7.).

Grawitacja w kosmosie

Między dwoma ciałami zawsze występuje siła grawitacji — nawet między dwoma ziarnami piasku. Dopiero gdy zlepimy ze sobą ogromną ilość materii, siła grawitacji jest na tyle duża, że możemy ją zauważyć; przykładem mogą być gwiazdy i planety.

Poniżej przedstawiam klasyczny wzór Newtona na siłę grawitacji między dwoma ciałami.

$$F_g = \frac{Gm_1m_2}{r^2}$$

F_g to wartość siły grawitacji, m_1 i m_2 to masy ciał. G to stała grawitacji wynosząca $6{,}67 \times 10^{-11}$ Nm^2/kg^2, a r to odległość środków mas obydwu ciał.

Siła grawitacji jest *wprost proporcjonalna* do masy obydwu obiektów. Jeśli dwukrotnie zwiększymy jedną z mas, siła grawitacji wzrośnie dwukrotnie, a gdy dwukrotnie zwiększymy obydwie masy, siła wzrośnie czterokrotnie.

Z drugiej strony pojawia się odwrotna proporcjonalność do kwadratu odległości. Jeśli odległość między ciałami zwiększymy dwukrotnie, siła grawitacji zmniejszy się do jednej czwartej oryginalnej wartości. Jeśli dwukrotnie zmniejszymy odległość między ciałami, siła grawitacji wzrośnie czterokrotnie.

We Flashu interesuje nas przede wszystkim efekt, a nie dokładność. Z tego powodu nie jest w zasadzie potrzebna stała grawitacji ani nawet masa obiektów, gdy sądzimy, że nie odgrywa żadnej znaczącej roli.

Ważne jest, aby zapamiętać, że siła grawitacji jest odwrotnie proporcjonalna do kwadratu odległości między obiektami. By znaleźć siłę grawitacji, możemy określić oddziaływanie między obiektami, znaleźć odległość między nimi (r) i podzielić oddziaływanie przez kwadrat odległości. .

Przyjrzyjmy się teraz przykładowi implementacji grawitacji w ActionScript. Najpierw przedstawmy całość kodu.

```
// grawitacja wokół kliknięty punktu
this.pos = new Vector (150, 100);
this.vel = new Vector (5, 0);
this.accel = new Vector (0, 0);
this.friction = 0;

// właściwości grawitacji
this.anchor = new Vector (250, 200);
this.strength = 5000;

this.doForce = function () {
    // obliczanie siły grawitacji
    this.netForce = this.pos.minusNew (this.anchor);
    var r = this.netForce.getLength();
    this.netForce.setLength (-this.strength / (r*r));
};

this.move = function () {
    // obliczanie nowego położenia
    this.vel.plus (this.netForce);
    this.vel.scale (1 - this.friction);
    this.pos.plus (this.vel);
    // rendering położenia
    this._x = this.pos.x;
    this._y = this.pos.y;
};

this.onEnterFrame = function () {
    this.doForce();
    this.move();
};

this.onMouseDown = function () {
    this.anchor.reset (this._parent._xmouse,
                       this._parent._ymouse);
};
```

Skrypt należy umieścić wewnątrz klipu filmowego z widoczną grafiką, na przykład wewnątrz klipu przedstawiającego piłkę. Pamiętajmy też o dołączeniu klasy `Vector` (*vector_class.as*). Po uruchomieniu przykładu możemy kliknąć dowolne miejsce i spowodować orbitowanie wokół kliknięty punktu. Przyjrzyjmy się teraz dokładniej kodowi.

Pierwsza część skryptu deklaruje główne wektory dotyczące położenia, prędkości, przyspieszenia i tarcia.

```
this.pos = new Vector (150, 100);
this.vel = new Vector (5, 0);
this.accel = new Vector (0, 0);
this.friction = 0;
```

Ustawienia te znamy już z wcześniejszych przykładów. Następnie pojawiają się dwa wiersze związane z grawitacją.

```
this.anchor = new Vector (250, 200);
this.strength = 5000;
```

Wektor `archor` to punkt na scenie, wokół którego orbituje obiekt. Właściwość `strength` określa siłę przyciągania między punktem a poruszającym się obiektem. Wypróbujmy różne wartości tego parametrem, by zobaczyć jego wpływ na przebieg animacji.

Później pojawia się metoda `doForce()`, której kod jest następujący.

```
this.doForce = function () {
    // obliczanie siły grawitacji
    this.netForce = this.pos.minusNew (this.anchor);
    var r = this.netForce.getLength();
    this.netForce.setLength (-this.strength / (r*r));
};
```

To właśnie w tej metodzie liczymy siłę wypadkową. Pierwszy wiersz kodu znajduje wektor przemieszczenia między wektorem punktu a wektorem położenia obiektu za pomocą odjęcia jednego od drugiego.

```
this.netForce = this.pos.minusNew (this.anchor);
```

Wprowadzam tutaj siłę wypadkową (`this.netForce`), ponieważ wektor siły wypadkowej jest w rzeczywistości *równoległy* do wektora przemieszczenia. W ten sposób uzyskuję odpowiedni *kierunek* wektora siły, a jego *wartość* — długość — dostosowuję przy użyciu poniższego kodu.

```
var r = this.netForce.getLength();
this.netForce.setLength (-this.strength / (r*r));
```

Pierwszy wiersz znajduje *odległość* `r` między punktem a obiektem za pomocą obliczenia długości wektora przemieszczenia (umieszczonego chwilowo w `this.netForce`). (Pamiętajmy nasz opis odległości i przemieszczenia we wcześniejszej części rozdziału). Na końcu nadajemy wektorowi siły odpowiednią długość, obliczając odwrotność odległości do kwadratu.

Znamy już siłę wypadkową, więc musimy ją jeszcze zamienić na prędkość i położenie. Zajmuje się tym metoda `move()` przy użyciu poniższego kodu.

```
this.move = function () {
    // obliczamy nowe położenie
    this.vel.plus (this.netForce);
    this.vel.scale (1 - this.friction);
    this.pos.plus (this.vel);
    // rendering położenia
    this._x = this.pos.x;
    this._y = this.pos.y;
};
```

Ta procedura powinna już być znana z wcześniejszych przykładów. Przyspieszenie dodajemy do prędkości, która jest skalowana tarciem. Następnie prędkość dodajemy do położenia. W tym przykładzie siłę wypadkową dodajemy bezpośrednio do położenia zamiast do przyspieszenia, ponieważ w naszym przykładzie siła ta jest równa przyspieszeniu.

Po tej ciężkiej pracy pozostało jeszcze wykonanie na klatkach pętli, która poprowadzi animację.

```
this.onEnterFrame = function () {
    this.doForce();
    this.move();
};
```

Dla każdej klatki procedura obsługi `onEnterFrame()` najpierw wywołuje metodę `doForce()`, aby policzyć nową siłę wypadkową, a następnie metodę `move()`, by wykonać animację obiektu.

Na końcu dodajemy jeszcze interakcję na kliknięcia myszy, definiując procedurę obsługi `onMouseDown()`.

```
this.onMouseDown = function () {
    this.anchor.reset (this._parent._xmouse,
                       this._parent._ymouse);
};
```

Gdy naciśniemy przycisk myszy, zmieniamy wektor `archor` na wartości współrzędnych myszy. Powyższy przykład obrazuje sposób symulacji grawitacji między obiektami w przestrzeni kosmicznej. Teraz przyjrzymy się grawitacji działającej na Ziemi.

Grawitacja w pobliżu powierzchni

Na Ziemi odczuwana siła grawitacji ma mniej więcej stałą wartość i działa tylko dla osi pionowej. W zasadzie to właśnie grawitacja określa, gdzie jest pion. Siła grawitacja stara się nas wepchnąć do środka Ziemi. Z tego powodu *dół* oznacza po prostu kierunek wskazujący na środek Ziemi. Kierunek ten jest prostopadły do powierzchni naszej planety.

W grawitacji występującej między dwoma obiektami w kosmosie masa każdego ciała ma wpływ na siłę grawitacji i przyspieszenie. Jednak na Ziemi przyspieszenie powodowane przez grawitację jest ogólnie tak samo niezależne do masy i wymiarów obiektu (nie uwzględniamy tarcia).

Kilka wieków temu w sławnym eksperymencie Galileusz obalił pogląd, że cięższe ciała spadają szybciej niż lżejsze. Eksperyment polegał na spuszczeniu dwóch kul o różnej wadze z krzywej wieży w Pizie. Obydwie kule armatnie zostały puszczone w tym samym czasie i w tym samym czasie uderzyły w ziemię. Nasza planeta ma tak ogromną masę w porównaniu z innymi obiektami znajdującymi się na Ziemi, że w równaniu na siłę grawitacji masa lżejszego ciała (obiekt spadający na Ziemię) jest w zasadzie niezauważalna.



Powodem wolniejszego spadania piórka od kamienia jest opór powietrza. W próżni obydwa obiekty spadają z taką samą szybkością.

Ponieważ siła grawitacja na powierzchni jest stała, stałe też jest powodowane przez nią przyspieszenie. Z tego względu bardzo łatwo zaimplementować w ActionScript ruch powodowany taką grawitacją. Przyspieszenie dla grawitacji definiujemy raz na początku filmu i bez żadnych zmian stosujemy w obliczeniach.

Poniższy przykład obrazuje sposób implementacji grawitacji występującej na powierzchni planety.

```
// przyspieszone opadanie spowodowane siłą grawitacji
this.pos = new Vector (this._x, this._y);
this.vel = new Vector (0, 0);
this.friction = 0;

// właściwości grawitacji
this.gravAccel = new Vector (0, .5);

this.move = function () {
    // obliczanie nowego położenia
    this.vel.plus (this.gravAccel);
    this.vel.scale (1 - this.friction);
    this.pos.plus (this.vel);
    // rendering położenia
    this._x = this.pos.x;
    this._y = this.pos.y;
};

this.onEnterFrame = function () {
    this.doForce();
    this.move();
};

this.onMouseDown = function () {
    this.pos.reset (this._parent._xmouse,
                    this._parent._ymouse);
    this.vel.reset (0, 0);
};
```

W trakcie działania skryptu każde kliknięcie myszy umieszcza klip filmowy w nowym położeniu i powoduje jego swobodne spadanie. W procedurze obsługi `onMouseDown()` wektor położenia `this.pos` ustawiany jest na współrzędne myszy, a wektor prędkości `this.vel` — na zero.

Przyspieszenie obiektu sterowane jest właściwością przyspieszenia grawitacyjnego `this.gravAccel`, zdefiniowanego na początku skryptu jako wektor [0, 0,5] wskazujący w dół. Używamy takiego samego procesu co w poprzednich przykładach w celu przekształcenia przyspieszenia w dynamiczny ruch.

Elastyczność (sprężystość)

Prawa dotyczące elastyczności możemy napotkać w wielu miejscach: rozciągając sprężynę, skacząc na trampolinie a nawet ubierając sweter. Gdy rozciągamy elastyczny obiekt, stara się on przeciwdziałać naszemu ruchowi. Pojawia się siła elastyczności nazywana też siłą sprężystości.

Nie wszystkie materiały są elastyczne; niektóre rozsypują się lub łamią, gdy je ściśniemy. Jednak istnieją materiały, które możemy rozciągać; cząsteczki tych obiektów oddalają się od siebie. W trakcie trwania procesu rozciągania atomy nadal wpływają na siebie siłą elektromagnetyczną (jedna z czterech podstawowych sił). To właśnie z niej bierze swój początek siła sprężystości.

Stan spoczynkowy

Elastyczny obiekt znajduje się w stanie spoczynku, gdy nic go nie popycha i nie ciągnie. Możemy bez problemu wyrwać obiekt z tego błogiego stanu spoczynku. Im bardziej rozciągasz ciało, tym większą siłę sprężystości ono generuje.

Istnieje bezpośredni związek między przemieszczeniem w czasie rozciągania a wartością siły sprężystości. Jeśli na przykład elastyczną linę rozciągniemy o jeden centymetr i otrzymamy daną wartość siły, po rozciągnięciu liny na dwa centymetry siła ta zwiększy się dwukrotnie. Musimy pamiętać o tym, że ciało sprężyste zawsze dąży do osiągnięcia stanu równowagi.

Prawo Hooke`a

Pamiętajmy, że to dzięki matematyce jesteśmy w stanie opisać związki za pomocą wzorów. Jak już się przekonaliśmy, związek między rozciąganiem a siłą sprężystości jest prosty. Z tego powodu jego wzór także jest prosty.

$$f = -kd$$

Równanie to nazywane jest prawem Hooke'a. Informuje ono, że siła sprężystości jest wprost proporcjonalna do wychylenia z punktu równowagi.



Robert Hooke odkrył to równanie w XVII wieku. Był kolegą Isaaca Newtona.

Zmienna F w równaniu to siła sprężystości. Zmienna d to przemieszczenie w trakcie rozciągania. W fizyce przemieszczenie podaje się typowo w metrach, a siłę — w Newtonach (metryczna jednostka siły). W animacjach Flasha nie musimy jednak martwić się o jednostki.

Stała sprężystości

Wartość k w prawie Hooke'a to stała sprężystości. Jest to zawsze wartość dodatnia, która określa stopień naprężenia materiału elastycznego; zawsze zawiera się w przedziale od 0 do 1. Im większe k, tym sztywniejsza jest substancja elastyczna.

Na przykład gruba lina elastyczna ma większe k niż cienka. Z tego powodu grubsza lina przy rozciąganiu generuje większą siłę sprężystości. Możemy też powiedzieć inaczej: to my musimy włożyć więcej siły w rozciągnięcie grubszej liny niż cienkiej.

Kierunek siły sprężystości

Dlaczego w równaniu pojawia się znak minus? Ponieważ siła sprężystości działa w *kierunku przeciwnym* do przemieszczenia w trakcie rozciągania. Przypuśćmy, że przymocowaliśmy jeden koniec liny, a drugi ciągniemy w prawo. Siła sprężystości stara się ciągnąć naszą rękę w lewo. Z tego powodu przemieszczenie jest dodatnie, a siła sprężystości — ujemna i na odwrót.

Implementacja w ActionScript

```
// sprężystość wokół punktu określonego kursorem myszy
this.pos = new Vector (this._x, this._y);
this.vel = new Vector (0, 0);
this.accel = new Vector (0, 0);
this.friction = .1;

// właściwości sprężystości
this.anchor = new Vector (150, 100);
this.tautness = .25;

this.doForce = function () {
    // obliczenie siły sprężystości
    this.netForce = this.pos.minusNew (this.anchor);
    this.netForce.scale (-this.tautness);
};

this.move = function () {
    // obliczenie nowego położenia
    this.vel.plus (this.netForce);
    this.vel.scale (1 - this.friction);
    this.pos.plus (this.vel);
    // rendering położenia
    this._x = this.pos.x;
    this._y = this.pos.y;
};

this.onEnterFrame = function () {
    this.doForce();
    this.move();
};

this.onMouseDown = function () {
    this.anchor.reset (this._parent._xmouse,
                       this._parent._ymouse);
};
```

Najpierw pojawiają się standardowe definicje wektorów położenia, prędkości i przyspieszenia oraz tarcie w cieczach ustawione na 10%. Następnie deklarujemy dwie właściwości dotyczące sprężystości: wektor równowagi `this.anchor` i stałą sprężystości `this.tautness`.

Punkt równowagi jest początkowo ustawiony na (150, 100), ale możemy go zmienić kliknięciem na aktualne współrzędne kursora myszy. Klip filmowy jest przeciągany wokół punktu równowagi: przyspiesza z jednej strony, zwalnia z drugiej, powraca z przyspieszeniem — obiekt oscyluje wokół punktu równowagi, zmniejszając powoli ogólną prędkość.

Główną metodą skryptu jest `doForce()`, gdzie obliczamy elastyczność.

```
this.doForce = function () {
    // obliczenie siły sprężystości
    this.netForce = this.pos.minusNew (this.anchor);
    this.netForce.scale (-this.tautness);
};
```

Pamiętamy postać równania prawa Hooke'a dla sprężystości: $F = -kd$. Przemieszczenie rozciągania d znajdujemy, odejmując w pierwszym wierszu kodu wektor punktu równowagi od wektora położenia.

```
this.netForce = this.pos.minusNew (this.anchor);
```

Teraz właściwość `this.netForce` przechowuje wektor przemieszczenia wskazujący z klipu filmowego na punkt równowagi, który jest źródłem siły sprężystości.

Drugi wiersz przeprowadza mnożenie $-kd$ za pomocą metody `Vector.scale()`.

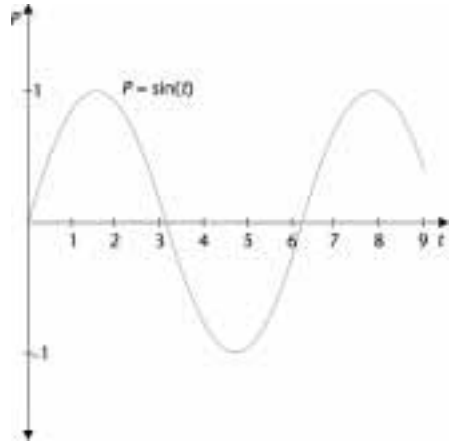
```
this.netForce.scale (-this.tautness);
```

Teraz wektor przechowuje siłę sprężystości.

Pozostałe metody: `move()`, `onEnterFrame()` i `onMouseDown()` są takie same jak we wcześniejszym przykładzie z grawitacją w kosmosie.

Ruch falowy

Wiele rzeczy oscyluje. Możemy powiedzieć, że ich ruch jest cykliczny lub okresowy. Wykres ich położenia w czasie tworzy falę sinusoidalną pokazaną na rysunku 8.2.



Rysunek 8.2. *Typowa fala sinusoidalna*

Fale sinusoidalne, jak przedstawiona, mogą bazować na funkcji sinus lub kosinus. My korzystamy z funkcji sinus.

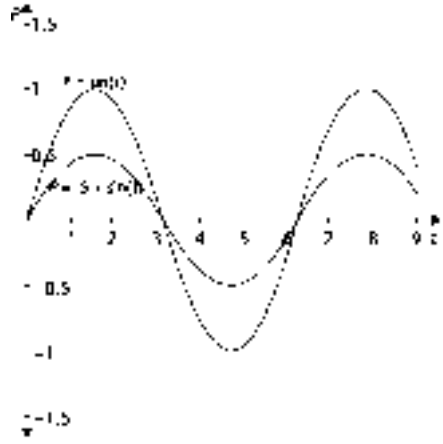
$$P = \sin(t)$$

W tym równaniu t to kąt w radianach. Koło to 2π radianów, więc jeden okres fali sinusoidalnej także trwa 2π radianów. Po przyjrzeniu się wykresowi możemy stwierdzić, że fala przecina oś X zaraz za wartością 3. Jest to połowa okresu, która odpowiada 3,1415… radianów lub 180°. Krzywa wraca do punktu początkowego, by rozpocząć kolejny okres w punkcie 6,2832 radianów (2π) lub 360°.

Amplituda

Amplituda to wartość oscylacji fali. Amplitudę możemy także zdefiniować jako maksymalne odchylenie fali od punktu równowagi w każdym z kierunków. Tradycyjna fala sinusoidalna posiada amplitudę o wartości 1. Rysunek 8.3 przedstawia dwie fale sinusoidalne z różnymi amplitudami. Amplituda fali odniesienia z rysunku jest dwa razy większa niż drugiej fali o wzorze $P = 0{,}5 \times \sin(t)$.

Rysunek 8.3. *Zmiana amplitudy fali sinusoidalnej*



Ogólny wzór na amplitudę fali sinusoidalnej możemy zapisać następująco.

$$P = amp \times \sin(t)$$

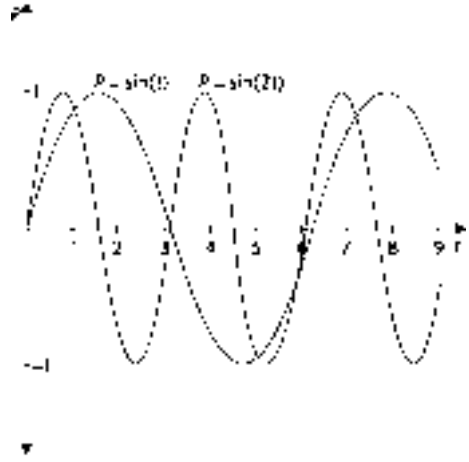
Wartość *amp* skaluje całą funkcję sinus na nową amplitudę.

Częstotliwość

Częstotliwość to stopień oscylacji fali. Mnożąc parametr t w równaniu $\sin(t)$, modyfikujemy częstotliwość fali. Rysunek 8.4 porównuje falę $\sin(t)$ z falą $\sin(2t)$.

Jak możemy się przekonać, patrząc na rysunek, pomnożenie t przez 2 podwaja częstotliwość. Podobnie pomnożenie t przez 0,5 zmniejszy ją o połowę. Poniższe równanie określa ogólną zależność.

$$P = \sin(2\pi \times częst \times t)$$



Rysunek 8.4.
Zmiana częstotliwości fali sinusoidalnej

Dlaczego π znalazło się w tym równaniu? Funkcja sin(*t*) powtarza się co 2π radianów (360°), co oznacza, że naturalna częstotliwość fali wynosi 1/(2π). Jeśli więc chcemy nadać fali nową częstotliwość, musimy najpierw pomnożyć *t* przez 2π, by znormalizować częstotliwość do 1 (1/(2π) × 2π = 1), a dopiero potem przez częstotliwość.



W fizyce i elektronice częstotliwość jest zwykle mierzona w hercach (Hz), które odpowiadają ilości cykli na sekundę. Dźwięk jest falą, której częstotliwość możemy wyrazić w hercach. Ludzkie ucho potrafi wychwycić dźwięki od około 20Hz do 20 kHz.

Okres

Okres to długość czasu, jaką fala poświęca na jeden obieg. Okres jest bezpośrednio powiązany z częstotliwością. Obydwie wartości są wzajemnymi odwrotnościami, więc możemy je wyrazić poniższymi równaniami.

$$okres = 1 \,/\, częstotliwość$$
$$częstotliwość = 1 \,/\, okres$$

Jeśli podzielimy jeden przez częstotliwość, otrzymamy okres, a jeśli podzielimy jeden przez okres, uzyskamy częstotliwość. Jeśli mamy do czynienia z obiektem oscylującym cztery razy na sekundę, jego częstotliwość wynosi 4 Hz, a okres — 0,25 sekundy.

Jeśli zastąpimy częstotliwość okresem we wcześniej podanym równaniu na falę sinusoidalną, uzyskamy poniższy wzór.

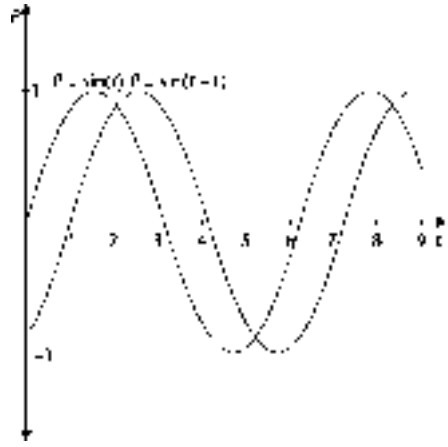
$$P = \sin(2\pi \times t / okres)$$

Okres fali jest niezależny od jej amplitudy — możemy zmienić okres bez zmiany amplitudy i na odwrót. Choć okres i częstotliwość są ze sobą powiązane, nie można mylić obu wartości, ponieważ stanowią dwie różne wielkości opisujące tę samą falę.

Przesunięcie czasu

Zmienna przesunięcia czasu powoduje przesunięcie fali w lewo lub w prawo na osi poziomej (patrz rysunek 8.5). Jak możemy zauważyć, odjęcie od t jedynki w $\sin(t - 1)$ przesuwa falę o jeden w prawo. Dodanie do t jeden, przesuwa falę w lewo o jeden.

Rysunek 8.5.
Przesunięcie czasowe fali sinusoidalnej

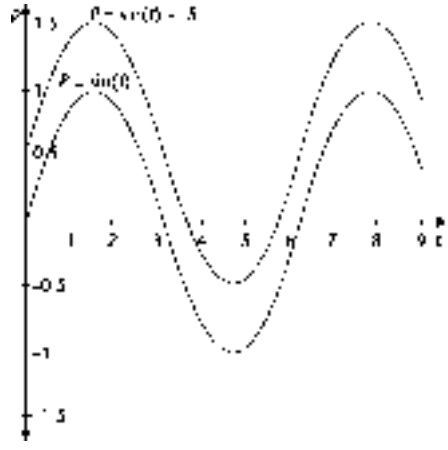


Przesunięcie czasu jest blisko związane z *przesunięciem fazy*, które także przemieszcza falę na osi czasu. Jednak przesunięcie fazy określane jest nieco inaczej — jako kąt. Możemy na przykład powiedzieć, że dwie fale są przesunięte względem siebie o 90° (co jest prawdą, gdy porównamy funkcje $\sin(t)$ i $\cos(t)$).

Offset

Zmienna offsetu to przesunięcie fali w górę lub w dół na osi pionowej (patrz rysunek 8.6). Offset definiuje środek fali. Domyślnie offset wynosi zero, więc fala oscyluje wokół osi czasu, czyli położenia 0. Na rysunku 8.6 funkcja $P = \sin(t) + 0{,}5$ posiada offset wynoszący 0,5, więc środek oscylacji jest przesunięty do góry o pół jednostki.

Rysunek 8.6.
Zmiana offsetu fali sinusoidalnej



Równanie fali

Wszystkie właściwości możemy umieścić w jednym równaniu.

$$P_w = amp \times \sin\left[\left(t - przesunięcie\right) \times 2\pi / okres\right] + offset$$

A oto implementacja równania w ActionScript jako funkcja `Math.wave()`.

```
Math.wave = function (t, amp, period, timeShift, offset) {
  return amp * Math.sin ((t-timeShift) * (2*Math.PI) / period) + offset;
};
```

Funkcja `Math.wave()` zapewnia wygodny interfejs do obliczania oscylacji ruchu. Wystarczy przekazać do funkcji intuicyjne właściwości fali i dostać z powrotem aktualną wartość fali bez martwienia się zbytnio o obliczenia.

Klasa WaveMotion

Utworzyłem klasę `WaveMotion`, aby zhermetyzować oscylacje i umożliwić prostą modyfikację właściwości fali. `WaveMotion` rozszerza klasę `Motion` (omówioną w poprzednim rozdziale). Jako klasa pochodna `WaveMotion` dziedziczy metody i właściwości klasy `Motion`, jak i dodaje własne. Dziedzicznie oznacza, że do filmu musimy dołączyć klasę `Motion`, aby wszystko działało prawidłowo. ActionScript jest bardzo pobłażliwy w sprawie błędów. W szczególności nie sprawdza, czy kod klasy bazowej został dołączony, gdy kompiluje klasy pochodne. Napisałem więc własny kod, który sprawdza istnienie klasy bazowej `Motion`.

```
if (typeof _global.Motion != 'function') trace (">> Błąd: brak klasy bazowej Motion");
```

Powyższy kod sprawdza istnienie konstruktora klasy bazowej `Motion` i jeśli go nie znajdzie, wysyła odpowiedni komunikat do okna *Output*. W następnych krokach deklaruje konstruktor `WaveMotion`, definiuje dziedziczenie i dodaje nowe metody do prototypu klasy.

Konstruktor WaveMotion

Pamiętamy, że konstruktor `Motion` przyjmował pięć parametrów: `obj`, `prop`, `begin`, `duration` i `useSeconds`. Do konstruktora fali dodałem dwa nowe parametry: `amp` i `period`. Amplituda i okres to dwie najważniejsze właściwości fali, ponieważ definiują jej kształt.

Poniższy kod definiuje konstruktor klasy `WaveMotion`.

```
_global.WaveMotion = function (obj, prop, begin, amp, period, duration, useSeconds) {
  this.superCon (obj, prop, begin, duration, useSeconds);
  this.setOffset (begin);
  this.setAmp (amp);
  this.setPeriod (period);
};
```

Funkcja przyjmuje siedem parametrów. Pięć z nich przekazujemy do konstruktora klasy bazowej, używając metody `superCon()` (omówionej w rozdziale 2.). Następnie inicjalizujemy właściwości `offset`, `amp` i `period` za pomocą odpowiednich metod ustawiających.

Następnie powodujemy dziedziczenie przez `WaveMotion` metod po `Motion`, używając metody `extend()` omówionej w rozdziale 2.

```
WaveMotion.extend (Motion);
```

Po ustawieniu konstruktora i dziedziczenia zaczynamy przypisywać metody do prototypu klasy. Często tworzę tymczasową zmienną przechowującą prototyp obiektu, by skrócić sobie zapis metod.

```
var WMP = WaveMotion.prototype;
```

Od tego momentu zamiast `WaveMotion.prototype` mogę używać `WMP`.

WaveMotion.getPosition()

Metoda `getPosition()` to podstawa każdej klasy dziedziczącej po `Motion`. To właśnie w niej definiujemy związek między czasem i położeniem. W klasie `Motion` metoda ta jest pusta, ponieważ klasy pochodne mają ją przysłonić, definiując własną wersję.

W klasie `WaveMotion` używamy równania fali.

```
WMP.getPosition = function (t) {
  if (t == undefined) t = this.$time;
  return this.$amp * Math.sin ((t-this.$timeShift) * (2*Math.PI) / this.$period) +
  ➥this.$offset;
};
```

Metody ustawiania i pobierania

Metody `WaveMotion.setAmp()` i `getAmp()` sterują amplitudą fali.

```
WMP.setAmp = function (a) {
  if (a != undefined) this.$amp = a;
};

WMP.getAmp = function () {
  return this.$amp;
};
```

Warto zauważyć, że funkcja ustawiająca dokonuje prostego sprawdzenia danej. Zmienia amplitudę, tylko wtedy gdy rzeczywiście przekażemy parametr.

Metody `WaveMotion.setPeriod()` i `getPeriod()` sterują okresem fali.

```
WMP.setPeriod = function (p) {
  if (p != undefined) this.$period = p;
};
```

```
WMP.getPeriod = function () {
  return this.$period;
};
```

Metoda `setPeriod()` sprawdza daną w podobny sposób, w jaki robi to `setAmp()`.

Metody `WaveMotion.setFreq()` i `getFreq()` modyfikują i zwracają częstotliwość fali.

```
WMP.setFreq = function (f) {
  this.setPeriod (1 / f);
};

WMP.getFreq = function () {
  return 1 / this.getPeriod();
};
```

Zauważmy, że te metody działają inaczej niż pozostałe metody pobierania i ustawiania. W odróżnieniu od amplitudy i okresu częstotliwość nie jest przechowywana jako właściwość obiektu `WaveMotion`. Konwertujemy częstotliwość na okres i przy obliczaniu fali korzystamy z okresu. Wspomniałem już wcześniej o tym, że częstotliwość i okres to wzajemne odwrotności. Z tego powodu dzielimy jeden przez częstotliwość, aby uzyskać okres.

Metody `WaveMotion.setTimeShift()` i `getTimeShift()` ustawiają i pobierają przesunięcie fali na osi czasu.

```
WMP.setTimeShift = function (t) {
  if (t != undefined) this.$timeShift = t;
};

WMP.getTimeShift = function () {
  return this.$timeShift;
};
```

W podobny sposób metody `setOffset()` i `getOffset()` przesuwają falę w wymiarze *położenia*.

```
WMP.setOffset = function (f) {
  if (f != undefined) this.$offset = f;
};

WMP.getOffset = function () {
  return this.$offset;
};
```

Dodałem jeszcze jedną metodę ustawiającą, `WaveMotion.setWavePhysics()`, która umożliwia ustawienie za jednym zamachem wszystkich czterech właściwości fali.

```
WMP.setWavePhysics = function (amp, period, timeShift, offset) {
  this.setAmp (amp);
  this.setPeriod (period);
  this.setTimeShift (timeShift);
  this.setOffset (offset);
};
```

Właściwości ustawiania i pobierania

Po zdefiniowaniu metod pobierania i ustawiania możemy utworzyć właściwości ustawiania i pobierania za pomocą metody `addProperty()` z Flasha MX.

```
with (WaveMotion.prototype) {
  addProperty ("amp", getAmp, setAmp);
  addProperty ("offset", getOffset, setOffset);
  addProperty ("period", getPeriod, setPeriod);
  addProperty ("freq", getFreq, setFreq);
  addProperty ("timeShift", getTimeShift, setTimeShift);
}
```

Na końcu usuwamy tymczasową zmienną `WMP`

```
delete WMP;
```

i wysyłamy prosty komunikat o wczytaniu klasy do okna *Output*.

```
trace(">> Wczytano klasę WaveMotion");
```

Używanie WaveMotion

Oto przykład tworzenia kopii `WaveMotion`.

```
falaX = new WaveMotion (this, "_x", 200, 80, 2, 0, true);
```

Jeśli umieścimy ten kod w pierwszej klatce klipu filmowego, klip zacznie oscylować w poziomie wokół wartości 200 pikseli z 2 sekundami na okres. Domyślnie będzie tak oscylował w nieskończoność. Tabela 8.1 wymienia argumenty konstruktora `WaveMotion`.

Tabela 8.1. *Parametry konstruktora WaveMotion*

Parametr	Typ	Wartość	Opis
`obj`	**referencja**	`this`	**Modyfikowany obiekt.**
`prop`	**tekst**	`"_x"`	**Nazwa właściwości, która oscyluje.**
`begin`	**liczba**	`200`	**Środek fali.**
`amp`	**liczba**	`80`	**Rozmiar oscylacji.**
`period`	**liczba**	`2`	**Czas trwania jednego cyklu oscylacji.**
`duration`	**liczba**	`0`	**Łączny czasu trwania oscylacji; jeśli wynosi 0, ruch jest nieskończony.**
`useSeconds`	**Boolean**	`true`	**Znacznik określający, czy chcemy korzystać z sekund, czy klatek jako licznika.**

Po utworzeniu kopii `WaveMotion` wybrany klip oscyluje automatycznie. Możemy zmieniać wartości parametrów fali w trakcie tego ruchu. Możemy na przykład dodać poniższy kod, aby w każdej klatce zwiększać amplitudę fali o jeden.

```
this.onEnterFrame = function() {
  falaX.amp++;
};
```

Ponieważ stale zwiększamy amplitudę, obiekt zaczyna stopniowo wahać się na coraz to większe odległości.

Możemy nawet dać użytkowi większą kontrolę nad amplitudą fali.

```
this.onEnterFrame = function() {
  falaX.setAmp (this._parent._ymouse);
};
```



W powyższym kodzie używam metody `setAmp()` do zmiany amplitudy. Mógłbym skorzystać z właściwości `amp` jak poprzednio, ale chciałem przedstawić obydwa podejścia. Właściwość `amp` wydaje się prostsza, ale `setAmp()` jest bardziej bezpośrednia, a co za tym idzie typowo szybsza.

Oto inny fragment kodu, który steruje dwiema właściwościami fali na podstawie położenia myszy.

```
this.onEnterFrame = function() {
  falaX.setAmp (this._parent._ymouse);
  falaX.setOffset (this._parent._xmouse);
};
```

Podobnie jak wcześniej położenie myszy w pionie steruje amplitudą. Jednak tym razem położenie myszy w poziomie odpowiada za offset. Powoduje to, że środek oscylacji zawsze znajduje się w miejscu, w którym jest kursor myszy.

Wnioski

Fizyka to fascynująca dziedzina nauki. W tym rozdziale poznaliśmy jej podstawy: prędkość, przyspieszenie, siłę i związki między nimi. Przyjrzeliśmy się bliżej pewnym rodzajom sił, takim jak tarcie, sprężystość i grawitacja. Obliczaliśmy i stosowaliśmy ją w naszych dynamicznie generowanych animacjach. Na końcu przyjrzeliśmy się fizyce fal i zaprojektowaliśmy własną klasę `WaveMotion`, która pozwala tworzyć prawie dowolne oscylacje sinusoidalne. Po dwóch intensywnych rozdziałach dotyczących ruchu, przejdziemy teraz do zabawy z kolorami.