helion.pl
KSIĘGARNIA INTERNETOWA WYDAWNICTWO

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 5.
Techniki zaawansowane

Autor: praca zbiorowa
ISBN: 83-7197-566-X
Tytuł oryginału: Flash 5 Studio
Format: , stron: 710
Zawiera CD-ROM

Chociaż sam program mieści się na niewielkiej płycie, to możliwości Flasha są ogromne. Pojawił się na scenie internetowej kilka lat temu, ale teraz wykrystalizował się jako poważne narzędzie zdolne do tworzenia atrakcyjnych wizualnie witryn internetowych wysokiej jakości. Lecz to jeszcze nie wszystko: implementacja skryptu ActionScript w piątej wersji Flasha przekształciła program w prawdziwe środowisko programistyczne, które umożliwia realizację w pełni interaktywnych projektów od interfejsu użytkownika do wewnętrznego przechowywania danych. W niniejszej książce przedstawimy pełen zakres możliwości Flasha, a celem autorów jest pokazanie kolejnych zastosowań tej użytecznej aplikacji.

Do powstania tej książki przyczynili się programiści, którzy w codziennej pracy stawiają sobie te same pytania co Ty: Jak powinienem to zrobić? Jak mogę to zrobić? Jakie mam możliwości do wyboru? W jaki sposób ta technologia rozwinie się w przyszłości? Jak mogę zapewnić, że witryna, nad którą pracuję, będzie odwiedzana przez maksymalną liczbę internautów? Jakie są wymagania moich klientów i jak mogę im sprostać? Wszyscy oni stali na linii ognia, walcząc z klientami, technologią czy kodem. Za pomocą tej książki chcą podzielić się z Tobą zdobytą wiedzą.

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

Spis treści

O Autorach 11

Niezbadany świat Flasha 17

Głębia, rozmach i zrozumienie 17
Podział książki 18
Preludium 18
Tworzenie zawartości 18
ActionScript 18
Dynamiczna zawartość 18
Konteksty 18
Układ książki 19
Zawartość płyty CD-ROM 20

Część I Preludium 21

Rozdział 1. Zasady projektowania witryny internetowej 23

Użyteczność 24
Użyteczność Flasha 24
Najczęściej spotykane problemy związane z użytecznością witryn internetowych wykonanych we Flashu 25
Adresy witryn internetowych poświęconych użyteczności 26
Rola projektanta witryn internetowych 27
Definiowanie wiadomości 27
Definiowanie publiczności 29
Rozwiązanie problemu 31
Projekt konstrukcyjny 32
Rozmiar plików, wstępne ładowanie, strumieniowanie i akcja loadMovie 33
Prototyp funkcjonalności 36
Użyteczność i doświadczenie użytkownika 38
Wybór i formatowanie tekstu 40
Spójność: narzędzie użyteczności 42
Widoczne elementy projektu 44
Twórczość pomimo ograniczeń 44
Szkicowanie, miniaturki i bazgroły 46
Szkicowanie 47
Kompozycja 48
Elementy kompozycji 49
Kierunek czytania 51
Szkic kompozycyjny 51

Schematy kolorów ... 52
Wybór schematu kolorów ... 56
Przykłady kolorów ... 57
Spójność po raz drugi ... 59
Profil graficzny ... 61
Ulepszanie projektów ... 62
Metoda drastyczna ... 62
Projektowanie witryny internetowej Titoonia.com — studium projektu ... 64
Przegląd witryny ... 64
Wiadomość ... 65
Docelowa publiczność ... 66
Projekt strukturalny ... 68
Struktura katalogów ... 69
Tworzenie prototypu ... 70
Szkicowanie ... 71
Kompozycja ... 72
Kolor ... 73
Czcionki ... 75
Tworzenie zawartości witryny Titoonia ... 76
Profil graficzny ... 81
Zakończenie ... 82

Część II Tworzenie zawartości ... 83

Rozdział 2. Sprytne klipy ... 85

Wszechstronność sprytnych klipów ... 89

Rozdział 3. Flash i zewnętrzne aplikacje 3D ... 103

Podstawy teoretyczne 3D ... 104
Perspektywa liniowa i rzut ortogonalny ... 104
Tworzenie zawartości 3D we Flashu ... 107
Ręczne trasowanie we Flashu ... 107
Ogólne wskazówki dotyczące stosowania polecenia Trace Bitmap ... 114
Tworzenie 3D w innych aplikacjach ... 116
Swift 3D ... 116
Rendering w programie Swift 3D ... 122
3D Studio Max ... 126
Vecta 3D ... 127
Pluginy Illustrate! 5.0 i Vecta 3D dla 3D Studio Max ... 128
Amorphium Pro ... 129
Poser ... 131
Który produkt wybrać? ... 131
Interaktywność ... 132
Prawdziwy trzeci wymiar ... 133
Optymalizacja zawartości 3D na potrzeby sieci ... 133
Darmowe modele w sieci ... 137
Ograniczony rendering ruchu ... 135
Zmniejszanie powierzchni ... 135
Renderowanie części ujęć ... 137
Zakończenie ... 138

Rozdział 4. Animacja kreskówek we Flashu ... 139
Początki ... 140
Planowanie animacji ... 141
Interaktywna zawartość ... 142
Podstawy animacji ... 143
Przenikanie ujęć — Onion Skinning ... 144
Animacja we Flashu ... 145
Projektowanie postaci ... 145
Skacząca piłka ... 147
Skaczący człowiek ... 149
Cykle ruchu ... 150
Optymalizacja ... 155
Zakończenie ... 157

Rozdział 5. Wideo we Flashu ... 159
Wybór właściwego klipu ... 160
Zastosowanie programów QuickTime Pro i Photoshop ... 168
Optymalizacja ... 176
Zmiana kolorów wideo ... 178
Skalowanie filmu ... 179
Zakończenie ... 180

Rozdział 6. Dźwięk we Flashu ... 181
Tworzenie stołu mikserskiego ... 182
Tworzenie i importowanie dźwięków ... 182
Obiekt koloru we Flashu 5 ... 197
Kontrolowanie dźwięku pośrednio ... 200
Rozbudowanie ćwiczenia ... 202
Zakończenie ... 208

Rozdział 7. Efekty przezroczystości ... 209
Aktualizacja oryginalnych obrazków ... 216
Optymalizacja rozmiaru pliku animacji ... 219
Zawansowana animacja map bitowych ... 221
Zakończenie ... 222

Rozdział 8. Maskowanie i efekty tekstowe ... 225
Warstwy ... 225
Podstawy maskowania ... 226
Animowanie masek ... 229
Zamaskowany tekst ... 229
Efekt reflektora ... 232
Efekt koła kolorów ... 235
Maski i ActionScript ... 239
Efekty tekstowe ... 245
Proste efekty tekstowe ... 245
Bardziej złożone efekty tekstowe ... 250
Zakończenie ... 263

Rozdział 9. Przyciski i menu 265
Przyciski 265
Tworzenie przycisków z podpowiedziami 266
Przyciski w klipach filmowych 268
Zastępowanie zachowania przycisku 269
Menu 274
Proste menu 275
Pływające menu 277
Poziome menu hierarchiczne 279
Pionowe menu hierarchiczne 281
Zakończenie 286

Część III ActionScript 287

Rozdział 10. Podstawy programowania w środowisku ActionScript 289
Najważniejsze zagadnienia programowania 290
Zmienne 290
Rodzaje zmiennych 291
Typy danych 292
Struktury programowania w języku ActionScript 296
Wyrażenia 296
Instrukcje 296
Bloki 297
Sterowanie przepływem 297
Konstrukcje rozgałęzień 297
Konstrukcje pętli 298
Funkcje 300
Obiekty 302
Przykłady: zmienne i sterowanie przepływem 303
Przykłady: funkcje 307
Przykłady: obiekty 312
Zakończenie 316

Rozdział 11. Integracja środowiska programowania ActionScript 317
Listwy czasowe, klipy filmowe i obiekty 317
Główna listwa czasowa 317
Unikanie blokowania listwy czasowej 318
Praca z klipami filmowymi 322
Praca z kilkoma listwami czasowymi 323
Praca z osadzonymi klipami filmowymi 323
Rysunki, klipy filmowe i przyciski 324
Kiedy wykonywany jest kod? 325
Wielokolorowe kule bilardowe wykonane za pomocą jednego symbolu 326
Klipy filmowe jako obiekty — rozwijane menu 334
Pole tekstowe 334
Pole listy 335
Zakończenie 350

Rozdział 12. Tworzenie efektów wizualnych z zastosowaniem języka ActionScript 351
Tablice 351
Efekty wizualne 352
Ścigacz znacznika myszy 352
Efekt cząsteczki: ogień 357
Przechwytywanie klawisza 363
Manipulacja linią 375
Zakończenie 387

Rozdział 13. Stosowanie prostych procedur w grach 389
Prosta gra w ping-ponga 395
Co powinienem zrobić teraz? 412
Typowe elementy gry 412
Zakończenie 414

Rozdział 14. Programowanie gier 415
Kosmiczna gra 415
Modyfikacja gry 436

Rozdział 15. Flash i trzeci wymiar 439
Co jest możliwe, a co praktyczne? 439
Wszystko o okręgach 441
Sinus i cosinus 442
Ruch po okręgu 443
Oś Z 446
Obrót wokół osi Y 447
Prawdziwa perspektywa 448
Obrót wokół osi X 450
Przemieszczanie kilku punktów 451
Obrót wokół osi Z 451
Obrót wokół osi Y 454
Obrót wokół osi X 456
Obiekt trójwymiarowy 456
Zakończenie 462

Rozdział 16. Obiektowe gry Flasha 463
Wytyczne 463
Ograniczenia Flasha jako środowiska gier 464
Tworzenie gier mimo ograniczeń 465
Szablon gry Flasha 467
Duszki gameSprite 467
Świat gier 468
Klipy filmowe zachowania 472
Wygląd i działanie gry 474
Na początku… 477
Dynamika obrotów 478
Spotkanie z wrogiem 485

Łączenie w całość ... 487
Plik Savior02.fla ... 487
Plik Savior03.fla ... 490
Plik Savior.fla ... 496
Zakończenie ... 498

Część IV Dynamiczna zawartość ... 501

Rozdział 17. Dynamiczna zawartość uzyskiwana z plików tekstowych ... 503

Dynamiczna zawartość uzyskiwana z plików tekstowych ... 504
Ładowanie danych tekstowych do Flasha ... 504
Baner reklamowy z możliwością aktualizacji ... 517
Zakończenie ... 525

Rozdział 18. Dynamiczne aplikacje internetowe ... 527

Interfejs ... 528
Architektura projektu ... 528
Planowanie z wyprzedzeniem ... 530
Komunikacja silnik — interfejs ... 532
Koszyk sklepowy Flasha ... 535
Architektura pliku FLA ... 537
Standardowe klony ... 541
Dane z pliku tekstowego ... 542
Dlaczego film został zaprojektowany w ten właśnie sposób? ... 543
Kod ... 543
Główne funkcje ... 544
Kod paska przewijania ... 550
Kod programu ładującego ... 552
„Uwielbiam, gdy udaje mi się zrealizować plan” ... 553
Techniki rozwiązywania problemów ... 554
Zakończenie ... 556

Rozdział 19. Flash i PHP ... 557

Zasady stosowania PHP z Flashem ... 557
Połączenia klient-serwer ... 558
Interpreter PHP oparty na CGI ... 558
Komunikacja między Flashem i PHP ... 559
Narzędzia ... 563
Serwer WWW Apache ... 564
PHP4 — preprocesor hipertekstu ... 566
Twoja pierwsza strona w języku PHP ... 567
Dokumentacja PHP4 ... 568
Co można zrobić z PHP i Flashem? ... 569
Zagadnienia związane z wydajnością PHP ... 569
Flash i PHP w akcji ... 570
Zakończenie ... 577

Rozdział 20. Flash i XML 579
Same fakty 579
Czy jest się czym przejmować? 580
Natura bestii 581
Podstawowa składnia języka XML 582
Zastosowanie języka XML w środowisku programowania ActionScript Flasha 5 584
Obiekt XML 584
Obróbka dokumentów XML 587
Testowanie kodu XML 590
Ładowanie dokumentu XML 593
Uzyskiwanie dostępu do bazy danych przy użyciu języka XML 598
Pisanie kodu ASP 601
Ulepszanie kodu ASP 607
Dodawanie funkcji przeszukiwania 609
Zakończenie 612

Część V Konteksty 613

Rozdział 21. Połączenie Flasha i HTML 615
Wprowadzenie do języka HTML i Flasha 616
Osadzanie w całym oknie przeglądarki 619
Proporcje filmu i okna 620
Osadzanie procentowe i o stałym rozmiarze 625
Osadzanie w ramkach 627
Wyskakujące okienka 628
Jeden film kontra kilka filmów 630
Jeden film 630
Kilka filmów 631
Filmy ułożone w stosy 632
Wykrywanie Flasha i flashowe witryny internetowe 633
Flash jako obrazek w tekście 634
Nagłówki Flasha 635
Zastosowanie ramek dla nagłówków Flasha 635
Strony Flasha ze stronami HTML 638
Zakończenie 638

Rozdział 22. Ładowanie wstępne i strumieniowanie 641
Co to jest strumieniowanie? 642
Co to jest ładowanie wstępne? 643
Ekrany ładujące, taktyka różnorodności i reakcje użytkownika 644
Wstęp do konstrukcji 644
Testowanie filmów strumieniowanych i Bandwidth Profiler Flasha 645
Modyfikacja kolejności ładowania 647
Generowanie raportów o rozmiarze 647
Demonstracja strumieniowania animacji i dźwięku 650
Analiza filmu 651
Podstawowa technika ładowania wstępnego 653

Przybliżony pasek ładowania wstępnego ... 655
Analiza filmu ... 656
Przemyślenia końcowe ... 660
Bardziej precyzyjny pasek ładowania wstępnego ... 661
Analiza filmu ... 661
Przemyślenia końcowe ... 664
Modularna witryna internetowa Flasha ... 665
Analiza filmu ... 665
Przemyślenia końcowe ... 670
Zakończenie ... 671

Rozdział 23. Optymalizacja Flasha dla wyszukiwarek ... 673

Typy wyszukiwarek ... 673
Szperacze WWW ... 674
Planowanie witryny przyjaznej wyszukiwarce ... 674
Strony wejściowe ... 675
Obsługa standardowych witryn HTML ... 675
Witryny internetowe z ramkami ... 676
Witryny internetowe utworzone we Flashu ... 676
Wybór słów kluczowych i docelowego natężenia ruchu w sieci ... 677
Inne rozważania związane z procesem wyboru słów kluczowych ... 678
Gdy już wybrałeś słowa kluczowe... ... 678
Znaczniki <META> i inne elementy kodu wspomagające pracę wyszukiwarek ... 680
Co to jest znacznik <META>? ... 680
Znaczniki <META> i słowa kluczowe ... 681
Komentarze HTML ... 682
Ukryte pola wejściowe ... 683
Znaczniki obrazków ... 684
Nazwy plików, adresy internetowe i łącza ... 685
Wykluczanie stron z wyszukiwarek ... 685
Plik robots.txt ... 686
Szperacze i znacznik <META> ... 687
Monitorowanie ruchu w sieci i listingi ... 688
Zgłaszanie stron do wyszukiwarek ... 689
Szczegóły dotyczące poszczególnych wyszukiwarek ... 690
Ćwiczenie ... 697
Zakończenie ... 701

Dodatki ... 703

Skorowidz ... 705

Rozdział 15.
Flash i trzeci wymiar

Odkąd powstał Flash, projektanci pracujący w tym programie usiłują dodać do swoich projektów trzeci wymiar. Wraz z ukazaniem się na rynku aplikacji, takich jak Vecta3D i Swift 3D, umieszczenie przestrzennej zawartości na stronie WWW stało się stosunkowo proste. Od tej pory elementy 3D stosowano z różnym powodzeniem w wielu witrynach internetowych i grach w trybie *online*. Efekty trójwymiarowe niezmiennie przyciągają uwagę internautów, lecz łatwo jest wpaść w pułapkę, stosując je w nieodpowiednich miejscach. Chociaż tematem tego rozdziału jest tworzenie trójwymiarowych obiektów we Flashu, sam jestem przeciwnikiem nadmiernego wykorzystywania efektów 3D w sieci. Zestaw narzędzi przeznaczonych do tworzenia grafiki trójwymiarowej może oczywiście pomóc w opracowaniu niezwykle atrakcyjnych i niezapomnianych efektów, niemniej jednak podczas projektowania należy pamiętać o ich wadach. Elementy trójwymiarowe mogą znacznie obciążyć procesor, a zbyt częste ich stosowanie niekorzystnie wpływa na cały projekt.

Dzięki wprowadzeniu środowiska ActionScript do Flasha 4 i poszerzeniu jego możliwości w 5. wersji tej aplikacji, zamiast polegać na predefiniowanych obliczeniach, możemy przeprowadzać kalkulacje 3D w czasie rzeczywistym. Mimo udoskonaleń wprowadzonych w nowym odtwarzaczu Flasha 5, wymagania dotyczące procesora podczas wykonywania skomplikowanych obliczeń we Flashu (z zachowaniem przyzwoitej prędkości filmu) nadal są dosyć wysokie. Są jednak sposoby na obejście tych ograniczeń.

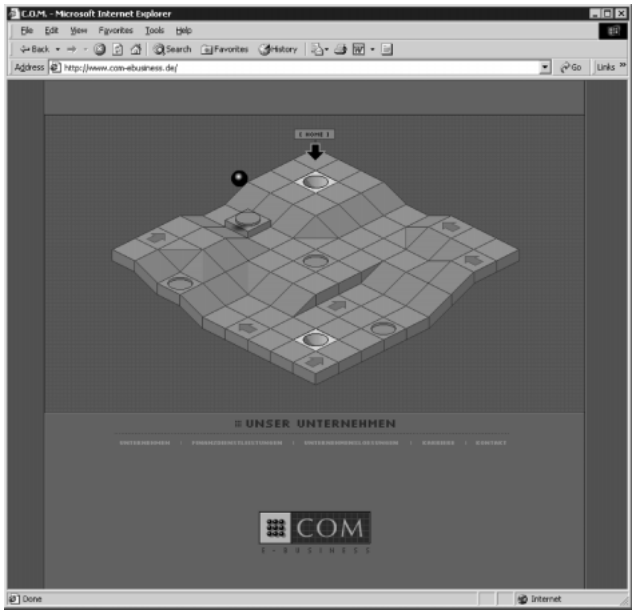
Co jest możliwe, a co praktyczne?

Mając na uwadze wspomniane wyższej utrudnienia, należy pamiętać, że przeprowadzanie obliczeń 3D w czasie rzeczywistym we Flashu nie musi oznaczać tworzenia złożonych form geometrycznych, gdyż leży to poza granicami rozsądku. Bardzo interesujące efekty można natomiast uzyskać za pomocą określania pozycji „obiektów" w przestrzeni, nadając im przestrzenne współrzędne. Przykładem zastosowania takiej techniki jest trójwymiarowa gra w rugby o nazwie Conversion Kings (*www.sportal.co.za/conversionkings*), która została wykonana przez moją firmę dla korporacji Sportal.

Aby obliczyć współrzędne x, y i z piłki w przestrzeni zastosowaliśmy prosty mechanizm 3D podobny do tych, którymi zajmiemy się w tym rozdziale. Stadionowi, stanowiskom i podłożu również nadaliśmy przestrzenne współrzędne, co umożliwiło nam wykrywanie kolizji piłki z tymi obiektami. Następnie na tło 3D nałożyliśmy przezroczystą „matrycę" trójwymiarowych współrzędnych, na której umieściliśmy ruch piłki.

Ponieważ współrzędne obliczane są w wirtualnym środowisku 3D, możemy także zmienić kąt i perspektywę widoku. Dzięki temu uzyskujemy widoki *z* różnych perspektyw, a nawet funkcję „powtórki" przedstawianą spoza stanowisk. Jedynymi elementami, które należy obliczyć metodą „ujęcie po ujęciu", są piłka i jej cień. Ich wykonanie można potraktować jako przykład realnego zastosowania trzeciego wymiaru we Flashu.

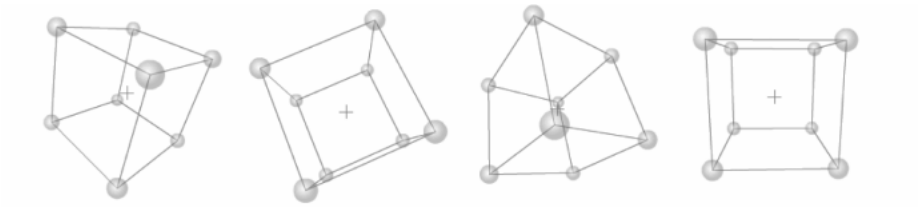
Podobnym projektem jest interfejs użytkownika przedstawiony na stronie WWW pod adresem *www.com-ebusiness.de*. I tym razem obliczenia 3D zastosowane zostały do określenia pozycji piłki w aksonometrycznej przestrzeni 3D, gdyż w tej grze piłka powinna zachowywać się zgodnie z prawami fizyki.



Takie połączenie prerenderowanych obiektów trójwymiarowych i obliczeń 3D w czasie rzeczywistym stanowi wielki potencjał, zwłaszcza na polu programowania gier we Flashu. Jak na razie, możliwości tej techniki nadal nie zostały w pełni zbadane. Trzeci wymiar tego rodzaju może być również *z* powodzeniem stosowany w projektowaniu interfejsów multimedialnych.

Wszystko o okręgach

Zanim weźmiemy pod uwagę takie zastosowania, niezbędne jest zrozumienie sposobu przeprowadzania tych „prostych" obliczeń 3D. To właśnie będzie stanowić główny temat tego rozdziału, w którym przedstawię także proces generowania we Flashu prostego „mechanizmu" 3D. Na jego serce składa się kilka okręgów i obliczeń kątów. W tym celu przygotowałem pliki w formacie FLA. Ich analiza powinna Ci ułatwić zrozumienie wszystkich operacji przeprowadzonych w tym rozdziale.



Pliki te, zapisane w formacie FLA, znajdziesz na płycie CD-ROM dołączonej do książki. Zalecam, abyś miał do nich dostęp w trakcie pracy nad ćwiczeniami. Jednym z najlepszych sposobów uczenia się jest eksperymentowanie, dlatego też warto poświęcić wystarczająco dużo czasu na przeanalizowanie poszczególnych etapów ćwiczenia i przykładów.

Na początek powinniśmy się dokładnie zastanowić, jakie efekty chcemy uzyskać. Naszym celem jest utworzenie we Flashu 5 efektów 3D przy użyciu języka ActionScript. Końcową wersję projektu (który za chwilę wykonamy) umieściłem w pliku *sample_13.fla*. Wykorzystałem w nim język ActionScript do obliczenia współrzędnych sześcianu w przestrzeni i stworzenia iluzji trzeciego wymiaru. Wyraz „iluzja" użyty w tym kontekście nie oznacza bynajmniej, że ćwiczenia przedstawione w tym rozdziale dotyczą tworzenia pseudotrójwymiarowych efektów. Wyraz ten został użyty dlatego, ponieważ obiekt trójwymiarowy umieszczony na dowolnej dwuwymiarowej powierzchni nie jest niczym innym jak iluzją. Uzyskuje się ją za pomocą kombinacji właściwości *x* i *y* poszczególnych punktów w trójwymiarowym obiekcie i skalowanie ich podczas ruchu punktów w przestrzeni. Nasz sześcian przypomina wyglądem obiekt trójwymiarowy dzięki przestrzeganiu podczas rysowania odpowiednich reguł, umożliwiających szybkie skojarzenie i wyobrażenie sobie obiektu umieszczonego w przestrzeni trójwymiarowej. Ponieważ żyjemy w trójwymiarowym świecie, intuicyjnie rozpoznajemy takie efekty perspektywy, jak zmniejszanie się obiektów wraz z oddalaniem się ich od nas, czy zbieganie się linii w niewidocznych punktach.

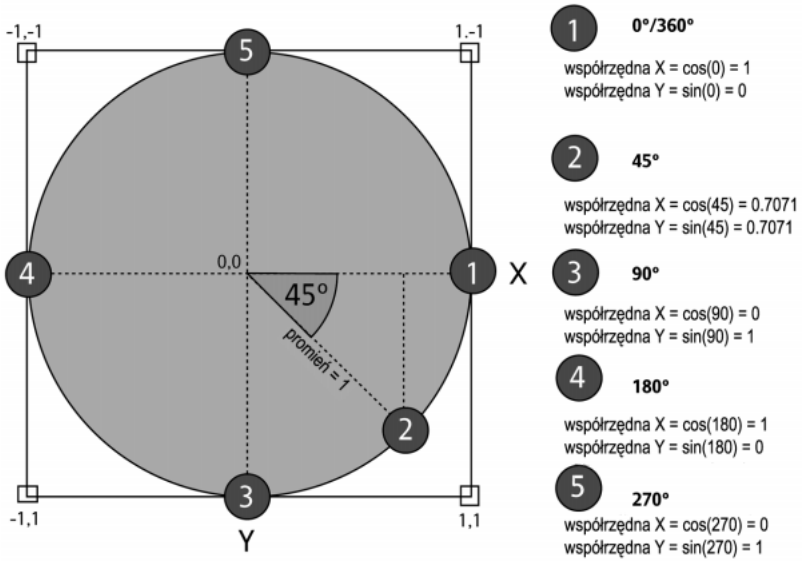
W pliku *sample_13.fla* kluczową rolę odgrywa iluzja. Dzięki niej elmenty w obiekcie wyglądają tak, jakby obracały się wokół punktu centralnego. Jeśli używałeś Flasha 3, na pewno próbowałeś animować obiekt wzdłuż spłaszczonej owalnej ścieżki w celu symulowania

obrotu 3D wokół punktu centralnego. Jest to właśnie jeden ze sposobów imitowania efektów trójwymiarowych w środowisku ActionScript. Jeśli potrafimy obliczyć kolistą ścieżkę zamiast animować ją, jesteśmy na dobrej drodze do utworzenia we Flashu trójwymiarowych efektów w czasie rzeczywistym.

Sinus i cosinus

W szkole podstawowej na pewno się uczyłeś o związku funkcji trygonometrycznych **sinus (sin)** i **cosinus (cos)** z okręgami. Nie będziemy wyjaśniać działania tych funkcji, gdyż nie jest to naszym zadaniem. Jeśli chciałbyś dowiedzieć się więcej o funkcjach trygonometrycznych, zajrzyj do podręczników lub skorzystaj z niezliczonych materiałów dostępnych w Internecie i szkołach. Przyjrzyjmy się teraz funkcjom trygonometrycznym i przekonajmy się, co mogą nam zaoferować.

Mając podaną wartość kąta, możemy zastosować funkcje sinus i cosinus do obliczenia współrzędnych punktu, w którym linia narysowana pod tym kątem przecina się z kołem o środku w punkcie, z którego rysowana jest linia. Podczas symulacji trzeciego wymiaru funkcje sinus i cosinus służą zatem do obliczania kolistej ścieżki wokół punktu centralnego. Cosinus reprezentuje wartość poziomą (X), a sinus wartość pionową (Y) współrzędnej. Poniżej przedstawiliśmy ilustrację działania tych dwóch funkcji.



Wartości sinusa i cosinusa dla danego kąta oparte są na okręgu o promieniu długości jednej jednostki, zwanym również **kołem jednostkowym** (jak na rysunku). Promień reprezentuje odległość między punktem środkowym (0,0) i krawędzią koła. Z tymi danymi równanie dla obliczenia współrzędnej *x* wygląda tak jak zostało przedstawione poniżej.

```
współrzędnaX = promień * cos (kąt)
```

A równanie dla współrzędnej *y* tak jak poniżej.

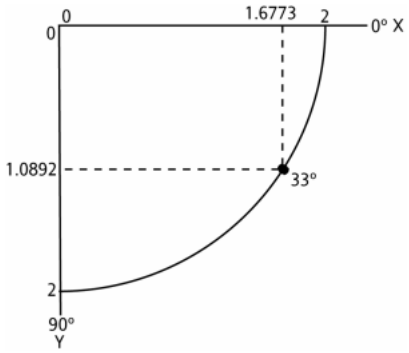
```
współrzędnaY = promień * sin (kąt)
```

Im większa będzie wartość *promienia*, tym większe będzie koło.

Poniżej przedstawione zostały równania dla kąta o wartości 33° z promieniem o wartości 2.

$X = 2 * \cos(33) = 2 * 0.8387 = 1.6773$

$Y = 2 * \sin(33) = 2 * 0.5446 = 1.0892$



Inaczej niż to miało miejsce w poprzednich wersjach, we Flashu 5 funkcje trygonometryczne są funkcjami wbudowanymi i możemy uzyskać do nich dostęp za pomocą nowego obiektu `Math`. W kodzie ActionScript możemy napisać równania w sposób, który został przedstawiony poniżej.

```
współrzednaX = promień * Math.cos (kąt);
współrzednaY = promień * Math.sin (kąt);
```

Wydaje się to proste, lecz jest oczywiście pewien „kruczek" — kąty stosowane przez obiekty `Math.cos` i `Math.sin` mierzone są w **radianach**, a nie w stopniach. Jeśli jednak wolisz używać stopni, możesz przekształcić je za pomocą poniższego równania.

```
radiany = stopnie * (π / 180)
```

Wartość π (pi) zastosowaną w powyższym równaniu można również uzyskać za pomocą nowego obiektu Flasha 5 — `Math`. W języku ActionScript równanie to będzie wyglądać tak jak poniżej.

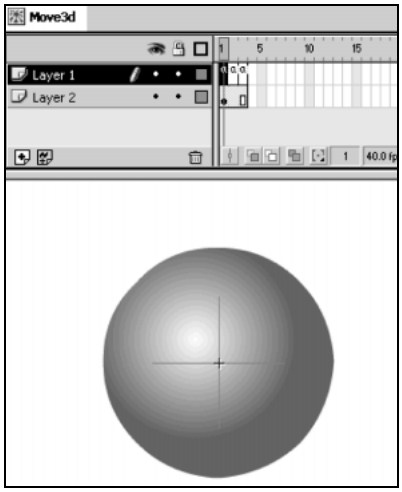
```
radiany = kąt * (Math.PI / 180);
współrzednaX = promień * Math.cos (radiany);
współrzednaY = promień * Math.sin (radiany);
```

Równania te umożliwią Ci obliczenie współrzędnych *x* i *y* dowolnego punktu przecinającego się z kołem o podanym promieniu.

Ruch po okręgu

Przeprowadzimy teraz test. Po otworzeniu pliku *sample_01.fla* zobaczysz, że listwa czasowa `_root` składa się z pojedynczej kopii klipu filmowego umieszczonej na środku sceny. Cały kod wpiszemy do tego klipu, więc dwukrotnie go kliknij, aby spowodować otwarcie do edycji.

Ten klip filmowy zawiera dwa kolejne klony klipów filmowych — kółko o nazwie kopii *Point* i krzyżyk o nazwie kopii *centerPoint*. Właściwości `_x` i `_y` klipu *centerPoint* służą jako odnośniki do współrzędnej, wokół której klip *Point* będzie się poruszać. W tym przypadku współrzędna ta odpowiada środkowi klipu filmowego (0,0).



Jeśli przyjrzysz się układowi listwy czasowej, zauważysz, że ten klip filmowy składa się tylko z trzech ujęć. Zawierają one kod ActionScript. Ujęcie 1. jest ujęciem konfiguracyjnym, w którym umieściliśmy różne funkcje. Dwukrotnie kliknij ujęcie 1., aby edytować jego zawartość, która została przedstawiona poniżej.

```
Radius = 100;
degrees = 0;

// funkcja setAngle;
function setAngle () {
  Angle = degrees * (Math.PI / 180);
  degrees = degrees + 2;
}

// funkcja drawPoints;
function drawPoints () {
  Xpos = Radius * Math.cos(Angle);
  Ypos = Radius * Math.sin(Angle);

  // rysuj
  Point._x = Xpos + centerPoint._x;
  Point._y = Ypos + centerPoint._y;
}
```

W tym ćwiczeniu zastosujemy zmienną `setAngle` do określenia wartości kąta `Angle` dla dowolnego obrotu, który będziemy chcieli obliczyć. W pierwszym przykładzie zmienna `setAngle` przypisuje zmiennej `Angle` wartości uzyskane z konwersji ze stopni na radiany, a następnie zwiększa o 2° wartości podane dla zmiennej `degrees` po każdorazowym wywołaniu polecenia `setAngle`.

Zmienna drawPoints zawiera kod obliczający punkt na kole w oparciu o omówione wcześniej równania, a następnie definiuje nową pozycję klipu *Point* na ekranie, określając jego właściwości _x i _y.

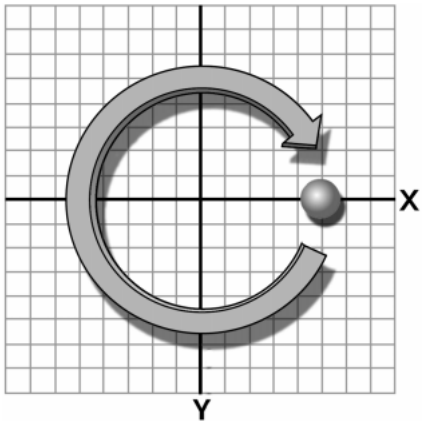
Ujęcie 2. zawiera kod, który wywołuje akcję setAngle i drawPoints. Został on przedstawiony poniżej.

```
setAngle();
drawPoints();
```

Skrypt w ujęciu 3. tworzy pętlę kodu w ujęciu 2.

```
gotoAndPlay(2);
```

Ponieważ wartość zmiennej Angle jest zwiększana przez ciągłe wywołania polecenia setAngle, zarówno sinus, jak i cosinus zostaną użyte w poleceniu drawPoints do obliczenia nowej pozycji klipu *Point* na okręgu o promieniu 100. W efekcie otrzymamy klip *Point* poruszający się po kolistej ścieżce wokół klipu *centerPoint*, jak widać poniżej.



Za pomocą zmiany wartości promienia (Radius) możemy kontrolować rozmiar kółka. Jeśli — zamiast podać określoną wartość 100 — zmodyfikujemy funkcję drawPoints tak, aby zwiększyć wartość promienia Radius po każdorazowym jej wywołaniu, kolista ścieżka będzie stopniowo się zwiększać (*sample_02.fla*).

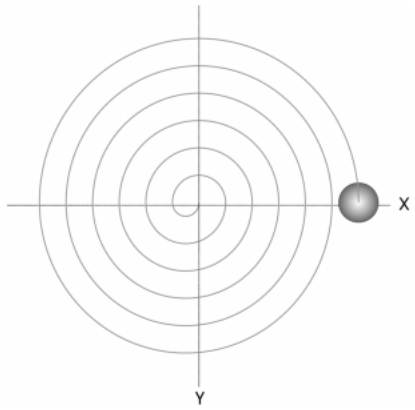
```
Radius = 0;
degrees = 0;

// funkcja setAngle
function setAngle () {
   Angle = degrees * (Math.PI / 180);
   degrees = degrees + 2;
}

// funkcja drawPoints
function drawPoints () {
   Radius = Radius + 0.5;
   Xpos = Radius * Math.cos (Angle);
   Ypos = Radius * Math.sin (Angle);
```

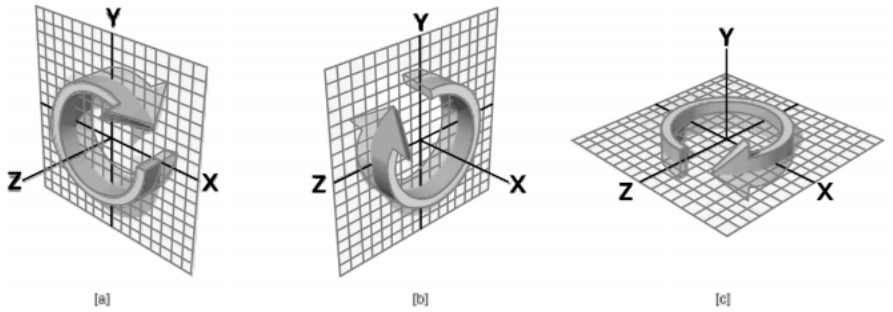
```
    // rysuj
    Point._x = Xpos + centerPoint._x;
    Point._y = Ypos + centerPoint._y;
}
```

W efekcie uzyskamy klip *Point* poruszający się po spiralnej ścieżce wokół klipu *centerPoint*.



Oś Z

W jaki sposób ta spirala związana jest z naszym ćwiczeniem 3D? Następny rysunek przedstawia analizę trzech osi w środowisku 3D (*x*, *y* i *z*). Jeśli przyjrzysz się pierwszej ilustracji, zrozumiesz, że w dwóch omówionych do tej pory przykładach obliczaliśmy obrót wokół osi *z* przy użyciu współrzędnej punktów znajdujących się na osi *x* i *y* w celu zdefiniowania kolistej ścieżki. Na ekranie komputera można wyobrazić sobie oś *z* jako oś skierowaną w naszym kierunku (jakby wychodziła z ekranu).



W oparciu o ten rysunek, możemy stwierdzić, że w środowisku 3D:

- definiując współrzędne *x* i *y* punktu, z zastosowaniem sinusa i cosinusa możemy obliczyć obrót punktu wokół osi *z*,
- definiując współrzędne *y* i *z* punktu, z zastosowaniem sinusa i cosinusa możemy obliczyć obrót punktu wokół osi *x*,

- definiując współrzędne *z* i *x* punktu, z zastosowaniem sinusa i cosinusa możemy obliczyć obrót punktu wokół osi *y*.

We Flashu możemy określić współrzędne *x* i *y* klonu przez zastosowanie właściwości `_x` i `_y`, lecz uzyskanie wartości współrzędnej *z* jest odrobinę trudniejsze. W środowisku trójwymiarowym pozycja obiektu na osi *z* zazwyczaj reprezentuje jego odległość od widza. Z powodu perspektywy im bliżej obiekt znajduje się przy widzu, tym wygląda na większy (i na odwrót). Oznacza to, że do przedstawienia współrzędnej klonu na osi *z* możemy zastosować jego właściwości `_xscale` i `_yscale`.

Obrót wokół osi Y

Przeprowadzimy teraz test. W oparciu o nasz pierwszy eksperyment (*sample_01.fla*) zmodyfikowałem kod dla funkcji `drawPoints` w celu symulowania obrotu wokół osi *y*. Zgodnie z wyżej wymienionymi regułami dokonujemy tego, określając współrzędne *x* i *z*. Nowy kod, wpisany do pliku *sample_03.fla*, został przedstawiony poniżej.

```
Radius = 100;
degrees = 0;

// funkcja setAngle
function setAngle () {
   Angle = degrees * (Math.PI / 180);
   degrees = degrees + 2;
}

// funkcja drawPoints
function drawPoints () {
   Xpos = Radius * Math.cos (Angle);
   Zpos = Radius * Math.sin (Angle);

   // rysuj
   Point._x = Xpos + centerPoint._x;
   Point._yscale = Point._xscale = Zpos + 200;
}
```

Wprowadziliśmy tutaj tylko dwie zmiany. Najpierw poniższy wiersz…

```
Ypos = Radius * Math.sin (Angle);
```

…zastąpiliśmy linijką przedstawioną poniżej.

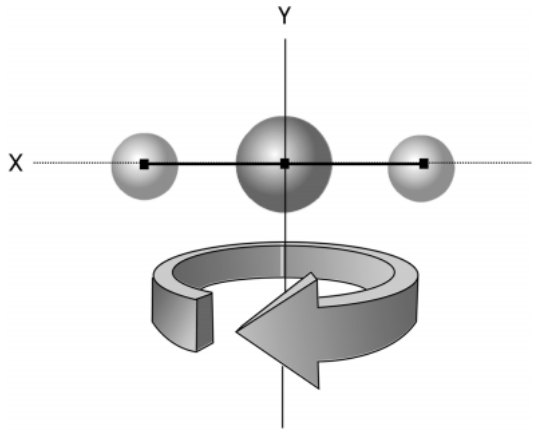
```
Zpos = Radius * Math.sin (Angle);
```

Zamiast definiować właściwości `_y` klipu *Point*, porównaliśmy jego właściwości `_xscale` i `_yscale` do zmiennej `Zpos + 200` (jak widać poniżej).

```
Point._yscale = Point._xscale = Zpos + 200;
```

W tym kodzie definiujemy właściwości `Point.yscale` i `Point.xscale` jako równe `Zpos + 200`. Wartość 200 zostaje następnie dodana do zmiennej `Zpos`, aby zrekompensować to, że z promieniem `Radius` o wartości 100 minimalna wartość zmiennej `Zpos` będzie wynosić –100 (przecież nie chcemy, aby klip *Point* posiadał ujemną wartość skali). Ruch klipu *Point* wzdłuż osi *x* pozostaje taki sam (jak został zdefiniowany przez cosinus kąta `Angle`),

lecz — zamiast poruszać się w dół i w górę osi *y* — klip *Point* przemieszcza się teraz „do przodu" i „do tyłu" po osi *z*. Czyli — mówiąc dokładniej — skaluje się tak, jak zostało to zdefiniowane przez sinus kąta `Angle`. W efekcie powstaje iluzja klipu *Point* poruszającego się po kolistej ścieżce w przestrzeni, co widać poniżej.



Prawdziwa perspektywa

Chociaż ten ruch wygląda dosyć realistycznie, nie jest jeszcze całkowicie poprawny. Dopasowaliśmy skalę klipu *Point*, ale nie wyregulowaliśmy ścieżki, którą podąża nasz „obiekt". Jako że ścieżka rozciąga się w naszą stronę, oddalając się od nas, powinna być widoczna w perspektywie. Aby uzyskać taki efekt, wprowadzimy zmienną `perspective` i zdefiniujemy właściwości `_x`, `_xscale` i `_yscale` klipu *Point* w stosunku do tej zmiennej.

Na początku fragmentu skryptu umieszczonego w ujęciu 1. pliku *sample_04.fla* nowej zmiennej `perspective` przypisaliśmy wartość 150, co widać poniżej.

```
Radius = 100;
perspective = 150;
degrees = 0;

// funkcja setAngle
function setAngle () {
   Angle = degrees * (Math.PI / 180);
   degrees = degrees + 2;
}
```

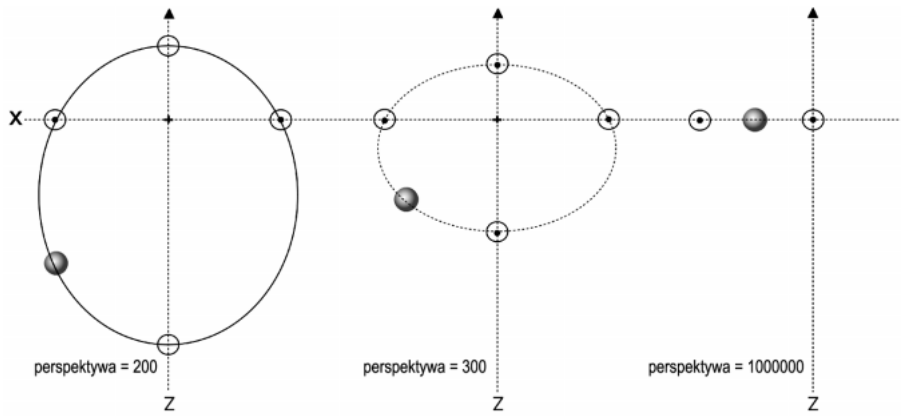
Zmienna wykorzystywana jest przez polecenie `drawPoints` do obliczania stopnia zachodzącego zniekształcenia perspektywy. Im mniejsza jest wartość zmiennej `perspective`, tym bardziej wyraziste będzie zniekształcenie (i na odwrót). W samym poleceniu `drawPoints` zdefiniowaliśmy nową zmienną `Depth`, opartą na wartościach zmiennych `perspective` i `Zpos`.

```
// funkcja drawPoints
function drawPoints () {
   Xpos = Radius * Math.cos (Angle);
   Zpos = Radius * Math.sin (Angle);
```

```
    // perspektywa
    Depth = 1 / (1 - (Zpos / perspective));

    // rysuj
    Point._x = (Xpos * Depth) + centerPoint._x;
    Point._y = centerPoint._y;
    Point._yscale = Point._xscale = Depth * 100;
}
```

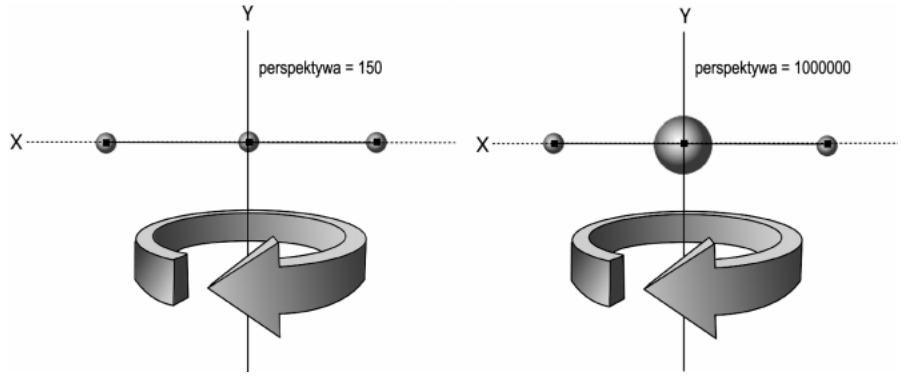
Wartość zmiennej `Depth` zostaje pomnożona przez wartość zmiennej `Xpos` w celu zdefiniowania właściwości `_x`, a następnie pomnożona przez 100, aby określić wartości właściwości `_xscale` i `_yscale` klipu filmowego *Point*. Im „bliżej" klip filmowy *Point* znajduje się widza (czyli im większa jest wartość zmiennej `Zpos`), tym większa będzie wartość zmiennej `Depth`. Poniższy rysunek (który jest widokiem wzdłuż osi *y*, czyli z „góry") przedstawia zniekształcenie ścieżki wokół środka i wpływ na nie różnych wartości zmiennej `perspective`.



Podczas oglądania rysunku wzdłuż osi *z* widoczny będzie efekt bardziej wyrazistego ruchu i skalowania w czasie „przybliżania" się obiektu do widza, a delikatniejszego ruchu, kiedy obiekt się oddala. Można to porównać do jazdy samochodem — najbliższe otoczenie wokół samochodu wygląda tak, jakby się poruszało o wiele szybciej niż, na przykład, góry w tle. Gdy wartość zmiennej `perspective` dochodzi do 1 000 000, nie nastąpi prawie żadne zniekształcenie. Powstanie wtedy **widok ortogonalny** (czyli taki, w którym nie jest obecne zniekształcenie perspektywiczne). Jeśli spojrzymy na nasze otoczenie z perspektywy ortogonalnej, wówczas będzie nas dzieliła od wszystkich obiektów jednakowa odległość.

Na razie zmiennej `perspective` w ujęciu 1. klipu filmowego *Move3D* przypisaliśmy wartość `1000000`, więc podczas oglądania filmu nie zauważysz znacznego zniekształcenia.

Spróbuj zmienić wartość na 150, a następnie uruchomić film, aby zobaczyć naprawdę wyrazisty efekt perspektywy. Można nawet utworzyć efekt przesuwania się obiektu poza ekran. W tym celu przypisz zmiennej `perspective` wartość 15 i spróbuj powstrzymać się od uchylania, gdy obiekt będzie zataczać pętlę nad Twoją głową. Po zmianie wartości zmiennej `perspective` modyfikacje ruchu i skalowania klipu *Point* są łatwo zauważalne.



Obrót wokół osi X

Tę samą technikę możemy zastosować do symulacji obrotu wokół osi *x*, obliczając współrzędne *z* i *y*. Widok wzdłuż osi *x* przedstawia płaszczyznę, na której *y* reprezentuje współrzędną pionową (sinus), a *z* współrzędną poziomą (cosinus).

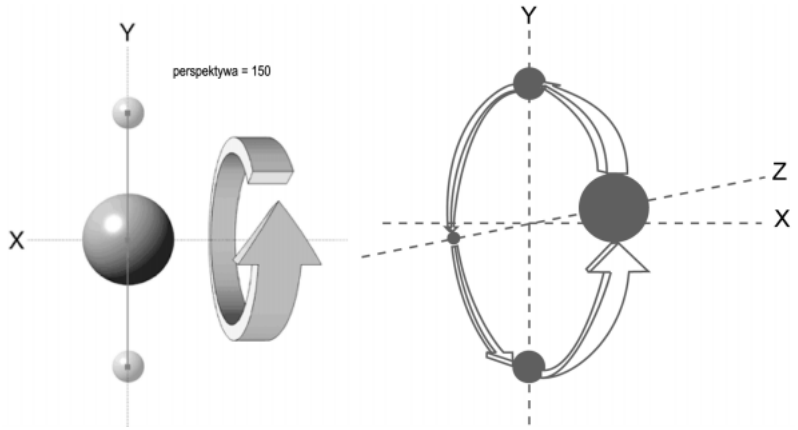
W skrypcie symulacji obrotu wokół osi *x* (*sample_05.fla*) polecenie `drawPoints` będzie wyglądać tak jak poniżej.

```
function drawPoints () {
   Ypos = Radius * Math.cos (Angle);
   Zpos = Radius * Math.sin (Angle);

   // perspektywa
   Depth = 1 / (1 - (Zpos / perspective));

   // rysuj
   Point._x = centerPoint._x;
   Point._y = (Ypos * Depth) + centerPoint._y;
   Point._yscale = Point._xscale = Depth * 100;
}
```

Rezultat wprowadzonych zmian został przedstawiony na poniższym rysunku.



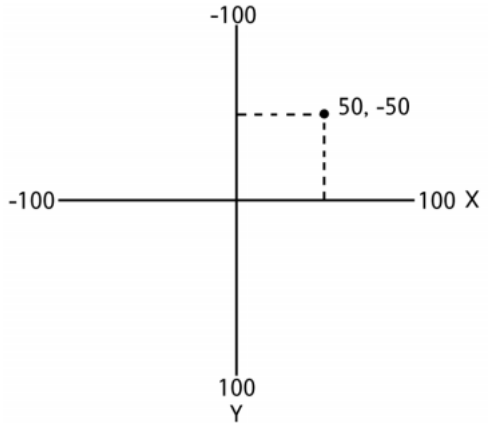
Nasza kula sprawia teraz takie wrażenie, jakby obracała się wokół osi *x*, przybliżając się i oddalając podczas obrotu.

Przemieszczanie kilku punktów

Skoro już wiesz, w jaki sposób stosujemy funkcje sinus i cosinus do symulacji obrotu pojedynczego punktu wokół osi w przestrzeni, przyjrzyjmy się teraz metodom tworzenia systemu, w którym wokół środka będzie się obracać kilka punktów.

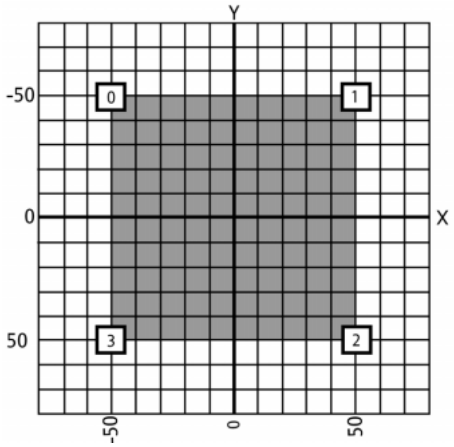
Przede wszystkim musimy określić rozmieszczenie punktów w systemie i sposób przechowywania informacji o nich. Aby maksymalnie uprościć wyjaśnienia, omówię to zagadnienie najpierw dla systemu dwuwymiarowego, a dopiero później dodam trzeci wymiar.

W systemie dwuwymiarowym możemy określić współrzędną *x* i *y* dla dowolnego punktu. Współrzędne te zdefiniujemy w stosunku do punktu środkowego (0,0) i będą one działać na tej samej zasadzie, co jednopunktowe systemy, którymi zajmowaliśmy się do tej pory. Oznacza to, że będziemy postępować zgodnie z konwencją Flasha. Według wspomnianej reguły dowolny punkt umieszczony *nad* osią *x* będzie posiadał *ujemną* wartość współrzędnej *y*, a punkt umieszczony *poniżej* będzie posiadał *dodatnią* wartość współrzędnej *y*. Tę samą zasadę zastosujemy dla współrzędnych *x* (punkt znajdujący się na lewo od osi *y* będzie miał ujemną wartość współrzędnej i na odwrót).



Obrót wokół osi Z

Zaczniemy od systemu składającego się z czterech punktów. Poniższy rysunek przedstawia współrzędne niezbędne do narysowania kwadratu o rozmiarze 100×100 jednostek (ze środkiem wyrównanym do środka siatki).



Współrzędne dla rogów tego kwadratu zostały przedstawione poniżej.

Punkt 0:	x = –50,	y = –50
Punkt 1:	x = 50,	y = –50
Punkt 2:	x = –50,	y = 50
Punkt 3:	x = 50,	y = 50

Do przechowywania tych wartości zastosujemy dwie tablice o nazwach `x` i `y`. W oparciu o pierwsze ujęcie pliku *sample_01.fla* (zajmowaliśmy się nim w pierwszym ćwiczeniu) wiersz definiujący wartość promienia `Radius` zastąpimy kodem, który definiuje dwie nowe tablice. Nowy skrypt (umieszczony w pliku *sample_06.fla*) został przedstawiony poniżej.

```
// określamy punkt na współrzędnych xyz
x = new Array(-50, 50, -50, 50);
y = new Array(-50, -50, 50, 50);
verticeNum = x.length;
```

W tym kodzie zmiennej `verticeNum` przypisujemy wartość zmiennej `length` tablicy `x` (jest to liczba punktów w systemie). Ponieważ chcemy, aby nasz system składał się z czterech punktów, dołączyłem procedurę, która tworzy kopie klipu *Point* i nadaje im nazwy *Point0*, *Point1*, *Point2* i *Point3*, a następnie czyni oryginalny klip filmowy niewidzialnym.

```
// powielamy punkty
for (c = 0; c < verticeNum; c++) {
   duplicateMovieClip ("Point", "Point" + c, c);
}

Point._visible = 0;
```

Główna zmiana w poleceniu `drawPoints` zachodzi w równaniach, które dotychczas stosowaliśmy do obliczania zmiennych `Xpos` i `Ypos`.

```
Xpos = promień * cos (kąt)
Ypos = promień * sin (kąt)
```

Zamiast promienia `Radius` zastosujemy współrzędne *x* i *y* (które właśnie zdefiniowaliśmy do podobnego działania). Nowe równania zostały przedstawione poniżej.

```
Xpos = x * sin (kąt) + y * cos (kąt)
Ypos = y * sin (kąt) - x * cos (kąt)
```

Zmienne `Xpos` i `Ypos` (jak również właściwości `_x` i `_y`) musimy jednak określić dla wszystkich punktów w systemie. Dokonamy tego, stosując pętlę wykonującą akcje, które czterokrotnie zdefiniują te wartości. Zmienna `c` służy do kontrolowania ilości wykonanych pętli i do udostępniania wartości tablic `x` i `y`.

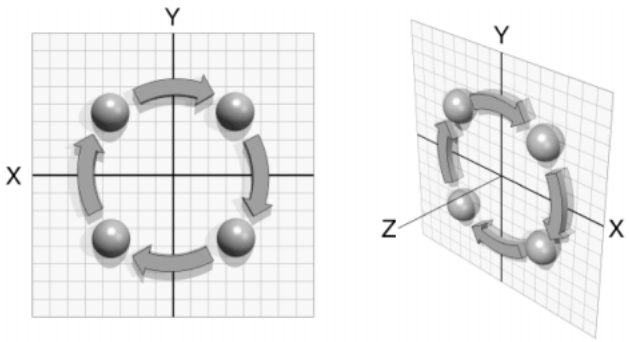
```
function drawPoints () {
   for (c = 0; c < verticeNum; c++) {

      // obrót Z
      Xpos = x[c] * Math.sin (Angle) + y[c] * Math.cos(Angle);
      Ypos = y[c] * Math.sin (Angle) - x[c] * Math.cos(Angle);

      // rysuj
      this["Point" + c]._x = Xpos + centerPoint._x;
      this["Point" + c]._y = Ypos + centerPoint._y;
  }
}
```

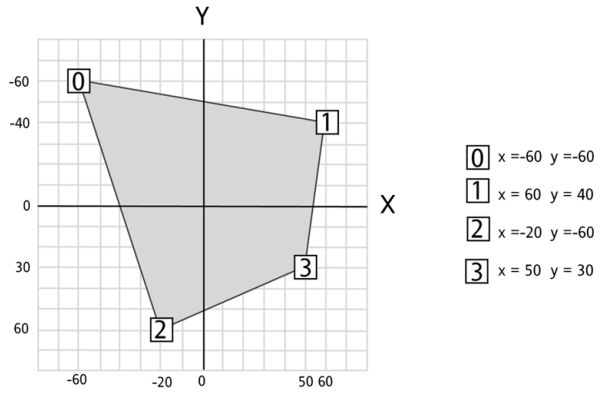
Po uruchomieniu filmu zmienna `this["Point" + c]` umieszczona w sekcji `Rysuj` powyższego kodu zostanie zmieniona na nazwy kopii klipu filmowego `Point0`, `Point1`, `Point2` i `Point3`.

Uruchamiając nowy skrypt (*sample_06.fla*), zauważysz, że po zwiększeniu kąta (za pomocą zmiennej `setAngle`) wszystkie cztery punkty poruszają się po kolistej ścieżce wokół klipu *centerPoint*, zachowując stałą odległość między sobą.



Prawdziwą zaletą stosowania tej metody jest możliwość zmiany współrzędnych *x* i *y* każdego punktu w systemie. Przeprowadzimy krótki eksperyment, który lepiej zilustruje to zagadnienie — zmienimy wartości w tablicach `x` i `y` (kształt przedstawiony na poniższym rysunku potraktujemy jako wzór). Kształt ten prezentuje nowy zestaw wartości.

Punkt0:	x = –60,	y = –60
Punkt1:	x = 60,	y = –40
Punkt2:	x = –20,	y = 60
Punkt3:	x = 50,	y = 30

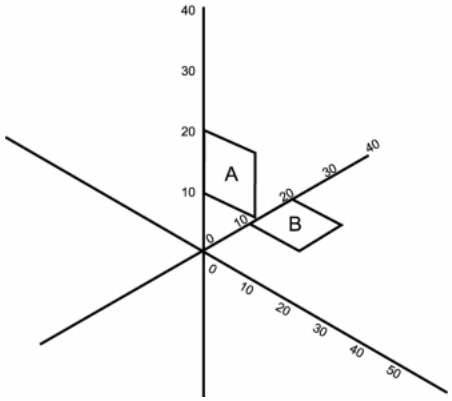


Po zastąpieniu wartości zdefiniowanych dla tablic `x` i `y` nowymi (*sample_07.fla*) uzyskamy nową konfigurację punktów w systemie przypominającą kształtem poprzednią figurę. Punkty nadal będą się jednak obracać wokół klipu *centerPoint* z zachowaniem stałych odległości między sobą. Poeksperymentuj ze zmianą tych wartości i obejrzyj rezultaty.

Obrót wokół osi Y

Podobnie jak w pierwszym przykładzie tego rozdziału, dokonaliśmy obliczeń, które reprezentują obrót wokół osi *z*. Na podstawie informacji zdobytych podczas ćwiczeń na plikach *sample_04.fla* i *sample_05.fla* możemy wykonać również obliczenia dla osi *x* i *y*. Zaczniemy od obrotu wokół osi *y*.

Aby symulować obrót wokół osi *y*, musimy określić wartości zmiennych `Xpos` i `Zpos`. W tym celu będziemy potrzebowali zestawu współrzędnych punktów znajdujących się na osiach *x* i *z*. Wyobraźmy sobie „obrócenie" płaszczyzny *x* — *y* o 90° wokół osi *x* w taki sposób, że stare współrzędne *y* stają się współrzędnymi *z*, a współrzędne *x* pozostają niezmienione.



A		B	
x	y	x	y
0,	20	0	20
10,	20	10	20
10,	10	10	10
0,	10	0	10

Nowe równanie dla obliczenia obrotu wokół osi *y* zostało przedstawione poniżej.

```
Xpos = x * sin(kąt) + z * cos(kąt)
Zpos = z * sin(kąt) - x * cos(kąt)
```

W tym ćwiczeniu wykorzystamy również kod, który zdefiniuje wartość zmiennej `Depth` stosowanej do określania właściwości `_xscale` i `_yscale` klipu *Point*. Wartość zmiennej `Depth` zostanie również pomnożona przez wartość zmiennej `Xpos` (dając w ten sposób właściwość `_x` klipu *Point*).

Ponieważ zmienna `Depth` zależna jest od wartości zmiennej `perspective`, musimy pamiętać o przypisaniu zmiennej `perspective` jakiejś wartości. Tak jak poprzednio umieszczamy ją na początku skryptu w ujęciu 1. Kod symulujący obrót wokół osi *y* dla polecenia *drawPoints* został przedstawiony poniżej (znajdziesz go również w pliku *sample_08.fla* na płycie CD-ROM dołączonej do książki).

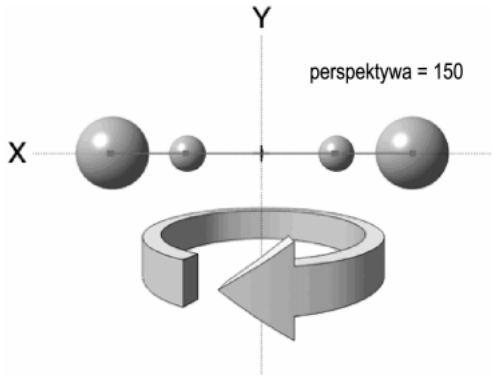
```
// funkcja drawPoints
function drawPoints () {
   for (c = 0; c < verticeNum; c++) {
      // obrót Y
      Xpos = x[c] * Math.sin (Angle) + z[c] * Math.cos (Angle);
      Zpos = z[c] * Math.sin (Angle) - x[c] * Math.cos (Angle);

      // perspektywa
      Depth = 1 / (1 - (Zpos / perspective));

      // rysuj
      this["Point" + c]._x = (Xpos * Depth) + centerPoint._x;
      this["Point" + c]._y = centerPoint._y;
      this["Point" + c]._xscale = this["Point" + c]._yscale = Depth * 100;

      // sortowanie Z
      this["Point" + c].swapDepths(Depth * 500);
   }
}
```

Ostatni wiersz kodu tej funkcji jest odpowiedzialny za **sortowanie z** (czyli określanie punktów, które należy narysować przed innymi podczas ich obrotu). We Flashu 4 wykonanie tej operacji było dosyć pracochłonne, lecz we Flashu 5 możemy w tym celu zastosować wbudowaną funkcję `swapDepths`. Funkcja ta przesuwa bieżącą kopię na nowy poziom określony wyrażeniem `Depth * 500`. Im większa jest zatem wartość zmiennej `Depth`, tym na wyższy poziom w bieżącym stosie zostanie przesunięty klon klipu *Point*. Funkcja ta nie *zastępuje* zawartości poziomu, do którego przechodzi, lecz *zamienia* zawartości dwóch poziomów. Jest to szczególnie pomocne wtedy, gdy dwa punkty posiadają jednakową wartość zmiennej `Depth`.



Obrót wokół osi X

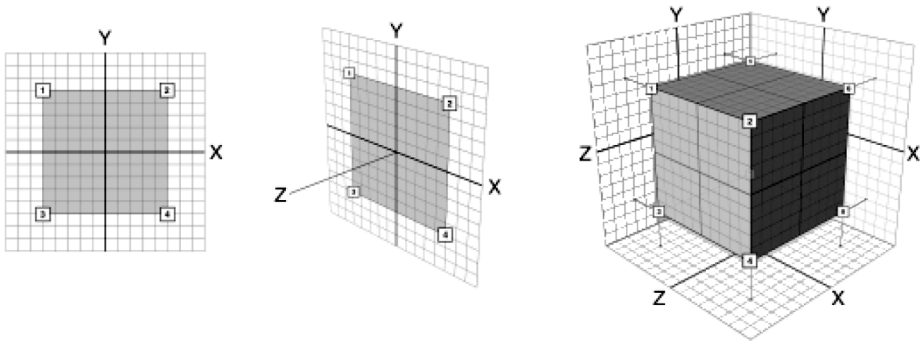
Równania dla obrotu wokół osi *x*, które służą do obliczania współrzędnych *y* i *z*, zostały przedstawione poniżej (patrz plik *sample_09.fla*).

```
Ypos = y * sin(kąt) + z * cos(kąt)
Zpos = z * sin(kąt) - y * cos(kąt)
```

Rezultat będzie zbliżony do efektu z pliku *sample_05.fla* z tym wyjątkiem, że teraz już cztery punkty obracają się wokół osi *x*.

Obiekt trójwymiarowy

Do tego momentu obliczaliśmy obrót wokół poszczególnych osi oddzielnie, za każdym razem przypisując do obliczenia po dwie współrzędne na punkt. Aby jednak powstała iluzja obracania obiektu trójwymiarowego, każdemu punktowi musimy przypisać współrzędną *x*, *y* i *z*, a obliczenia dla wszystkich trzech osi należy wykonać dla każdego punktu. Poniższy rysunek ilustruje sytuację, w której — do wytłoczenia dwuwymiarowego kwadratu w celu wygenerowania sześcianu — musimy zastosować cztery dodatkowe punkty i do każdego z nich przypisać współrzędną *z*.



Końcowy sześcian unosi się w trójwymiarowej przestrzeni współrzędnych o punkcie początkowym (0,0,0). W oparciu o ten rysunek możemy stwierdzić, że trójwymiarowe współrzędne dla sześcianu (takiego jak ten, czyli o rozmiarze 100 na 100 na 100 jednostek z punktem środkowym w pozycji (0,0,0)) będą wyglądać tak jak poniżej.

Punkt0:	x = –50,	y = –50,	z = 50
Punkt1:	x = 50,	y = –50,	z = 50
Punkt2:	x = –50,	y = 50,	z = 50
Punkt3:	x = 50,	y = 50,	z = 50
Punkt4:	x = –50,	y = –50,	z = –50
Punkt5:	x = 50,	y = –50,	z = –50
Punkt6:	x = –50,	y = 50,	z = –50
Punkt7:	x = 50,	y = 50,	z = –50

Zamiast wykonywać obliczenia dla obrotu wokół pojedynczej osi (tak jak to robiliśmy do tej pory z zastosowaniem funkcji `drawPoints`), dokonamy obliczeń dla wszystkich

trzech osi jednocześnie. Z tego względu musimy wprowadzić trzy nowe zmienne, które będą kontrolowały kąt obrotu wokół każdej osi. Nazwiemy je `Xangle`, `Yangle` i `Zangle` i zastąpimy nimi ogólną zmienną `Angle` (którą stosowaliśmy do tej pory).

Zmienna `Xangle` będzie reprezentować kąt obrotu wokół osi *x*, zmienna `Yangle` — wokół osi *y*, a zmienna `Zangle` — wokół osi *z*. Nowy zestaw równań można zatem podzielić na trzy części. Wszystkie równania zostały przedstawione poniżej.

Obrót wokół osi *x*:

```
Ypos_Temp = y * sin(Xangle) + z * cos(Xangle)
Zpos_Temp = z * sin(Xangle) - y * cos(Xangle)
```

Obrót wokół osi *y*:

```
Xpos_Temp = x * sin(Yangle) + Zpos_Temp * cos(Yangle)
Zpos = Zpos_Temp * sin(Yangle) - x * cos(Yangle)
```

Obrót wokół osi *z*:

```
Xpos = Xpos_Temp * sin(Zangle) + Ypos_Temp * cos(Zangle)
Ypos = Ypos_Temp * sin(Zangle) - Xpos_Temp * cos(Zangle)
```

W tych równaniach zmienne `Xpos_Temp`, `Ypos_Temp` i `Zpos_Temp` służą jako tymczasowe zmienne stosowane w miejsce zmiennych `x`, `y` i `z` (w celu uzyskania dostępu do wartości zmiennych `Xpos`, `Ypos` i `Zpos`).

Po przypisaniu wartości zmiennym `Xpos`, `Ypos` i `Zpos` do powyższych równań dodamy kod definiujący zmienną `Depth`. Nowe pozycje punktów zostaną narysowane na ekranie za pomocą pomnożenia wartości zmiennej `Depth` przez wartości zmiennych `Xpos` i `Ypos` oraz przy użyciu określenia właściwości `_x` i `_y` każdego punktu w taki sam sposób jak w poprzednim przykładzie. W tym przypadku również zastosowaliśmy zmienną `Depth` do określenia właściwości `_xscale` i `_yscale` dla każdego klonu *Point*.

Skrypt tworzący funkcję `drawPoints` został przedstawiony poniżej (oraz w pliku *sample_10.fla*).

```
// funkcja drawPoints
function drawPoints () {
   for (c = 0; c < verticeNum; c++) {
      // obrót X
      Ypos_Temp = y[c] * Math.sin (Xangle) + z[c] * Math.cos (Xangle);
      Zpos_Temp = z[c] * Math.sin (Xangle) - y[c] * Math.cos (Xangle);

      // obrót Y
      Xpos_Temp = x[c] * Math.sin (Yangle) + Zpos_Temp * Math.cos (Yangle);
      Zpos = Zpos_Temp * Math.sin (Yangle) - x[c] * Math.cos (Yangle);

      // obrót Z
      Xpos = Xpos_Temp * Math.sin (Zangle) + YPos_Temp * Math.cos (Zangle);
      Ypos = Ypos_Temp * Math.sin (Zangle) - Xpos_Temp * Math.cos (Zangle);

      // perspektywa
      Depth = 1 / (1 - (Zpos / perspective));
```

```
        // rysuj
        this["Point" + c]._x = (Xpos * Depth) + centerPoint._x;
        this["Point" + c]._y = (Ypos * Depth) + centerPoint._y;
        this["Point" + c]._xscale = this["Point" + c]._yscale = Depth * 100;

        // sortowanie Z
        this["Point" + c].swapDepths(Depth * 500);
    }
}
```

Pamiętaj, że w celu utworzenia efektów trójwymiarowych nie zawsze konieczne jest obliczanie obrotu wokół wszystkich trzech osi. Czasami niezbędne jest obrócenie obiektu wokół osi *z*, lecz też nie zawsze jest to konieczne. Ogólnie mówiąc — czasami warto ograniczać kod stosowany we Flashu. Wystarczy, abyśmy obliczyli tylko obroty wokół osi *x* i *y*.

Obrót wokół osi *x*:

```
Ypos = y * sin(kątX) + z * cos(kątX)
Zpos = z * sin(kątX) - y * cos(kątX)
```

Obrót wokół osi *y*:

```
Xpos = x * sin(kątY) + Zpos * cos(kątY)
Zpos = Zpos * sin(kątY) - x * cos(kątY)
```

Dzięki temu nie będą już nam potrzebne dodatkowe zmienne tymczasowe (patrz plik *sample_11.fla*). Dla potrzeb tego ćwiczenia nadal jednak będziemy korzystać z równań obliczających obrót wokół wszystkich trzech osi.

Aby przejść z pliku *sample_09.fla* do *sample_10.fla*, musimy zmienić funkcję `setAngle` w celu zdefiniowania wartości zmiennych wszystkich trzech kątów (`Xangle`, `Yangle` i `Zangle`). Dokonamy tego, trzykrotnie powielając istniejący skrypt (zmieniając tylko nazwy zmiennych).

```
function setAngle () {
  Xangle = Xdegrees * (Math.PI / 180);
  Yangle = Ydegrees * (Math.PI / 180);
  Zangle = Zdegrees * (Math.PI / 180);
  Xdegrees = Xdegrees + 2;
  Ydegrees = Ydegrees + 2;
  Zdegrees = Zdegrees + 2;
}
```

Wartości dodawane do zmiennej `degrees` na końcu kodu określają to, o ile każda oś ma się obrócić. Na przykład wartość 0 zatrzyma sześcian w miejscu, a wartość ujemna spowoduje obracanie się obiektu w przeciwnym kierunku. Oczywiście wartości te można definiować, stosując inne wyrażenia. Aby na przykład obrócić obiekt w zależności od ruchu myszy, możemy zastosować poniższy skrypt (umieszczony również w pliku *sample_12.fla*).

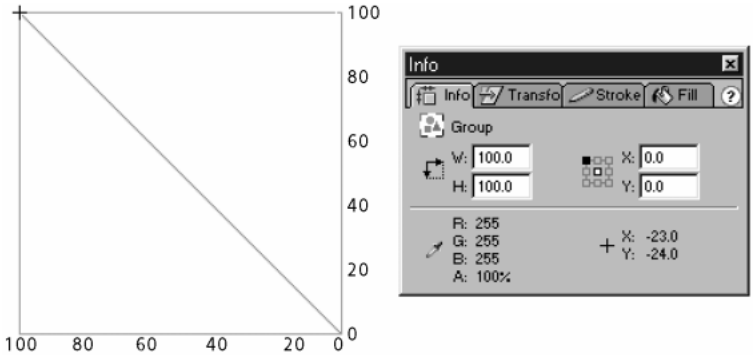
```
function setAngle () {
   Xangle = Xdegrees * (Math.PI / 180);
   Yangle = Ydegrees * (Math.PI / 180);
   Zangle = Zdegrees * (Math.PI / 180);
   Xdegrees = this._xmouse;
   Ydegrees = this._ymouse;
   Zdegrees = 0;
}
```

Ustawienie stopni `Xdegrees` i `Ydegrees` na bieżącą lokalizację kursora myszy jest najprostszym sposobem uzyskania omawianego efektu, lecz możemy także zastosować bardziej skomplikowane skrypty, jeśli chcemy otrzymać inny rodzaj ruchu czy interakcji.

W ten sposób dotarliśmy do ostatniego etapu projektu. Na razie utworzyliśmy w systemie 8 punktów poruszających się wokół punktu środkowego. Aby nadać obiektowi określony kształt, punkty połączymy liniami. W tym celu utworzymy kolejną funkcję o nazwie `drawLines`, która będzie kreować pętlę dla umieszczanych linii. Do połączenia punktów w celu utworzenia sześcianu będziemy potrzebowali dwanaście linii. We Flashu nie możemy niestety definiować współrzędnych poszczególnych punktów linii za pomocą języka ActionScript. Musimy znaleźć na to jakiś inny sposób.

Zastosujemy klip filmowy zawierający pojedynczą ukośną linię. Zeskalujemy ją poziomo i pionowo między punktami, które mają zostać połączone. Wartości zmiennych `_width`, `_height` i `_scale` zastosowanych w filmie muszą być identyczne, aby jego punkty końcowe odpowiadały punktom przeznaczonym do połączenia. Innymi słowy — rozmiar klipu filmowego powinien wynieść 100 na 100 jednostek. Oznacza to, że jeśli znamy odległość pomiędzy dwoma punktami i chcemy nadać linii w klipie filmowym odpowiednią długość, wystarczy ustawić wartości właściwości `_scale` klipu filmowego na tę odległość.

Aby utworzyć taki klip filmowy, narysuj ukośną linię biegnącą w dół od lewej do prawej strony pod katem 45°. W panelu *Info* ustaw wartości X i Y na 0 tak, aby górny lewy koniec linii stał się „środkiem" klipu filmowego. Ustaw szerokość i wysokość na 100.



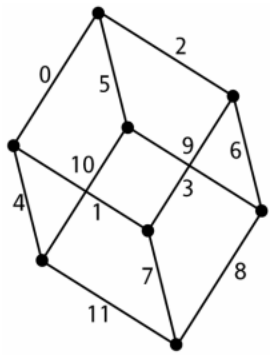
Nazwij tę kopię klipu filmowego *Line*.

W ujęciu 1. klipu filmowego *Move3D* w pliku *sample_13.fla* przypiszemy zmiennej o nazwie `LineNum` wartość 12. Reprezentuje ona liczbę linii, którymi połączymy wszystkie punkty w sześcianie.

Następnym krokiem będzie powielenie klipu *Line* dwanaście razy (nadając kopiom nazwy `Line0`, `Line1`, `Line2` i tak dalej) i ustawienie widoczności oryginalnego klonu na `0`.

```
lineNum = 12;

// powielamy linie
for (c = 0; c < lineNum; c++) {
```

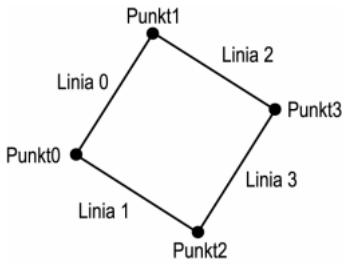



```
    duplicateMovieClip ("Line", "Line" + c, c + verticeNum);
  }
  Line._visible = 0;
```

Aby skrypt odpowiedzialny za łączenie wszystkich linii uzyskał informację, które punkty należy połączyć, przygotujemy dla każdej linii listę punktów przeznaczonych do połączenia. Dokonamy tego, przypisując do każdej z linii zmienne o nazwie `Line`*X*`Start` i `Line`*X*`End`, gdzie *X* oznacza liczbę, którą dodamy do nazwy każdego klipu filmowego w ostatnim fragmencie kodu. Kompletna lista została przedstawiona poniżej.

```
// określamy informacje o liniach
Line0Start =  0, Line0End =  1;
Line1Start =  0, Line1End =  2;
Line2Start =  1, Line2End =  3;
Line3Start =  2, Line3End =  3;
Line4Start =  0, Line4End =  4;
Line5Start =  1, Line5End =  5;
Line6Start =  2, Line6End =  6;
Line7Start =  3, Line7End =  7;
Line8Start =  7, Line8End =  6;
Line9Start =  7, Line9End =  5;
Line10Start = 5, Line10End = 4;
Line11Start = 6, Line11End = 4;
```

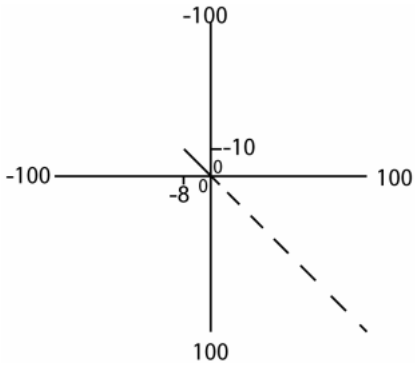
Dla linii `Line0` punktem zakotwiczenia będzie zatem punkt `Point0` (`Line0Start`), a zostanie ona rozciągnięta do punktu `Point1` (`Line0End`). Linia `Line1` zostanie połączona od punktu `Point0` do `Point2` i tak dalej. Pierwsze cztery linie wyglądać będą tak jak na poniższym rysunku.



Następnie dodamy funkcję — `DrawLines` — która połączy punkty. Poniższy kod wpisz w ujęciu 1.

```
// funkcja drawLines;
function drawLines () {
   for (c = 0; c < lineNum; c++) {
      this["Line" + c]._x = this["Point" + (this["Line" + c + "Start"])]._x;
      this["Line" + c]._y = this["Point" + (this["Line" + c + "Start"])]._y;
      this["Line" + c]._xscale = this["Point" + (this["Line" + c + "End"])]._x -
      ↳this["Point" + (this["Line" + c + "Start"])]._x;
      this["Line" + c]._yscale = this["Point" + (this["Line" + c + "End"])]._y -
      ↳this["Point" + (this["Line" + c + "Start"])]._y;
   }
}
```

W tym skrypcie wartości właściwości `_x` i `_y` każdej linii są równe wartościom właściwości `_x` i `_y` punktu reprezentującego punkt początkowy linii (`LineXStart`). Powyższy kod skaluje linię, stosując różnicę między wartościami właściwości `_x` i `_y` dwóch punktów przez nią połączonych. Cały mechanizm działa poprawnie. Jeśli zeskalujemy jakiś element za pomocą ujemnych wartości, odbijemy go przez jego oś. Zatem w celu uzyskania klipu *Line* przedstawionego na poniższym rysunku…

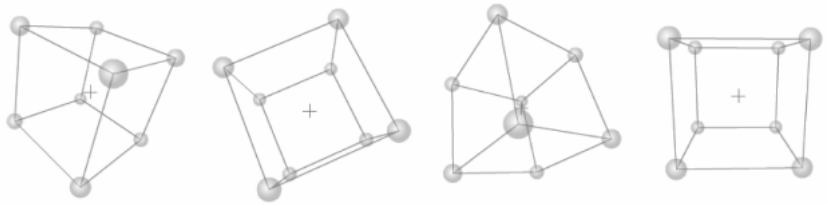


…zastosujemy poniższe równania.

```
StartX = 0, EndX = -8
StartY = 0, EndY = -10
_xscale = -8 - 0 = -8
_yscale = -10 - 0 = -10
```

Za pomocą powyższych równań skalujemy klip *Line* przez –8% na osi *x* i określamy szerokość klipu filmowego na –8. Następnie skalujemy klip na osi *y* przez –10% i określamy wysokość klipu filmowego na –10.

Po zastosowaniu wartości dla wszystkich linii uzyskaliśmy zamierzony przez nas efekt, czyli sześcian w przestrzeni trójwymiarowej (który został przedstawiony poniżej).



W ten sposób zakończyliśmy konstrukcję podstawowego programu do tworzenia obiektów 3D, którą możesz zastosować w innych projektach. Ukończony model znajduje się w pliku *sample_13.fla* na płycie CD-ROM dołączonej do książki.

Zakończenie

Teraz poeksperymentuj, pozmieniaj współrzędne poszczególnych punktów i przekonaj się, w jaki sposób zmiany te wpłyną na model. Spróbuj również umieścić dodatkowe punkty w systemie, zmieniając wartość zmiennej `verticeNum` i określić dla nich początkowe współrzędne. Stosunkowo łatwym zadaniem będzie przekształcenie tego modelu w system dynamiczny, w którym użytkownik będzie mógł kontrolować głębię lub liczbę punktów za pomocą wejściowego pola tekstowego (lub przyciskami plus i minus) i obserwować zmiany zachodzące w czasie rzeczywistym.

Pamiętaj, że współrzędne nie muszą być statyczne. Największą zaletą tego systemu jest to, że można w nim resetować pozycje punktów po każdej wykonanej pętli systemu (mniej więcej w taki sam sposób, w jaki określaliśmy zmienne odpowiedzialne za definiowanie kątów obrotu). Daje to możliwość przesuwania punktów w obrębie systemu, a to z kolei stanowi podstawę, na której zostały zrealizowane projekty omówione na początku tego rozdziału.