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

Java. Tworzenie gier



Autorzy: David Brackeen, Bret Barker, Laurence Vanhelsuwe
Tłumaczenie: Sławomir Dzieniszewski (rozdz. 7, 8, 12), Paweł Gonera (rozdz. 1 – 6, 14 – 16), Mikołaj Szczepaniak (rozdz. 9 – 11, 13, 17 – 19)
ISBN: 83-7361-411-7
Tytuł oryginału: Developing Games in Java
Format: B5, stron: 756
Przykłady na ftp: 4924 kB

Wykorzystaj do tworzenia gier najpopularniejszy język programowania ery internetu

Java jest nowoczesnym i prostym językiem programowania zorientowanym obiektowo. Trudno nie doceniać jej zalet – czytelna i zrozumiała składnia, uniwersalny, niezależny od platformy kod i przede wszystkich bezpłatny dostęp do doskonałych narzędzi programistycznych. Javę doceniło już wielu twórców oprogramowania, wśród których brakowało jednak twórców gier i aplikacji „rozrywkowych". Dotychczas w Javie tworzono jedynie proste układanki, gry karciane i łamigłówki lub nieśmiertelne aplety typu „padający śnieg", które przez długi czas straszyły nas z przeglądarek internetowych. Czas na zmianę! Wykorzystaj swoje umiejętności programowania w Javie, sięgnij po wiadomości zawarte w tej książce i napisz prawdziwą grę – z grafiką, inteligentnymi przeciwnikami, wydajnym silnikiem 3D wspomaganym sprzętowo i przestrzennym dźwiękiem.

„Java. Tworzenie gier" to książka o programowaniu gier, na jaką czekałeś. Zawiera zarówno opis podstawowych mechanizmów używanych w grach, jak i szczegółowe omówienie zaawansowanych technik. Dowiesz się, jak wykorzystać platformę Java 1.4 do tworzenia szybkich, pełnoekranowych gier akcji, przygodówek i trójwymiarowych strzelanek. Nauczysz się tworzyć wspomaganą sprzętowo grafikę, algorytmy sztucznej inteligencji i znajdowania drogi, realistyczne efekty dźwiękowe i mechanizmy obsługi gry dla wielu graczy.

- Algorytmy wyświetlania grafiki 2D
- Tworzenie interfejsu użytkownika z wykorzystaniem komponentów Swing
- Programowanie efektów dźwiękowych działających w czasie rzeczywistym
- Klient i serwer gry wieloosobowej
- Wyświetlanie grafiki 3D
- Mapowanie tekstur i symulacja oświetlenia
- Drzewa BSP
- Algorytmy detekcji kolizji i wykrywania drogi
- Sztuczna inteligencja i tworzenie botów
- Zapisywanie stanu gry
- Optymalizacja kodu
- System sterowania grą

Udowodnij „fachowcom" krytykującym szybkość Javy, że nie mają racji. Napisz wspaniałą grę w Javie. W tej książce znajdziesz wszystkie wiadomości, które są do tego niezbędne.

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

Spis treści

O Autorze **15**

Wstęp **17**

Część I Podstawy gier w języku Java **25**

Rozdział 1. Wątki w języku Java **27**

Co to jest wątek? 28
Tworzenie i uruchamianie wątków w języku Java 28
Rozszerzanie klasy Thread 29
Implementacja interfejsu Runnable 29
Użycie anonimowej klasy wewnętrznej 29
Oczekiwanie na zakończenie wątku 30
Uśpione wątki 30
Synchronizacja 30
Po co nam synchronizacja? 30
Jak synchronizować? 31
Kiedy należy synchronizować? 32
Kiedy nie synchronizować? 33
Unikanie zakleszczeń 33
Użycie wait() oraz notify() 34
Model zdarzeń Javy 35
Kiedy używać wątków? 36
Kiedy nie używać wątków? 36
Podsumowanie: pule wątków 36
Podsumowanie 41

Rozdział 2. Grafika 2D oraz animacja **43**

Grafika pełnoekranowa 44
Układ ekranu 44
Kolor piksela i głębia koloru 45
Częstotliwość odświeżania 46
Przełączanie do trybu pełnoekranowego 46
Anti-aliasing 50
Który tryb graficzny należy zastosować? 51
Rysunki 52
Przezroczystość 52
Formaty plików 52
Odczytywanie rysunków 53

Rysunki przyspieszane sprzętowo ... 56
Program testujący wydajność rysowania rysunków ... 58
Animacja ... 61
Renderowanie aktywne ... 64
Pętla animacji ... 64
Usuwanie migotania i szarpania ... 67
Podwójne buforowanie ... 67
Przełączanie stron ... 68
Odświeżanie monitora i szarpanie ... 70
Klasa BufferStrategy ... 70
Tworzenie zarządcy ekranów ... 71
Duszki ... 78
Proste efekty ... 84
Przekształcenia rysunków ... 84
Podsumowanie ... 89

Rozdział 3. Interaktywność i interfejs użytkownika ... 91
Model zdarzeń AWT ... 94
Zdarzenia klawiatury ... 95
Zdarzenia myszy ... 99
Przesuwanie myszy metodą „rozglądania się” ... 103
Ukrywanie wskaźnika myszy ... 107
Tworzenie klasy InputManager ... 108
Zastosowanie obiektu InputManager ... 119
Zatrzymywanie gry ... 119
Dodajemy grawitację ... 120
Projektowanie intuicyjnych interfejsów użytkownika ... 125
Wskazówki do projektu interfejsu użytkownika ... 126
Wykorzystanie komponentów Swing ... 127
Podstawy Swing ... 127
Stosowanie Swing w trybie pełnoekranowym ... 128
Tworzenie prostego menu ... 130
Konfiguracja klawiatury przez użytkownika ... 135
Podsumowanie ... 141

Rozdział 4. Efekty dźwiękowe oraz muzyka ... 143
Podstawy dźwięku ... 144
API Java Sound ... 144
Otwieranie pliku dźwiękowego ... 145
Zastosowanie interfejsu Line ... 145
Odtwarzanie dźwięków ... 146
Tworzenie architektury filtrów działających w czasie rzeczywistym ... 151
Tworzenie filtra „echo”, działającego w czasie rzeczywistym ... 155
Emulacja dźwięku 3D ... 159
Mechanizmy potrzebne do tworzenia filtra 3D ... 160
Implementacja filtra 3D ... 161
Testowanie filtra 3D ... 163
Tworzenie obiektu zarządzającego dźwiękiem ... 167
Klasa Sound ... 167
Klasa SoundManager ... 168
Zmienne lokalne dla wątków ... 175
Odtwarzanie muzyki ... 176
Odtwarzanie dźwięku CD ... 177
Odtwarzanie plików MP3 i Ogg Vorbis ... 177

Odtwarzanie muzyki MIDI ... 178
Tworzenie muzyki adaptacyjnej ... 182
Podsumowanie ... 184

Rozdział 5. Tworzenie dwuwymiarowej gry platformowej ... 185
Tworzenie mapy złożonej z kafelków ... 186
Implementacja mapy korzystającej z kafelków ... 187
Ładowanie mapy złożonej z kafelków ... 190
Rysowanie mapy złożonej z kafelków ... 193
Rysowanie duszków ... 195
Przewijanie z paralaksą ... 195
Premie ... 197
Proste obiekty wrogów ... 200
Wykrywanie kolizji ... 207
Detekcja kolizji ... 207
Obsługa kolizji ... 208
Kolizje duszków ... 212
Dokończenie i przyspieszanie gry ... 213
Tworzenie wykonywalnego pliku .jar ... 213
Pomysły na rozszerzenie gry ... 215
Podsumowanie ... 216

Rozdział 6. Gry wieloosobowe ... 217
Rewolucja w bibliotekach wejścia-wyjścia w języku Java ... 219
Przegląd bibliotek NIO z JDK 1.4 ... 220
Kanały ... 220
Bufory ... 223
Selektory oraz klasy SelectionKey ... 226
ChatterBox, prosta aplikacja dla wielu użytkowników ... 227
Serwer: ChatterServer ... 227
Kompilowanie i uruchamianie serwera ... 233
Klient: ChatterClient ... 235
Kompilowanie i uruchamianie klienta ... 236
Szkielet serwera gry wieloosobowej ... 237
Cele projektu i taktyka ... 237
Projekt ... 238
Wspólne klasy i interfejsy ... 242
Implementacja serwera ... 244
Klient ... 252
Przykładowa gra: RPS (kamień, papier, nożyczki) ... 253
Klasy ... 253
Uruchamianie gry RPS ... 256
Wykończanie gry: rozbudowa szkieletu ... 258
Interfejs klienta ... 258
Trwałość ... 258
Listy znajomych, pokoje i czat ... 258
Administracja serwera ... 259
Rejestracja zdarzeń ... 259
Uruchamianie i wyłączanie ... 260
Konsole administracyjne serwera ... 261
Śledzenie gry ... 261
Zagadnienia zaawansowane ... 262
Rozłączenia i ponowne połączenia ... 262
Tunelowanie HTTP ... 263

Testowanie za pomocą botów ... 265
Te nieznośne modemy ... 266
Profilowanie i tworzenie statystyk wydajności ... 266
Dostrajanie wydajności ... 268
Podsumowanie ... 270

Część II Grafika trójwymiarowa i zaawansowane techniki programowania gier ... 271

Rozdział 7. Grafika trójwymiarowa ... 273
Typy renderowania grafiki trójwymiarowej ... 274
Nie zapominajmy o matematyce ... 275
Trygonometria i trójkąty prostokątne ... 276
Wektory ... 276
Podstawy grafiki trójwymiarowej ... 281
Algebra trzech wymiarów ... 284
Wielokąty ... 289
Przekształcenia przestrzeni trójwymiarowej ... 292
Rotacje ... 293
Hermetyzacja przekształceń rotacji i translacji ... 295
Stosowanie transformacji ... 298
Porządek rotacji ... 300
Prosty potok tworzenia grafiki 3D ... 301
Ruch kamery ... 305
Bryły i usuwanie niewidocznych powierzchni ... 305
Iloczyn skalarny wektorów ... 307
Iloczyn wektorowy wektorów ... 308
Dodatkowe właściwości iloczynu skalarnego i wektorowego ... 311
Rysowanie wielokątów za pomocą konwertera skanującego ... 312
Optymalizowanie konwertera skanującego za pomocą liczb stałoprzecinkowych ... 317
Przycinanie w trzech wymiarach ... 321
Ostateczny potok renderowania ... 324
Podsumowanie ... 332

Rozdział 8. Mapowanie tekstur i oświetlenie ... 333
Podstawy mapowania tekstur uwzględniającego perspektywę ... 334
Wyprowadzenie równań wykorzystywanych do mapowania tekstur ... 335
Prosty mechanizm mapowania tekstur ... 340
Wady naszego prostego mechanizmu renderującego ... 347
Optymalizowanie mapowania tekstur ... 348
Przechowywanie tekstur ... 349
Prosta optymalizacja ... 352
Rozwijanie metod w miejscu wywołania ... 356
Przykładowy program korzystający z szybkiego mapowania tekstur ... 358
Prosty mechanizm generowania oświetlenia ... 359
Odbicie rozproszone ... 359
Światło otoczenia ... 360
Uwzględnianie intensywności światła pochodzącego ze źródła światła ... 360
Spadek intensywności światła wraz z odległością ... 360
Implementowanie punktowego źródła światła ... 361
Implementowanie oświetlania tekstur ... 362
Tworzenie zaawansowanych trików oświetleniowych za pomocą map cieniowania ... 369
Odnajdywanie prostokąta ograniczającego ... 369
Stosowanie mapy cieniowania ... 371

Budowanie mapy cieniowania ... 373
Budowanie powierzchni ... 375
Przechowywanie powierzchni w pamięci podręcznej ... 378
Przykład z cieniowaniem powierzchni ... 384
Dodatkowe pomysły ... 385
Sugerowanie głębi ... 385
Fałszywe cienie ... 386
Mapowanie MIP ... 386
Interpolacja dwuliniowa ... 386
Interpolacja trójliniowa ... 387
Mapy wektorów normalnych i mapy głębokości ... 387
Inne typy oświetlenia ... 388
Podsumowanie ... 388

Rozdział 9. Obiekty trójwymiarowe ... 389
Usuwanie ukrytych powierzchni ... 390
Algorytm malarza ... 390
Odwrotny algorytm malarza ... 391
Z-bufor ... 391
Z-bufor z wartościami 1/z ... 393
Obliczanie z-głębokości ... 396
Animacja trójwymiarowa ... 397
Ruch postępowy ... 400
Ruch obrotowy ... 402
Grupy wielokątów ... 407
Iteracyjna obsługa wszystkich wielokątów należących do grupy ... 411
Wczytywanie grup wielokątów z pliku OBJ ... 413
Format pliku OBJ ... 414
Format pliku MTL ... 420
Obiekty w grze ... 421
Zarządzanie obiektami w grze ... 425
Łączenie elementów ... 427
Możliwe rozszerzenia w przyszłości ... 433
Podsumowanie ... 434

Rozdział 10. Zarządzanie sceną trójwymiarową za pomocą drzew BSP ... 435
Wprowadzenie do drzew BSP ... 436
Podstawy drzew binarnych ... 437
Jednowymiarowe drzewo BSP ... 440
Dwuwymiarowe drzewo BSP ... 442
Przykład budowy drzewa BSP ... 443
Przykład przeglądania drzewa BSP ... 447
Implementacja dwuwymiarowego drzewa BSP ... 448
Linia podziału BSP ... 450
Wyznaczanie położenia punktu względem linii ... 450
Dwójkowy podział wielokąta ... 454
Przeglądanie drzewa BSP ... 455
Przeglądanie poprzeczne ... 457
Przeglądanie od przodu do tyłu ... 458
Budowa drzewa ... 459
Znajdowanie punktu przecięcia dwóch prostych ... 463
Przycinanie wielokątów do linii ... 465
Usuwanie pustych przestrzeni T-złączy ... 467
Testowanie drzewa BSP ... 469

Rysowanie wielokątów od przodu do tyłu ... 471
Pierwszy przykład wykorzystania drzewa BSP ... 479
Rysowanie obiektów na scenie ... 480
Wczytywanie map z pliku ... 482
Łączenie elementów ... 486
Rozszerzenia ... 486
Podsumowanie ... 488

Rozdział 11. Wykrywanie kolizji ... 489
Podstawy kolizji ... 490
Kolizje typu obiekt-obiekt ... 491
Eliminowanie testów ... 491
Sfery otaczające ... 493
Walce otaczające ... 495
Problem przetwarzania dyskretno-czasowego ... 498
Kolizje typu obiekt-świat ... 499
Prostopadłościany otaczające, wykorzystywane do wykrywania kolizji z podłogami ... 499
Znajdowanie liścia drzewa BSP dla danego położenia ... 500
Implementacja testów wysokości podłogi i sufitu ... 501
Prostopadłościany otaczające, wykorzystywane do testowania kolizji ze ścianami ... 503
Punkt przecięcia z odcinkiem wielokąta reprezentowanego w drzewie BSP ... 504
Problem narożników ... 508
Implementacja wykrywania kolizji typu obiekt-świat ... 509
Prosty program demonstracyjny wykrywający kolizje ... 511
Obsługa kolizji z przesuwaniem ... 512
Przesuwanie obiektu wzdłuż innego obiektu ... 512
Przesuwanie obiektu wzdłuż ściany ... 515
Grawitacja i płynny ruch na schodach (przesuwanie obiektu wzdłuż podłogi) ... 517
Skakanie ... 520
Program demonstracyjny obsługujący kolizje z przesuwaniem ... 521
Rozszerzenia ... 522
Podsumowanie ... 522

Rozdział 12. Odnajdywanie drogi w grze ... 523
Podstawowa wiedza na temat technik odnajdywania drogi ... 524
Pierwsze przymiarki do odnajdywania drogi w grze ... 524
Przeszukiwanie wszerz ... 527
Podstawy algorytmu A* ... 530
Stosowanie algorytmu A* w grze ... 535
Algorytm A* w połączeniu z drzewami BSP ... 536
Przejścia ... 536
Implementowanie portali ... 537
Uniwersalny mechanizm odnajdywania drogi ... 541
Przygotowywanie robota PathBot ... 545
Sposoby ulepszania przeszukiwania A* ... 549
Podsumowanie ... 550

Rozdział 13. Sztuczna inteligencja ... 551
Podstawy sztucznej inteligencji ... 552
Pozbawianie botów ich boskiej mocy ... 553
„Widzenie” ... 554
„Słyszenie” ... 556

Maszyny stanów i obsługa reakcji ... 559
Maszyny probabilistyczne ... 561
Przydatne funkcje generujące liczby losowe ... 563
Podejmowanie decyzji ... 564
Wzorce ... 566
Unikanie ... 567
Atakowanie ... 569
Uciekanie ... 572
Celowanie ... 573
Strzelanie ... 574
Tworzenie obiektów ... 575
Łączenie elementów ... 576
Mózgi! ... 576
Zdrowie i umieranie ... 577
Dodawanie HUD-a ... 581
Uczenie się ... 585
Wskrzeszanie botów ... 586
Uczenie się botów ... 588
Rozszerzenia programu demonstracyjnego ... 593
Inne podejścia do sztucznej inteligencji ... 593
Zespołowa sztuczna inteligencja ... 594
Podsumowanie ... 594

Rozdział 14. Skrypty gry ... 595
Książka kucharska skryptów: czego potrzebujemy ... 596
Implementacja powiadomień wejścia i wyjścia ... 597
Wyzwalacze ... 599
Nasłuch obiektów gry ... 600
Skrypty ... 604
Projektowanie skryptu ... 606
Wbudowywanie BeanShell ... 608
Zdarzenia opóźnione ... 612
Tworzenie zdarzeń opóźnionych w BeanShell ... 615
Łączymy wszystko razem ... 616
Rozszerzenia ... 617
Podsumowanie ... 619

Rozdział 15. Trwałość — zapisywanie gry ... 621
Podstawy zapisywania gier ... 621
Wykorzystanie API serializacji do zapisywania stanu gry ... 623
Wprowadzenie do serializacji ... 623
Serializacja: podstawy ... 623
Serializacja: zasady ... 626
Serializacja: pułapki ... 629
Zmiana domyślnego działania mechanizmu serializacji ... 633
Tworzenie zrzutu ekranu gry ... 636
Tworzenie miniatury ze zrzutem ekranu ... 638
Zapisywanie rysunku ... 639
Zapisywanie gier we właściwym miejscu ... 641
Podsumowanie ... 642

Część III Optymalizacja i kończenie gry 643

Rozdział 16. Techniki optymalizacji 645

Zasady optymalizacji 646
Profilowanie 646
Testowanie wydajności 647
Użycie programu profilującego HotSpot 647
HotSpot 651
Optymalizacje specyficzne dla języka Java 652
Eliminacja nieużywanego kodu 652
Wyciąganie niezmienników pętli 653
Eliminacja wspólnych podwyrażeń 653
Propagacja stałych 653
Rozwijanie pętli (tylko maszyna wirtualna server) 654
Metody inline 654
Sztuczki optymalizacji 655
Algorytmy 655
Zmniejszanie siły operacji: przesuwanie bitów 656
Zmniejszanie siły operacji: reszta z dzielenia 657
Zmniejszanie siły operacji: mnożenie 657
Zmniejszanie siły operacji: potęgowanie 658
Więcej na temat wyciągania niezmienników pętli 658
Tablice wartości funkcji 659
Arytmetyka stałoprzecinkowa 662
Wyjątki 662
Wejście-wyjście 662
Pliki mapowane w pamięci 663
Wykorzystanie pamięci i zbieranie nieużytków 664
Sterta Java oraz zbieranie nieużytków 664
Monitorowanie zbierania nieużytków 665
Monitorowanie użycia pamięci 666
Dostrajanie sterty 670
Dostrajanie procesu zbierania nieużytków 671
Redukowanie tworzenia obiektów 672
Ponowne wykorzystanie obiektów 672
Pule obiektów 673
Zauważalna wydajność 674
Rozdzielczość zegara 674
Podsumowanie 679

Rozdział 17. Tworzenie dźwięków i grafiki gry 681

Wybór wyglądu i sposobu działania gry 682
Szukanie inspiracji 682
Zachowywanie spójności 683
Zdobywanie darmowych materiałów do gry 683
Praca z grafikami i autorami dźwięków 684
Narzędzia 685
Tworzenie dźwięków 685
Formaty plików dźwiękowych 687
Tworzenie tekstur i duszków 687
Formaty plików graficznych 688
Tworzenie tekstur bezszwowych 689
Tworzenie tekstur zastępczych 690

Tworzenie tekstur przejściowych ... 691
Tworzenie wielopoziomowych tekstur ... 692
Tworzenie obrazów tytułowych i grafiki dla wyświetlaczy HUD ... 694
Tworzenie grafiki interfejsu użytkownika ... 694
Dostosowywanie komponentów Swing ... 694
Tworzenie własnych czcionek ... 696
Podsumowanie ... 701

Rozdział 18. Projekt gry i ostatnie 10% prac ... 703
Ostatnie 10% prac ... 704
Efekty ... 704
Maszyna stanów gry ... 705
Elementy projektu gry ... 711
Środowiska ... 712
Fabuła ... 712
Właściwa gra ... 714
Uczenie gracza sposobu gry ... 716
Tworzenie edytora map ... 717
Szukanie błędów ... 719
Problemy z szukaniem błędów w Java2D ... 722
Rejestrowanie zdarzeń ... 723
Ochrona kodu ... 725
Dystrybucja gry ... 726
Dystrybucja gry za pomocą Java Web Start ... 728
Pobieranie zasobów z plików .jar ... 728
Podpisywanie plików .jar ... 728
Tworzenie pliku JNLP ... 729
Konfigurowanie serwera WWW ... 731
Dystrybucja gry w postaci kompilacji natywnej ... 732
Aktualizacje i łatki ... 733
Problem obciążenia serwera ... 733
Opinie użytkowników i testy beta ... 734
Zarabianie pieniędzy ... 736
Łączymy wszystko razem ... 737
Podsumowanie ... 737

Rozdział 19. Przyszłość ... 739
Ewolucja Javy ... 739
Java Community Process ... 740
Bug Parade ... 740
Przyszłość: Java 1.5 „Tiger” ... 740
Szablony (JSR 14) ... 741
Wyliczenia (JSR 201) ... 742
Statyczne importowanie (JSR 201) ... 743
Poprawiona pętla for (JSR 201) ... 744
Interfejs API kompilatora (JSR 199) ... 744
Format transferu sieciowego (JSR 200) ... 745
Współdzielona maszyna wirtualna (JSR 121) ... 745
Wymagania stawiane platformie Java ... 746
Potrzebne: więcej opcji dla obsługi myszy i klawiatury ... 746
Potrzebne: obsługa joysticka ... 747
Potrzebne: przyspieszane sprzętowo, półprzezroczyste obrazy ... 747
Potrzebne: dokładniejszy zegar ... 747

Potrzebne: grafika przyspieszana sprzętowo i tryb pełnoekranowy w systemie Linux ... 748
Potrzebne: trójwymiarowa grafika przyspieszana sprzętowo, włączona do środowiska Javy ... 748
Potrzebne: optymalizacja rozkazów SIMD w maszynie HotSpot ... 749
Pożądane: więcej opcji wygładzania czcionek ... 750
Pozostałe możliwości ... 752
Nowe urządzenia i Javy Games Profile (JSR 134) ... 752
Podsumowanie ... 753

Dodatki ... 755

Skorowidz ... 757

Rozdział 11.
Wykrywanie kolizji

W tym rozdziale:

- Podstawy kolizji.
- Kolizje typu obiekt-obiekt.
- Kolizje typu obiekt-świat.
- Prosty program demonstracyjny wykrywający kolizje.
- Obsługa kolizji z przesuwaniem.
- Program demonstracyjny obsługujący kolizje z przesuwaniem.
- Rozszerzenia.
- Podsumowanie.

Pamiętasz grę *Pong*?

Niezależnie od tego, czy mamy do czynienia z piłeczką odbijaną rakietkami, laserem trafiającym robota, poszukiwaczem skarbów wpadającym w pułapkę, bohaterem znajdującym dodatkową amunicję, dwoma walczącymi potworami czy po prostu z graczem idącym przy ścianie — niemal w każdej grze trzeba zastosować jakiś mechanizm wykrywania kolizji.

W rozdziale 5., „Tworzenie dwuwymiarowej gry platformowej", stworzyliśmy prosty dwuwymiarowy system wykrywania kolizji, który świetnie się sprawdzał w przypadku nieskomplikowanej gry dwuwymiarowej. W tym rozdziale spróbujemy rozszerzyć omówione tam zagadnienia i więcej czasu poświęcić na wykrywanie kolizji pomiędzy obiektami (typu obiekt-obiekt) oraz pomiędzy obiektami a wielokątami przechowywanymi w drzewach BSP (typu obiekt-świat).

Zajmiemy się także niemniej ważnym zagadnieniem wyboru sposobu obsługi kolizji po jej wykryciu, np. poruszaniem się wzdłuż ścian lub umożliwieniem obiektom w grze samodzielnej obsługi swojego zachowania w przypadku kolizji.

Podstawy kolizji

Wykrywanie kolizji, wbrew nazwie, tak naprawdę nie sprowadza się do samego wykrywania. Z kolizjami związane są trzy interesujące nas zagadnienia:

- **Decydowanie, które kolizje chcemy wykrywać**. Byłoby stratą czasu testowanie, czy dwa obiekty ze sobą nie kolidują, jeśli znajdują się na dwóch różnych krańcach świata przedstawionego w grze. Należy się także zastanowić, czy w świecie z 1 000 poruszających się obiektów ma sens testowanie wystąpienia kolizji pomiędzy każdą parą obiektów — czyli w sumie 999 000 testów? Powinniśmy więc próbować maksymalnie ograniczyć liczbę obiektów, w przypadku których staramy się wykryć wystąpienie zdarzenia kolizji. Najprostszym sposobem na wprowadzenie takiego ograniczenia jest testowanie tylko obiektów znajdujących się stosunkowo blisko gracza.
- **Wykrywanie kolizji**. Wybór techniki wykrywania kolizji zależy od oczekiwanej dokładności kolizji w grze. Można oczywiście zastosować doskonały algorytm wykrywania kolizji i sprawdzać wszystkie wielokąty należące do obiektu ze wszystkimi wielokątami tworzącymi inny obiekt, należy jednak brać pod uwagę związany z takim działaniem koszt obliczeniowy. Podobnie, w świecie dwuwymiarowym moglibyśmy testować ewentualne kolizje wszystkich pikseli jednego dwuwymiarowego obiektu ze wszystkimi pikselami innego obiektu. W grach komputerowych stosuje się zwykle mniej dokładne techniki wykrywania kolizji, które jednak wykonują swoje zadanie znacznie szybciej.
- **Obsługa kolizji**. Jeśli dany obiekt koliduje z innym elementem sceny, należy być przygotowanym na różne sposoby obsługi różnych typów kolizji. Przykładowo, pocisk kolidujący z robotem może spowodować zniszczenie zarówno pocisku, jak i robota. Obiekt dochodzący do ściany może się dalej przemieszczać wzdłuż tej ściany. Podobnych przykładów jest wiele.

Podsumujmy — w grze powinny być podejmowane próby realizowania następujących celów w zakresie wykrywania kolizji:

- eliminowanie jak najwięcej testów wystąpienia kolizji;
- szybkie podejmowanie decyzji, czy kolizja wystąpiła;
- zapewnianie mechanizmu wykrywania kolizji o wystarczającej precyzji;
- obsługiwanie kolizji w naturalny sposób, który nie będzie niepotrzebnie zwracał uwagi gracza podczas gry.

Ostatni cel oznacza także to, że nie powinniśmy zbytnio ograniczać możliwości ruchu gracza podczas gry. Przykładowo gracz nie powinien być całkowicie zatrzymywany po kolizji ze ścianą. Zamiast tego powinniśmy umożliwić mu poruszanie się wzdłuż ściany lub spowodować jego nieznaczne odbicie.

Nie chcemy także, by nasz mechanizm wykrywania kolizji był na tyle niedokładny, by gracz mógł oszukiwać, np. chowając się w ścianie w niektórych miejscach mapy.

Zaczniemy od prostego algorytmu wykrywania kolizji. Wszystkie poruszające się elementy w grze będziemy traktować jak obiekty, niezależnie od tego, czy będzie to potwór, gracz, pocisk lub cokolwiek innego. W przypadku każdego tak ogólnie zdefiniowanego obiektu będziemy wykonywali następujące kroki:

1. Zaktualizuj położenie obiektu.
2. Sprawdź, czy nie występuje kolizja z innymi obiektami lub z elementami środowiska.
3. Jeśli wykryto kolizję, ustaw obiekt na jego wcześniejszej pozycji.

Zauważ, że po każdym ruchu obiektu sprawdzamy, czy nie wystąpiła kolizja. Alternatywnym rozwiązaniem jest przeniesienie w nowe miejsca wszystkich obiektów i dopiero potem sprawdzenie występowania ewentualnych kolizji. W takim przypadku należałoby jednak przechowywać dane o poprzednich położeniach wszystkich obiektów — pojawiłby się problem, gdyby się okazało, że trzy lub więcej obiektów powoduje kolizje.

Zauważ także, że ten podstawowy algorytm w przypadku wystąpienia kolizji po prostu odstawia obiekt na jego wcześniejsze położenie. Zwykle będziemy chcieli stosować inne sposoby obsługi kolizji, zależne od ich typu.

Skoro podstawowy algorytm został już omówiony, przejdźmy do wykrywania i obsługi kolizji typu obiekt-obiekt w praktyce.

Kolizje typu obiekt-obiekt

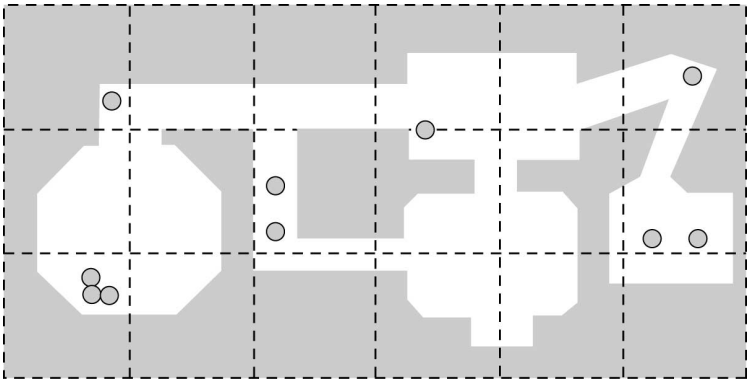
Niezależnie od tego, jakiej precyzji oczekujemy od stosowanego algorytmu wykrywania kolizji, idealnym rozwiązaniem jest wyeliminowanie maksymalnej liczby testów kolizji typu obiekt-obiekt i wcześniejsze wykonanie kilku innych testów, wskazujących na duże prawdopodobieństwo wystąpienia kolizji pomiędzy parą obiektów.

Eliminowanie testów

Jest oczywiste, że w przypadku obiektu, który nie wykonuje żadnego ruchu od ostatniej klatki, nie są potrzebne żadne testy wykrywające kolizję powodowaną przez ten obiekt. Przykładowo skrzynia znajdująca się w pomieszczeniu nie może powodować kolizji z żadnym innym elementem sceny. Inne obiekty mogą oczywiście kolidować z tą skrzynią, jednak takie kolizje są obsługiwane przez te obiekty, a nie przez statyczną skrzynię.

Aby wyeliminować maksymalną liczbę przyszłych testów na występowanie kolizji, powinniśmy ograniczyć zbiór badanych obiektów do tych, które znajdują się w swoim bezpośrednim sąsiedztwie.

Jednym ze sposobów takiej eliminacji jest przyporządkowanie obiektów do pól specjalnej siatki (patrz rysunek 11.1). Każdy obiekt musi należeć do dokładnie jednego z takich pól. Nawet jeśli obiekt częściowo zajmuje obszar wielu pól, trzeba zdecydować



Rysunek 11.1. *Aby ograniczyć liczbę testów na występowanie kolizji typu obiekt-obiekt, możemy przyporządkować obiekty do pól siatki i wykonywać testy dla obiektów znajdujących się w tych samych i sąsiadujących polach*

o jego przyporządkowaniu do dokładnie jednego z nich. Dzięki temu obiekt będzie musiał testować wystąpienie kolizji wyłącznie z obiektami z pola, w którym się znajduje, oraz z pól sąsiadujących z tym polem.

Innymi sposobami izolowania obiektów jest tworzenie siatek jedno- i trójwymiarowych. Przykładowo, w dwuwymiarowej grze przewijanej, obiekty można posortować zgodnie z ich współrzędną *x*, dzięki czemu będzie można wykonywać testy tylko dla obiektów sąsiadujących na liście. W przypadku gier trójwymiarowych obiekty można odizolować za pomocą siatki trójwymiarowej zamiast dwuwymiarowej — każda komórka będzie wówczas miała kształt sześcianu, nie kwadratu. W tym rozdziale do wykrywania kolizji w grach z mechanizmem trójwymiarowym będziemy jednak stosowali siatkę w prostszej wersji dwuwymiarowej.

Izolowanie obiektów za pomocą siatki umożliwia także łatwe usuwanie obiektów ze sceny. Przykładowo możemy zastosować rozwiązanie, w którym rysowane są tylko obiekty w widocznych polach. Jeśli wykorzystujemy drzewo BSP, możemy rysować tylko te obiekty, które znajdują się w polach z widocznymi liśćmi.

Kod przypisujący obiekty do pól siatki jest trywialny — na potrzeby przykładów z tego rozdziału zaimplementowaliśmy go w klasie `GridGameObjectManager`. Kiedy obiekt jest aktualizowany, wywoływana jest metoda `checkObjectCollision()` (patrz listing 11.1), która odpowiada za wykrywanie ewentualnych kolizji pomiędzy tym obiektem a obiektami znajdującymi się w tych samych i w przylegających polach siatki.

Listing 11.1. *Sprawdzanie przylegających komórek (plik GridGameObjectManager.java)*

```
/**
    Klasa Cell reprezentuje komórkę siatki. Klasa zawiera listę obiektów
    w grze i wskaźnik ich widoczności.
*/
private static class Cell {
    List objects;
    boolean visible;

    Cell() {
        objects = new ArrayList();
        visible = false;
    }
}
```

```
...

/**
    Sprawdza, czy dany obiekt koliduje z którymkolwiek z pozostałych
    obiektów.
*/
public boolean checkObjectCollision(GameObject object, Vector3D oldLocation)
{

    boolean collision = false;

    // Użyj współrzędnych (x,z) dla obiektu (pozycja w poziomie).
    int x = convertMapXtoGridX((int)object.getX());
    int y = convertMapYtoGridY((int)object.getZ());

    // Sprawdź 9 komórek otaczających obiekt.
    for (int i=x-1; i<=x+1; i++) {
        for (int j=y-1; j<=y+1; j++) {
            Cell cell = getCell(i, j);
            if (cell != null) {
                collision |= collisionDetection.checkObject(object, cell.objects,
                oldLocation);
            }
        }
    }

    return collision;
}
```

Powyższy kod wywołuje zdefiniowaną w klasie `CollisionDetection` metodę `checkObject()`, która wykrywa kolizję pomiędzy danym obiektem a listą obiektów. Implementacją klasy `CollisionDetection` zajmiemy się za chwilę.

Sfery otaczające

Niedokładną, ale szybką techniką wykrywania kolizji jest zastosowanie tzw. *sfer otaczających*; przykład takiej sfery otaczającej obiekt przedstawia rysunek 11.2.



Rysunek 11.2. *Do wykrywania kolizji można użyć sfer otaczających*

Kiedy sfery otaczające dwa obiekty kolidują ze sobą, ich kolizja jest traktowana jak kolizja otaczanych przez nie obiektów. Oto nasza pierwsza próba zaimplementowania testu wykrywającego kolizję dwóch sfer otaczających:

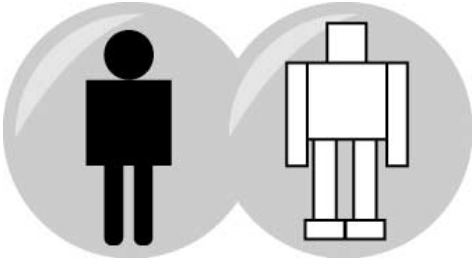
```
dx = objectA.x - objectB.x;
dy = objectA.y - objectB.y;
dz = objectA.z - objectB.z;
minDistance = objectA.radius + objectB.radius;
if (Math.sqrt(dx*dx + dy*dy + dz*dz) < minDistance) {
    // Wykryto kolizję.
}
```

Wywołanie funkcji `Math.sqrt()` wiąże się jednak z wykonaniem dużej ilości obliczeń. Możemy uniknąć tego wywołania, podnosząc do kwadratu obie strony nierówności i otrzymując prostsze oraz znacznie szybciej obliczane wyrażenie warunkowe:

```
if (dx*dx + dy*dy + dz*dz < minDistance* minDistance) {
    // Wykryto kolizję.
}
```

Jeśli Twoja gra wyświetla obraz dwuwymiarowy zamiast scen trójwymiarowych, zamiast kolizji sfer możesz testować wystąpienia kolizji okręgów, wystarczy usunąć z równań składniki odnoszące się do współrzędnej *z*.

Testowanie występowania kolizji pomiędzy sferami otaczającymi jest stosunkowo proste, ale także bardzo niedokładne. Przykładowo na rysunku 11.3 widać sferę otaczającą gracza, która koliduje ze sferą otaczającą robota, chociaż same obiekty gracza i robota wcale ze sobą nie kolidują.

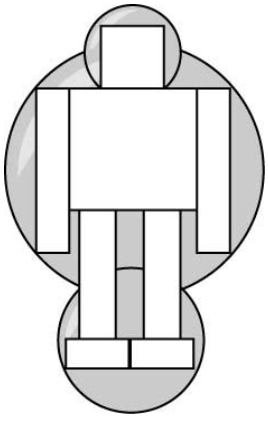


Rysunek 11.3. *Niedokładność sfer otaczających: para sfer ze sobą koliduje, mimo że otaczane obiekty znajdują się w pewnej odległości*

Oczywiście taka niedokładność w wielu grach nie będzie dla gracza zauważalna. Przykładowo w przypadku szybkiej gry akcji, w której biegamy po pomieszczeniach oraz podnosimy apteczki i amunicję, prawdopodobnie nie będzie dla nas miało znaczenia, czy podnosimy te obiekty na moment przed ich faktycznym dotknięciem. Jednak w innych sytuacjach taki brak precyzji może być irytujący, np. kiedy uda Ci się zranić przeciwnika, którego nawet nie dotknąłeś.

Po otrzymaniu pozytywnego wyniku testu kolizji sfer otaczających możemy pójść krok dalej i wykonać bardziej szczegółowe testy, np. sprawdzić ewentualne występowanie kolizji pomiędzy wszystkimi parami wielokątów tworzących oba obiekty.

Inną metodą jest wykorzystanie do testów zbioru sfer otaczających (patrz rysunek 11.4). Na rysunku dla robota skonstruowano trzy sfery otaczające, które bardziej precyzyjnie opisują jego kształt. Po otrzymaniu pozytywnego wyniku testu kolizji podstawowych (najmniej dokładnych) sfer otaczających możemy przetestować drugi (bardziej dokładny) zbiór sfer. Jeśli którakolwiek ze zbioru sfer gracza koliduje z którąkolwiek ze zbioru sfer robota, wówczas możemy uznać, że oba obiekty ze sobą kolidują.



Rysunek 11.4.
Wiele sfer otaczających można wykorzystać do przeprowadzania bardziej precyzyjnych testów występowania kolizji

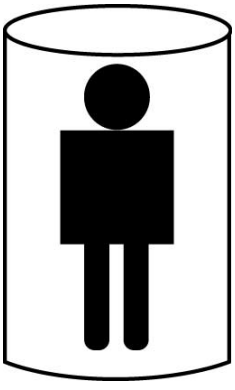
Mamy więc dwa poziomy sfer dla wszystkich obiektów w grze. Nie musimy oczywiście na tym poprzestawać. Moglibyśmy dodać jeszcze kilka poziomów, z których każdy opierałby się na większej liczbie sfer otaczających i zapewniałby większą dokładność. Taka technika jest często nazywana drzewem sfer lub podziałem sfer. Możemy w ten sposób szybko wykluczyć z przetwarzania obiekty, które ze sobą nie kolidują, oraz wykonać bardziej precyzyjne testy dla obiektów powodujących potencjalną kolizję. Omawiane podejście pozwala nam także stwierdzić, która część obiektu została trafiona, dzięki czemu możemy wykrytą kolizję odpowiednio obsłużyć. Przykładowo robota trafionego pociskiem rakietowym możemy pozbawić uszkodzonej nogi.

Zauważ, że drzewa sfer otaczających musza się obracać wraz z obracającym się otaczanym obiektem. Przykładowo sfery muszą zmieniać swoje położenie wraz ze zmianą położenia ramienia robota. W kodzie zaprezentowanym w rozdziale 9., „Obiekty trójwymiarowe", zdefiniowaliśmy trójwymiarowe obiekty reprezentowane jako zagnieżdżone grupy wielokątów. Ponieważ była to już struktura drzewiasta, w celu zaimplementowania drzewa sfer możemy nadać każdej grupie wielokątów jej własny zbiór sfer i zapewnić przełożenie ruchu grup na odpowiedni ruch sfer. W tym rozdziale nie będziemy implementować drzew sfer, warto jednak brać pod uwagę takie rozwiązanie podczas tworzenia gier.

Podsumujmy: przed narysowaniem typowej klatki większość obiektów nie wymaga przeprowadzania testów wystąpienia kolizji. W niektórych przypadkach konieczne jest przeprowadzenie prostego testu, a kilka obiektów wymaga najbardziej skomplikowanych i kosztownych obliczeniowo testów na wystąpienie kolizji.

Walce otaczające

Alternatywą dla sfer otaczających są pionowe walce otaczające (patrz rysunek 11.5). Takie rozwiązanie pozwala zredukować testy na występowanie kolizji do sprawdzenia dwuwymiarowych okręgów i testów położenia pionowych linii (a więc w jednym wymiarze).



Rysunek 11.5. *Do wykrywania kolizji można także wykorzystać pionowy walec otaczający*

Pionowe walce otaczające najlepiej nadają się do — lepszego niż w przypadku pojedynczej sfery otaczającej — opisywania wysokich, cienkich obiektów (np. graczy lub potworów). W tym rozdziale w naszym mechanizmie trójwymiarowym zaimplementujemy wykrywanie kolizji typu obiekt-obiekt właśnie przy wykorzystaniu pionowych walców otaczających.

Całość prezentowanego kodu odpowiadającego za proste wykrywanie kolizji umieściliśmy w klasie `CollisionDetection`. Zawarte w tej klasie metody obsługujące kolizje typu obiekt-obiekt przedstawiono na listingu 11.2.

Listing 11.2. *Sprawdzanie obiektów CollisionDetection.java*

```
/**
    Sprawdza, czy dany obiekt koliduje z dowolnym innym obiektem
    przechowywanym na przekazanej liście.
*/
public boolean checkObject(GameObject objectA, List objects, Vector3D oldLocation)
{
    boolean collision = false;
    for (int i=0; i<objects.size(); i++) {
        GameObject objectB = (GameObject)objects.get(i);
        collision |= checkObject(objectA, objectB,
            oldLocation);
    }
    return collision;
}


/**
    Zwraca wartość true, jeśli dwa przekazane obiekty ze sobą kolidują.
    Obiekt A porusza się, a obiekt B podlega sprawdzeniu. Metoda wykorzystuje
    do wykrywania kolizji pionowe walce otaczające (o okrągłej podstawie).
*/
public boolean checkObject(GameObject objectA,
    GameObject objectB, Vector3D oldLocation)
{
    // Obiekt nie może kolidować z samym sobą.
    if (objectA == objectB) {
        return false;
    }
```

```
        PolygonGroupBounds boundsA = objectA.getBounds();
        PolygonGroupBounds boundsB = objectB.getBounds();

        // Najpierw sprawdzamy kolizje w osi y (zakładamy, że wysokość jest stała).
        float Ay1 = objectA.getY() + boundsA.getBottomHeight();
        float Ay2 = objectA.getY() + boundsA.getTopHeight();
        float By1 = objectB.getY() + boundsB.getBottomHeight();
        float By2 = objectB.getY() + boundsB.getTopHeight();
        if (By2 < Ay1 || By1 > Ay2) {
            return false;
        }

        // Następnie sprawdzamy dwa wymiary - kolizje powierzchni x/z
        // (okrągłych podstaw walców).
        float dx = objectA.getX() - objectB.getX();
        float dz = objectA.getZ() - objectB.getZ();
        float minDist = boundsA.getRadius() + boundsB.getRadius();
        float distSq = dx*dx + dz*dz;
        float minDistSq = minDist * minDist;
        if (distSq < minDistSq) {
             return handleObjectCollision(objectA, objectB, distSq,
                minDistSq, oldLocation);
        }
        return false;
    }


    /**
        Obsługuje kolizję obiektu. Obiekt A porusza się, natomiast obiekt B jest tym
        obiektem, z którym koliduje obiekt A.
    */
    protected boolean handleObjectCollision(GameObject objectA, GameObject objectB,
    float distSq, float minDistSq, Vector3D oldLocation)
    {
        objectA.notifyObjectCollision(objectB);
        return true;
    }
```

W powyższym kodzie obiekty w grze są związane z obiektami klasy `PolygonGroupBounds`, które otaczają tworzące je grupy wielokątów odpowiednimi walcami. Opis walca otaczającego składa się z promienia podstawy walca oraz wysokości, na jakiej znajduje się podstawa dolna walca (zwykle 0) i podstawa górna.

Metoda `checkObject()` jedynie sprawdza, czy dwa walce otaczające ze sobą kolidują; jeśli tak się dzieje, wywoływana jest metoda `handleObjectCollision()`.

Metoda `handleObjectCollision()` jedynie sygnalizuje (za pomocą metody `notifyObjectCollision()`) poruszającemu się obiektowi, że jego ruch spowodował kolizję. Ta i inne metody umożliwiające sygnalizowanie podobnych zdarzeń znajdują się w klasie `GameObject`, nie ma w ich przypadku jednak domyślnie zdefiniowanych działań — w razie potrzeby podklasy klasy `GameObject` mogą przesłaniać te metody. Przykładowo zdefiniowana w klasie `Blast` metoda `notifyObjectCollision()` jest wykorzystywana do niszczenia robota, który koliduje z pociskiem reprezentowanym przez obiekt tej klasy:

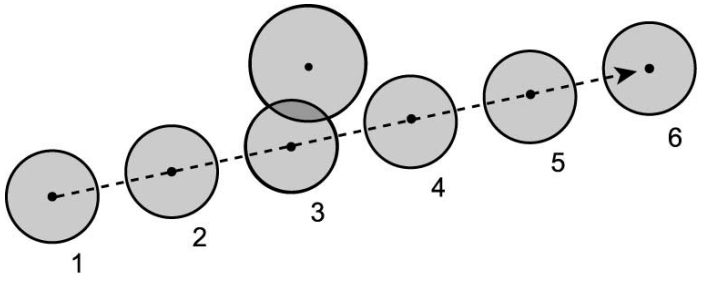
```
public void notifyObjectCollision(GameObject object) {
    // Niszczy robota i sam pocisk.
    if (object instanceof Bot) {
        setState(object, STATE_DESTROYED);
        setState(STATE_DESTROYED);
    }
}
```

Na razie w klasie `GridGameObjectManager` zastosujemy rozwiązanie, w którym poruszający się obiekt powodujący kolizję jest przenoszony do swojej poprzedniej lokalizacji. W przyszłości opracujemy bardziej realistyczną reakcję na kolizje typu obiekt-obiekt, czyli przemieszczanie się wzdłuż obiektu.

Problem przetwarzania dyskretno-czasowego

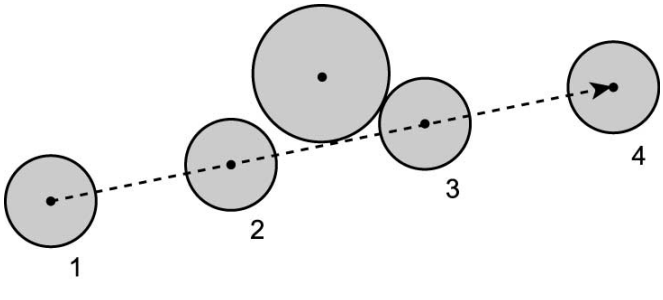
Typowa gra aktualizuje swój stan w dyskretnych odstępach czasowych — w taki właśnie sposób aktualizujemy położenie każdego obiektu na podstawie czasu, jaki upłynął od ostatniej aktualizacji. Przykładowo na rysunku 11.6 widać widziany z góry ruch obiektu, ujęty w kolejnych odstępach czasu. Widać wyraźnie, jak poruszający się obiekt koliduje z większym obiektem w trzeciej klatce.

Rysunek 11.6. *Widziane z góry kolejne lokalizacje poruszającego się obiektu*

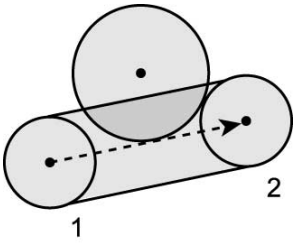


Niestety, taki sposób obsługi ruchu obiektów może uniemożliwiać prawidłowe wykrywanie kolizji. Wyobraź sobie, że obiekt porusza się szybciej lub szybkość odtwarzania klatek jest mniejsza. W takiej sytuacji poruszający się obiekt może „minąć" obiekt, z którym faktycznie powinien kolidować. Przykładowo na rysunku 11.7 poruszający się obiekt koliduje z większym obiektem pomiędzy drugą a trzecią klatką.

Rysunek 11.7. *Problem powodowany przez przetwarzanie dyskretno-czasowe: obiekt może „minąć" inny obiekt w sytuacji, gdy powinna zostać wykryta kolizja*



Istnieje kilka rozwiązań tego problemu. Bardziej precyzyjnym, ale też bardziej kosztownym obliczeniowo sposobem jest połączenie walców (lub innych brył) otaczających poruszający się obiekt w jedną bryłę, od początkowej do końcowej lokalizacji (patrz rysunek 11.8).



Rysunek 11.8. *Poruszający się obiekt można traktować jak „rurę", co rozwiązuje problem powodowany przez przetwarzanie dyskretno-czasowe*

Alternatywnym rozwiązaniem jest testowanie wystąpień kolizji w dodatkowych punktach pomiędzy początkową i końcową lokalizacją poruszającego się obiektu. Na rysunku 11.7 takie punkty moglibyśmy dodać w połowie odległości pomiędzy każdą klatką.

Kolizje typu obiekt-świat

Kolizje obiektów ze środowiskiem, w którym występują, powinny być obsługiwane ze szczególną starannością. Nie chcemy przecież, by gracz lub inny obiekt mógł przechodzić przez ścianę lub by nienaturalnie się trząsł podczas przemieszczania się wzdłuż ściany.

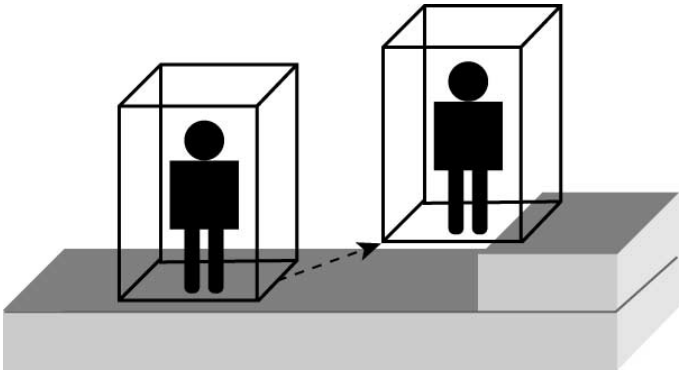
W rozdziale 5. zaimplementowaliśmy kolizje typu obiekt-świat, zmieniając w tym samym czasie tylko jedną współrzędną (najpierw *x*, potem *y*), co sprawdzało się doskonale w przypadku prostego świata dwuwymiarowego, w którym obiekty nie przemieszczają się szybciej niż o jedną jednostkę w klatce.

W trójwymiarowym świecie nie istnieją tak zdefiniowane jednostki — zwykle możemy jednak opisać taki świat za pomocą struktury ułatwiającej wykrywanie kolizji, np. drzewa BSP. Do zaimplementowania kolizji obiektów z podłogami, sufitami i ścianami wykorzystamy opracowane w poprzednim rozdziale dwuwymiarowe drzewo BSP.

Prostopadłościany otaczające, wykorzystywane do wykrywania kolizji z podłogami

W dwuwymiarowej grze, stworzonej w rozdziale 5., do wykrywania kolizji potrzebowaliśmy prostokątów otaczających. W środowisku trójwymiarowym, poza stosowaniem sfer, okręgów, walców i prostopadłościanów otaczających, istnieje jeszcze kilka innych popularnych mechanizmów wykrywania kolizji. Dostępne są dwa typy prostopadłościanów otaczających: o dowolnym ustawieniu i dopasowane do osi. Prostopadłościany otaczające o dowolnym ustawieniu można swobodnie obracać, natomiast prostopadłościany dopasowane do osi muszą zawsze być wyrównane do osi *x*, *y* i *z*. W tym rozdziale będziemy wykorzystywali pionowe walce otaczające do wykrywania

kolizji typu obiekt-obiekt, natomiast do wykrywania kolizji typu obiekt-świat użyjemy dopasowanych do osi prostopadłościanów otaczających. W pierwszej kolejności zajmiemy się kolizjami z podłogami i sufitami. Chcielibyśmy, by w środowisku, w którym podłogi mogą się znajdować na różnych wysokościach, obiekty znajdowały się na najwyższym poziomie podłogi pod prostopadłościanem otaczającym (patrz rysunek 11.9). Podobnie nie chcemy, by obiekty poruszały się w obszarach, gdzie strop jest dla nich zbyt niski. Chcielibyśmy także umożliwić obiektom w grze pokonywanie niewysokich stopni bez zatrzymywania. W sytuacji przedstawionej na poniższym rysunku gracz może płynnie pokonać stopień znajdujący się nieznacznie wyżej niż podłoga.



Rysunek 11.9. *Kolizja prostopadłościanu otaczającego z podłogą (stopniem): prostopadłościan otaczający gracza częściowo znajduje się na stopniu, zatem wysokość tego stopnia jest wykorzystywana jako wysokość „podłogi" pod graczem*

Na rysunku 11.9 najwyższy poziom pod zastosowaną bryłą otaczającą obiekt jest wykorzystywany do wyznaczenia wysokości, na której ten obiekt się znajduje. Testując występowanie ewentualnych kolizji pomiędzy obiektem a podłogą lub sufitem w danym pomieszczeniu, możemy sprawdzić, czy którykolwiek z wierzchołków prostopadłościanu nie znajduje się pod powierzchnią podłogi lub nad powierzchnią sufitu. Oznacza to, że sprawdzenie, na jakiej wysokości powinien się znaleźć obiekt, wymaga przetestowania czterech wierzchołków prostopadłościanu otaczającego.

Warto zaznaczyć, że prostopadłościany otaczające można wykorzystać także do wykrywania kolizji typu obiekt-obiekt. Możemy także stworzyć drzewo prostopadłościanów otaczających, tak jak drzewo sfer otaczających.

Znajdowanie liścia drzewa BSP dla danego położenia

Informacje o podłodze są przechowywane w liściu dwuwymiarowego drzewa BSP. Istnieje możliwość stosunkowo łatwego odszukania właściwego liścia dla danej lokalizacji — wystarczy wykorzystać algorytm podobny do tego, za pomocą którego przeglądaliśmy drzewo w poprzednim rozdziale (patrz listing 11.3).

Listing 11.3. *Metoda getLeaf() dostępna w pliku BSPTree.java*

```
/**
    Zwraca liść, w którym znajduje się punkt o współrzędnych x, z.
*/
public Leaf getLeaf(float x, float z) {
    return getLeaf(root, x, z);
}
```

```
protected Leaf getLeaf(Node node, float x, float z) {
    if (node == null || node instanceof Leaf) {
        return (Leaf)node;
    }
    int side = node.partition.getSideThin(x, z);
    if (side == BSPLine.BACK) {
        return getLeaf(node.back, x, z);
    }
    else {
        return getLeaf(node.front, x, z);
    }
}
```

Za pomocą powyższej metody znajdujemy oczywiście liść tylko dla jednej lokalizacji, natomiast prostopadłościan otaczający obiekt może teoretycznie rozciągać się na kilka liści. Oznacza to, że powinniśmy znajdować liście dla wszystkich wierzchołków prostopadłościanu otaczającego.

Implementacja testów wysokości podłogi i sufitu

Testowanie wszystkich wierzchołków prostopadłościanu otaczającego w porównaniu z wysokościami podłogi i sufitu jest realizowane za pomocą kilku metod dostępnych w klasie `CollisionDetection` (patrz listing 11.4).

Listing 11.4. *Sprawdzanie kolizji z podłogą i sufitem (plik CollisionDetection.java)*

```
/**
    Wierzchołki otaczające obiekt w grze są wykorzystywane do
    wykrywania kolizji z poziomymi wielokątami (podłogami
    i sufitami), przechowywanymi w drzewie BSP. Wierzchołki są
    uporządkowane w kolejności albo zgodnej z kierunkiem ruchu
    wskazówek zegara, albo zgodnej z odwrotnym kierunkiem.
*/
private static final Point2D.Float[] CORNERS = {
    new Point2D.Float(-1, -1), new Point2D.Float(-1, 1),
    new Point2D.Float(1, 1), new Point2D.Float(1, -1),
};

...

    getFloorAndCeiling(object);
    checkFloorAndCeiling(object, elapsedTime);

...

/**
    Zwraca wartości opisujące podłogę i sufit dla danego obiektu
    w grze (obiektu klasy GameObject). Wywołuje metody
    object.setFloorHeight() i object.setCeilHeight() w celu
    ustawienia odpowiednich wartości dla podłogi i sufitu.
*/
public void getFloorAndCeiling(GameObject object) {
    float x = object.getX();
    float z = object.getZ();
```

```
        float r = object.getBounds().getRadius() - 1;
        float floorHeight = Float.MIN_VALUE;
        float ceilHeight = Float.MAX_VALUE;
        BSPTree.Leaf leaf = bspTree.getLeaf(x, z);
        if (leaf != null) {
            floorHeight = leaf.floorHeight;
            ceilHeight = leaf.ceilHeight;
        }

        // Sprawdza cztery otaczające punkty.
        for (int i=0; i<CORNERS.length; i++) {
            float xOffset = r * CORNERS[i].x;
            float zOffset = r * CORNERS[i].y;
            leaf = bspTree.getLeaf(x + xOffset, z + zOffset);
            if (leaf != null) {
                floorHeight = Math.max(floorHeight, leaf.floorHeight);
                ceilHeight = Math.min(ceilHeight, leaf.ceilHeight);
            }
        }

        object.setFloorHeight(floorHeight);
        object.setCeilHeight(ceilHeight);
    }


    /**
        Sprawdza, czy obiekt koliduje z podłogą i sufitem.
        Wykorzystuje metody object.getFloorHeight() i object.getCeilHeight()
        do otrzymania wartości opisujących podłogę i sufit.
    */
    protected void checkFloorAndCeiling(GameObject object, long elapsedTime)
    {
        boolean collision = false;

        float floorHeight = object.getFloorHeight();
        float ceilHeight = object.getCeilHeight();
        float bottomHeight = object.getBounds().getBottomHeight();
        float topHeight = object.getBounds().getTopHeight();

        if (!object.isFlying()) {
             object.getLocation().y = floorHeight - bottomHeight;
        }
        // Sprawdza, czy część obiektu nie znajduje się poniżej podłogi.
        if (object.getY() + bottomHeight < floorHeight) {
            object.notifyFloorCollision();
            object.getTransform().getVelocity().y = 0;
            object.getLocation().y = floorHeight - bottomHeight;
        }
        // Sprawdza, czy część obiektu nie znajduje się powyżej sufitu.
        else if (object.getY() + topHeight > ceilHeight) {
            object.notifyCeilingCollision();
            object.getTransform().getVelocity().y = 0;
            object.getLocation().y = ceilHeight - topHeight;
        }

    }
```

Metoda `getFloorAndCeiling()` zwraca wysokość, na jakiej znajduje się podłoga i sufit w miejscu, gdzie znajduje się obiekt — weryfikuje dane z liści odpowiadających położeniu czterech wierzchołków obiektu. Pamiętaj, że w danym momencie obiekt może się znajdować w więcej niż jednym liściu drzewa BSP.

Do ustawiania położenia obiektu w osi *y* wykorzystujemy metodę `checkFloorAndCeiling()`. Jeśli obiekt nie unosi się w powietrzu, wartość *y* odpowiada wysokości, na jakiej znajduje się podłoga. W przeciwnym przypadku metoda sprawdza, czy obiekt nie koliduje z podłogą lub sufitem. Jeśli tak się dzieje, wywoływana jest udostępniana przez obiekt metoda `notifyFloorCollision()` lub `notifyCeilingCollision()`. Podobnie jak w przypadku metody `notifyObjectCollision()`, wymienione dwie metody nie mają domyślnie zdefiniowanych działań — podklasy klasy `GameObject` mogą przykrywać te metody.

Zaprezentowany sposób implementowania kolizji z podłogą i sufitem jest dosyć prymitywny — ruch gracza jest mało realistyczny. W dalszej części tego rozdziału zaimplementujemy efekt grawitacji i możliwość płynnego poruszania się po schodach, dzięki czemu mechanizm obsługi kolizji stanie się znacznie bardziej realistyczny.

Prostopadłościany otaczające, wykorzystywane do testowania kolizji ze ścianami

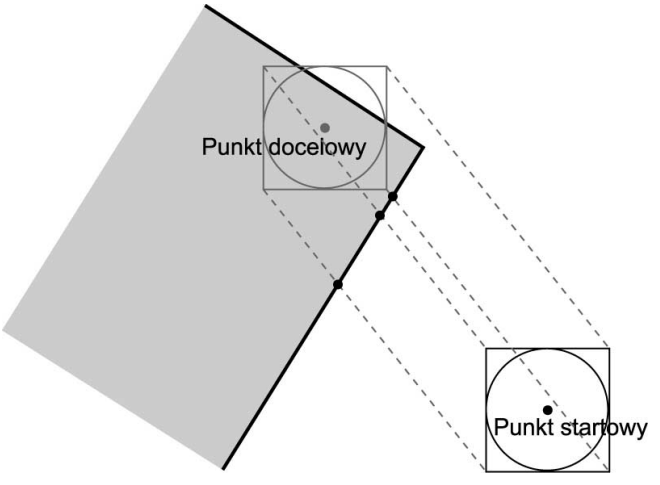
W przypadku podłóg i sufitów przeprowadzaliśmy testy na wystąpienia kolizji dopiero po tym, jak interesujący nas obiekt wykonał jakiś ruch — mogliśmy dzięki temu określić, na jakiej wysokości znajduje się podłoga pod obiektem.

Ściany są jednak cienkimi liniami, zatem jeśli będziemy przeprowadzać testy na wystąpienie kolizji tylko po wykonaniu przez obiekt ruchu, istnieje możliwość, że nasza reakcja będzie spóźniona i nie zarejestrujemy przejścia obiektu przez ścianę. Aby dokładnie wykrywać, czy obiekt uderzył w ścianę, trzeba analizować całą drogę przebytą przez obiekt od czasu ostatniej aktualizacji (między kolejnymi klatkami).

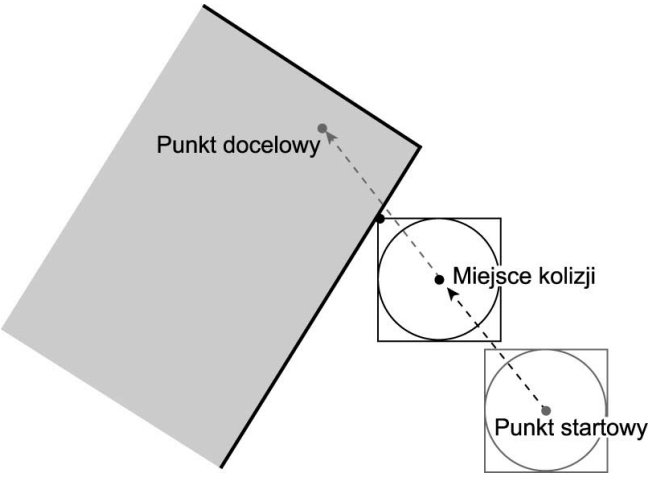
Jeśli używamy dwuwymiarowego drzewa BSP, droga przebyta przez obiekt pomiędzy dwiema kolejnymi klatkami jest odcinkiem, możemy więc sprawdzać, czy istnieje punkt przecięcia tego odcinka z którąkolwiek z linii reprezentujących ścianę (podobne rozwiązanie moglibyśmy zastosować w przypadku trójwymiarowego drzewa BSP — wówczas badalibyśmy istnienie punktu przecięcia prostej reprezentującej przebytą drogę z płaszczyzną reprezentującą ścianę).

Obiekty są oczywiście bryłami, a nie punktami. Jeśli więc do wykrywania kolizji używamy prostopadłościanów otaczających obiekty, konieczne jest testowanie wszystkich czterech wierzchołków podstawy prostopadłościanu otaczającego ze ścianami na scenie (patrz rysunek 11.10). Na poniższym rysunku testujemy punkty przecięcia czterech ścieżek (po jednej dla każdego wierzchołka) ze ścianą — takie punkty istnieją dla trzech ścieżek.

Jeśli wykryjemy więcej niż jeden punkt przecięcia (jak na rysunku 11.10), do obsługi kolizji wykorzystujemy najkrótsza ścieżkę od punktu startowego do punktu przecięcia (patrz rysunek 11.11). W tym przypadku lewy górny wierzchołek otoczenia obiektu jest



Rysunek 11.10. *Sprawdzanie punktów przecięcia ścian i prostych reprezentujących ścieżki przebyte przez wierzchołki podstawy prostopadłościanu otaczającego obiekt. Każdy odcinek jest porównywany w ten sposób ze wszystkimi ścianami przechowywanymi w drzewie BSP*



Rysunek 11.11. *Miejsce kolizji jest określane na podstawie tego wierzchołka prostopadłościanu otaczającego, który znajduje się najbliżej odpowiedniego punktu przecięcia*

tym punktem, który jako pierwszy koliduje ze ścianą (znajduje się najbliżej odpowiedniego punktu przecięcia), zatem właśnie w oparciu o ten punkt wyznaczamy miejsce kolizji.

Trzeba znaleźć możliwie szybką metodę określania, czy istnieje punkt przecięcia danej ścieżki z którymkolwiek z wielokątów trójwymiarowego świata. Jeśli taki punkt istnieje, należy znaleźć pierwszy odcinek kolizji w drzewie BSP.

Punkt przecięcia z odcinkiem wielokąta reprezentowanego w drzewie BSP

Rozważmy ścieżkę z punktu ($x1$, $y1$) do punktu ($x2$, $y2$). Naszym celem jest znalezienie najbliższego punktu przecięcia (jeśli w ogóle taki punkt istnieje) tej ścieżki z dowolnym wielokątem przechowywanym w drzewie BSP. Najbliższy punkt przecięcia to taki, który znajduje się najbliżej punktu ($x1$, $y1$).

Oto algorytm realizujący to zadanie, począwszy od korzenia drzewa BSP:

1. Sprawdź istnienie punktu przecięcia ścieżki z podziałem reprezentowanym przez ten węzeł. Jeśli ścieżka znajduje się z przodu lub z tyłu linii podziału, sprawdź odpowiednio przednie lub tylne węzły.
2. W przeciwnym przypadku — jeśli ścieżka przecina linię podziału — podziel tę ścieżkę w punkcie przecięcia z linią podziału na dwie ścieżki. Jedna ścieżka reprezentuje wówczas pierwszą część ścieżki, druga reprezentuje drugą część tej samej ścieżki.
   a. Sprawdź, czy istnieją ewentualne punkty przecięcia pierwszej część ścieżki (patrz krok 1).
   b. Jeśli nie znajdziesz punktu przecięcia, sprawdź, czy nie istnieją punkty przecięcia z wielokątami przechowywanymi w tym węźle.
   c. Jeśli nie znajdziesz punktów przecięcia, sprawdź, czy nie istnieją punkty przecięcia drugiej części ścieżki z wielokątami przechowywanymi w tym węźle (patrz krok 1).
   d. Jeśli znajdziesz punkt przecięcia, zwróć dane tego punktu. W przeciwnym przypadku zwróć wartość `null`.

Mówiąc najprościej, zadaniem powyższego algorytmu jest zbadanie wszystkich linii podziałów przecinających się ze ścieżką — od pierwszej (najbliższej) do ostatniej (najdalszej) linii podziału — i zwrócenie pierwszego znalezionego punktu przecięcia (jeśli taki punkt w ogóle istnieje).

Zauważ, że fakt wzajemnego przecinania się ścieżki z linią podziału nie musi oznaczać występowania kolizji. Aby taka kolizja zaistniała, muszą być spełnione trzy warunki:

- Ścieżka musi przecinać przechowywany w drzewie BSP odcinek, który reprezentuje wielokąt.
- Wielokąt nie może być na tyle mały, aby obiekt mógł przejść nad nim, oraz musi znajdować się na wysokości uniemożliwiającej swobodne poruszanie się obiektu (nie może być ani za nisko, ani za wysoko).
- Ścieżka musi prowadzić od przedniej do tylnej strony wielokąta.

Algorytm znajdujący punkt przecięcia (i sprawdzający powyższe warunki występowania kolizji) zaimplementowaliśmy za pomocą kodu przedstawionego na listingu 11.5.

Listing 11.5. *Punkt przecięcia z odcinkiem wielokąta reprezentowanego w drzewie BSP (plik CollisionDetection.java)*

```
private BSPTree bspTree;
private BSPLine path;
private Point2D.Float intersection;

...
```

```
/**
    Zwraca punkt przecięcia (jeśli w ogóle istnieje) pomiędzy ścieżką
    (x1,z1)->(x2,z2) a ścianami reprezentowanymi w drzewie BSP.
    Zwraca albo pierwszy przecinany obiekt BSPPolygon, albo wartość null (jeśli
    nie istnieje żaden punkt przecięcia).
*/
public BSPPolygon getFirstWallIntersection(float x1, float z1, float x2, float z2,
float yBottom, float yTop)
{
    return getFirstWallIntersection(bspTree.getRoot(), x1, z1, x2, z2, yBottom, yTop);
}


/**
    Zwraca punkt przecięcia (jeśli w ogóle istnieje) pomiędzy ścieżką
    (x1,z1)->(x2,z2) a ścianami reprezentowanymi w drzewie BSP, począwszy
    od danego węzła. Zwraca albo pierwszy przecinany obiekt BSPPolygon,
    albo wartość null (jeśli nie istnieje żaden punkt przecięcia).
*/
protected BSPPolygon getFirstWallIntersection(BSPTree.Node node, float x1, float z1,
float x2, float z2, float yBottom, float yTop)
{
    if (node == null || node instanceof BSPTree.Leaf) {
        return null;
    }

    int start = node.partition.getSideThick(x1, z1);
    int end = node.partition.getSideThick(x2, z2);
    float intersectionX;
    float intersectionZ;

    if (end == BSPLine.COLLINEAR) {
        end = start;
    }

    if (start == BSPLine.COLLINEAR) {
        intersectionX = x1;
        intersectionZ = z1;
    }
    else if (start != end) {
        path.setLine(x1, z1, x2, z2);
        node.partition.getIntersectionPoint(path,intersection);
        intersectionX = intersection.x;
        intersectionZ = intersection.y;
    }
    else  {
        intersectionX = x2;
        intersectionZ = z2;
    }

    if (start == BSPLine.COLLINEAR && start == end) {
        return null;
    }
```

```
        // Sprawdza przednią część linii podziału.
        if (start != BSPLine.COLLINEAR) {
            BSPPolygon wall = getFirstWallIntersection(
                (start == BSPLine.FRONT)?node.front:node.back,
                x1, z1, intersectionX, intersectionZ,
                yBottom, yTop);
            if (wall != null) {
                return wall;
            }
        }

        // Testuje otoczenie obiektu.
        if (start != end || start == BSPLine.COLLINEAR) {
            BSPPolygon wall = getWallCollision(node.polygons,
                    x1, z1, x2, z2, yBottom, yTop);
            if (wall != null) {
                intersection.setLocation(intersectionX,
                    intersectionZ);
                return wall;
            }
        }

        // Sprawdza tylną część linii podziału.
        if (start != end) {
            BSPPolygon wall = getFirstWallIntersection(
                (end == BSPLine.FRONT)?node.front:node.back,
                intersectionX, intersectionZ, x2, z2,
                yBottom, yTop);
            if (wall != null) {
                return wall;
            }
        }

        // Nie znaleziono punktu przecięcia.
        return null;
    }


    /**
        Sprawdza, czy określona ścieżka koliduje z którymkolwiek z listy
        wielokątów leżących na linii. Ścieżka przecina linię reprezentowaną
        przez wielokąty, jednak nie zawsze oznacza to, że przecina same
        wielokąty.
    */
    protected BSPPolygon getWallCollision(List polygons, float x1, float z1, float x2,
    float z2, float yBottom, float yTop)
    {
        path.setLine(x1, z1, x2, z2);
        for (int i=0; i<polygons.size(); i++) {
            BSPPolygon poly = (BSPPolygon)polygons.get(i);
            BSPLine wall = poly.getLine();

            // Sprawdza, czy nie jest to ściana.
            if (wall == null) {
                continue;
            }
```

```
            // Sprawdza, czy znajduje się na tej samej wysokości (współrzędna y) co ściana.
            if (wall.top <= yBottom || wall.bottom > yTop) {
                continue;
            }

            // Sprawdza, czy ruch nie odbywa się od tyłu ściany.
            if (wall.getSideThin(x2, z2) != BSPLine.BACK) {
                continue;
            }

            // Sprawdza, czy ścieżka przecina samą ścianę.
            int side1 = path.getSideThin(wall.x1, wall.y1);
            int side2 = path.getSideThin(wall.x2, wall.y2);
            if (side1 != side2) {
                return poly;
            }
        }
        return null;
    }
```

W powyższym kodzie metoda `getFirstWallIntersection()` odpowiada za realizację omówionego wcześniej algorytmu, natomiast metoda `getWallCollision()` sprawdza wymienione warunki istnienia kolizji pomiędzy ścieżką ruchu obiektu a wielokątem.

Gdybyśmy wykorzystywali trójwymiarowe drzewo BSP, konieczne byłoby napisanie podobnego kodu do określania wysokości, na jakiej znajduje się podłoga pod obiektem. Wystarczyłoby użyć tego samego algorytmu do znajdowania najwyżej znajdującego się wielokąta pod graczem, czyli znaleźć pierwszy punkt przecięcia z odcinkiem znajdującym się bezpośrednio pod obiektem.

Mamy już niemal wszystkie potrzebne składniki implementacji kolizji prostopadłościanu otaczającego z wielokątami przechowywanymi w drzewie BSP — pozostaje nam jednak analiza jeszcze jednego zagadnienia, które może być źródłem problemów.

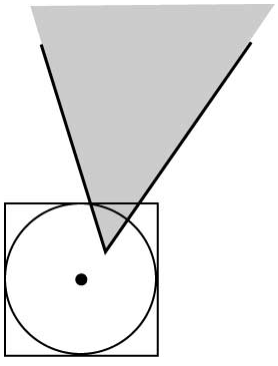
Problem narożników

Niestety zastosowanie prostopadłościanów otaczających nie daje całkowitej pewności co do skuteczności wykrywania kolizji! Jeśli sprawdzamy, czy którykolwiek z wierzchołków prostopadłościanu otaczającego nie koliduje ze światem, nie obsługujemy sytuacji, w której świat koliduje z prostopadłościanem, chociaż nie koliduje z żadnym z jego wierzchołków (patrz rysunek 11.12).

Na powyższym rysunku obiekt koliduje z ostrym narożnikiem, który nie dotyka żadnego z wierzchołków prostopadłościanu otaczającego ten obiekt.

Jednym ze sposobów ominięcia tego problemu jest sprawdzanie ewentualnych punktów przecięcia każdej krawędzi prostopadłościanu otaczającego z wielokątem reprezentowanym w drzewie BSP. Jeśli którakolwiek z krawędzi przecina wielokąt w drzewie BSP, obiekt jest cofany na oryginalną pozycję. Odpowiednią metodę zaimplementujemy jeszcze w tym rozdziale. Alternatywnym rozwiązaniem jest tylko nieznaczne wycofywanie

Rysunek 11.12. *Problemem może być kolizja obiektu z ostrym narożnikiem*



obiektu i ponowne przeprowadzenie testu na występowanie kolizji. Jeszcze innym rozwiązaniem jest upewnienie się, że poziomy nie będą projektowane w sposób uniemożliwiający wystąpienie tego typu sytuacji. Takie ograniczenia dotyczące poziomów w grze mogą jednak negatywnie wpłynąć na realizm gry i znacznie utrudnić pracę projektantom poziomów, dla których stosowanie ostrych narożników (podobnych do tego z powyższego przykładu) jest zupełnie naturalne.

Implementacja wykrywania kolizji typu obiekt-świat

Mamy już wszystkie elementy potrzebne do wykrywania kolizji typu obiekt-świat z wykorzystaniem drzewa BSP. Resztę kodu umieściliśmy na listingu 11.6.

Listing 11.6. *Sprawdzanie kolizji ze ścianami (CollisionDetection.java)*

```
// Sprawdź ściany, jeśli zmieniła się współrzędna x lub z.
if (object.getX() != oldLocation.x || object.getZ() != oldLocation.z)
{
    wallCollision = (checkWalls(object, oldLocation, elapsedTime) != null);
}

...

/**
    Sprawdza, czy występuje kolizja obiektu w grze ze ścianami
    reprezentowanymi w drzewie BSP. Zwraca pierwszą ścianę, z którą
    obiekt koliduje, lub wartość null, jeśli kolizja nie występuje.
*/
public BSPPolygon checkWalls(GameObject object,
    Vector3D oldLocation, long elapsedTime)
{
    Vector3D v = object.getTransform().getVelocity();
    PolygonGroupBounds bounds = object.getBounds();
    float x = object.getX();
    float y = object.getY();
    float z = object.getZ();
    float r = bounds.getRadius();
    float stepSize = 0;
    if (!object.isFlying()) {
        stepSize = BSPPolygon.PASSABLE_WALL_THRESHOLD;
    }
```

```
float bottom = object.getY() + bounds.getBottomHeight() +
    stepSize;
float top = object.getY() + bounds.getTopHeight();

// Wybierz najbliższy punkt przecięcia dla 4 wierzchołków.
BSPPolygon closestWall = null;
float closestDistSq = Float.MAX_VALUE;
for (int i=0; i<CORNERS.length; i++) {
    float xOffset = r * CORNERS[i].x;
    float zOffset = r * CORNERS[i].y;
    BSPPolygon wall = getFirstWallIntersection( oldLocation.x+xOffset,
    oldLocation.z+zOffset, x+xOffset, z+zOffset, bottom, top);
    if (wall != null) {
        float x2 = intersection.x-xOffset;
        float z2 = intersection.y-zOffset;
        float dx = (x2-oldLocation.x);
        float dz = (z2-oldLocation.z);
        float distSq = dx*dx + dz*dz;
        // Wybierz najbliższą ścianę lub - jeśli odległości
        // do ścian są równe - wybierz bieżącą ścianę (jeśli
        // jej przód jest zgodny ze zwrotem wektora ruchu).
        if (distSq < closestDistSq ||
            (distSq == closestDistSq &&
            MoreMath.sign(xOffset) == MoreMath.sign(v.x) &&
            MoreMath.sign(zOffset) == MoreMath.sign(v.z)))
        {
            closestWall = wall;
            closestDistSq = distSq;
            object.getLocation().setTo(x2, y, z2);
        }
    }
}

if (closestWall != null) {
    object.notifyWallCollision();
}

// Upewnij się, że bryła otaczająca gracza jest pusta
// (nie koliduje z ostrymi narożnikami).
x = object.getX();
z = object.getZ();
r-=1;
for (int i=0; i<CORNERS.length; i++) {
    int next = i+1;
    if (next == CORNERS.length) {
        next = 0;
    }
    // Użyj wartości (r-1), by nie przeszkadzać w obliczeniach
    // dotyczących zwykłych kolizji.
    float xOffset1 = r * CORNERS[i].x;
    float zOffset1 = r * CORNERS[i].y;
    float xOffset2 = r * CORNERS[next].x;
    float zOffset2 = r * CORNERS[next].y;

    BSPPolygon wall = getFirstWallIntersection(
        x+xOffset1, z+zOffset1, x+xOffset2, z+zOffset2,
        bottom, top);
```

```
            if (wall != null) {
                object.notifyWallCollision();
                object.getLocation().setTo(
                    oldLocation.x, object.getY(), oldLocation.z);
                return wall;
            }
        }

        return closestWall;
    }
```

Na początku w powyższym kodzie testowane jest występowanie punktów przecięcia pomiędzy ścianami a czterema ścieżkami, po jednej dla każdego wierzchołka podstawy prostopadłościanu otaczającego. Jeśli algorytm wykryje więcej niż jeden punkt przecięcia, zostanie wybrany punkt najbliższy.

Jeśli dwa punkty przecięcia znajdują się w tej samej odległości od obiektu (bryły otaczającej), wybierany jest punkt położony bliżej wektora ruchu obiektu. Takie rozwiązanie ułatwi nam w dalszej części tego rozdziału implementację przemieszczania się obiektu wzdłuż ściany.

Jeśli zostanie znaleziony i wybrany dokładnie jeden punkt przecięcia, obiekt jest ustawiany w miejscu kolizji, bezpośrednio przy ścianie, z którą koliduje.

Po wykonaniu tych testów w przedstawionym kodzie sprawdzane jest, czy prostopadłościan otaczający obiekt na pewno jest pusty (nie koliduje z ostrymi narożnikami — patrz rysunek 11.12). Jeśli tak się dzieje, obiekt jest przywracany na pierwotną pozycję.

To wszystko — zaimplementowaliśmy właśnie prosty mechanizm wykrywania i obsługi kolizji! Wypróbujmy teraz, jak nasze rozwiązanie działa w praktyce.

Prosty program demonstracyjny wykrywający kolizje

Program demonstracyjny `CollisionTest` (dołączony do kodów źródłowych opracowanych dla tej książki) pokazuje działanie utworzonego do tej pory mechanizmu wykrywania kolizji. Program bardzo przypomina program `BSPMapTest` z poprzedniego rozdziału — dodano jedynie mechanizm wykrywania kolizji i zastosowano klasę `GridGameObjectManager`.

W programie obiekt (gracz) jest zatrzymywany w momencie, gdy dojdzie do ściany; gracz może także wchodzić po schodach na wyższe poziomy. Gracz jest ponadto zatrzymywany, kiedy dojdzie do innego obiektu, np. robota lub skrzyni. Wystrzeliwane przez gracza pociski niszczą roboty i — dla uzyskania ciekawszego efektu — wbijają się w ściany, podłogi, sufity i inne obiekty (zamiast przez nie przelatywać).

Mechanizm wykrywania kolizji w tym programie działa doskonale, jednak wymaga kilku istotnych ulepszeń:

- Kolizja ze ścianą powoduje zablokowanie możliwości ruchu. Takie rozwiązanie byłoby nie do przyjęcia w trójwymiarowej grze FPP lub TPP, w której gracze oczekują możliwości poruszania się wzdłuż ścian i bezpośrednio przy nich.
- Poruszanie się w górę i w dół po schodach wygląda mało realistycznie, ponieważ gracz jest momentalnie przenoszony w górę lub w dół zamiast płynnie pokonywać kolejne stopnie lub opadać na skutek działania siły grawitacji.
- Gracz nie może przekroczyć małego obiektu na swojej drodze. Wystarczy strzelić w podłogę i spróbować przekroczyć wbity pocisk — jest to niestety niemożliwe.
- Bohater nie może skakać. Nie dotyczy to oczywiście mechanizmu wykrywania kolizji, ale tego elementu wyraźnie brakuje.

Chyba nas trochę poniosło — przecież celem tego rozdziału jest obsługa kolizji, wróćmy więc do rozwiązywania związanych z tym problemów! Mimo że tak naprawdę to zagadnienie nie jest związane z obsługą kolizji, w tym momencie warto poświęcić trochę czasu na zaimplementowanie zarówno skakania, jak i wykrywania kolizji w czasie, gdy obiekt uderza w coś, będąc w powietrzu (podczas skoku).

Obsługa kolizji z przesuwaniem

Kolejnym krokiem jest opracowanie takiego mechanizmu obsługi kolizji, który nie będzie przeszkadzał graczowi — czyli obsługi kolizji zgodnej z oczekiwaniami gracza.

Zamiast blokowania kolidujących obiektów zaimplementujemy ich przesuwanie się względem siebie. Przykładowo, zamiast zatrzymywać gracza wchodzącego na dany obiekt, gracz będzie ten obiekt płynnie omijał. Gracz będzie także mógł się poruszać wzdłuż ścian oraz przekraczać niewielkie obiekty znajdujące się na podłodze.

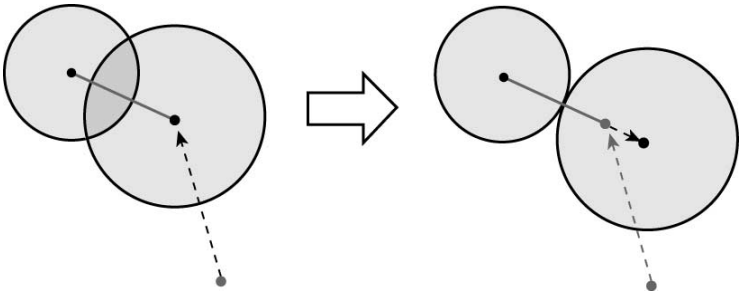
W tym podrozdziale zaimplementujemy także kilka prostych elementów lepiej odwzorowujących fizyczne zachowania obiektów — możliwe będzie płynne pokonywanie schodów, uwzględnianie siły grawitacji, a także skakanie gracza.

Przesuwanie obiektu wzdłuż innego obiektu

Do tej pory, kiedy następowała kolizja typu obiekt-obiekt, po prostu przywracaliśmy poruszający się obiekt na jego pierwotną pozycję. W efekcie obiekt był momentalnie „odbijany” od statycznego obiektu, z którym kolidował.

Aby to poprawić, należy zapewnić możliwość poruszania się obiektu wzdłuż jednej z krawędzi statycznego obiektu. Logicznym rozwiązaniem jest przesuwanie poruszającego się obiektu w taki sposób, by po przebyciu jak najkrótszej drogi wzdłuż obiektu statycznego mógł on swobodnie poruszać się dalej. Oznacza to, że kierunek przesuwania obiektu jest definiowany za pomocą wektora od środka obiektu statycznego do

środka obiektu, który się porusza (patrz rysunek 11.13). Na rysunku widać poruszający się większy obiekt, który koliduje z mniejszym obiektem statycznym. Większy obiekt omija mniejszy obiekt w taki sposób, że oba obiekty są styczne, ale ze sobą nie kolidują.



Rysunek 11.13. *Przesuwanie obiektu wzdłuż innego obiektu: poruszający się obiekt jest „odpychany" od obiektu statycznego*

Długość drogi wzdłuż obiektu statycznego jest różnicą pomiędzy drogą minimalną a drogą rzeczywistą:

```
float minDist = objectA.radius + objectB.radius;
float slideDistance = minDist - actualDist;
```

Oznacza to, że ponieważ wektor łączący środki obu obiektów ma długość `actualDist`, wzór przyjmuje postać:

```
float scale = slideDistance / actualDist;
vector.multiply(scale);
```

Jeśli zamiast tego chcemy otrzymać tylko odległość podniesioną do kwadratu, wykonamy instrukcje:

```
float scale = (float)Math.sqrt(minDistSq / actualDistSq) - 1;
vector.multiply(scale);
```

Funkcja obliczająca pierwiastek kwadratowy działa co prawda stosunkowo wolno, jednak trzeba ją wywołać tylko w momencie, gdy obiekt zderza się z innym obiektem, a taka sytuacja nie zdarza się zbyt często.

Przesuwanie obiektów powoduje problemy, kiedy obiekt przemieszczający się wzdłuż obiektu statycznego powoduje kolizję ze ścianą lub innym obiektem. W takim przypadku przesuwany obiekt jest po prostu przenoszony do początkowego położenia, co oznacza, że pozostaje efekt „odbijania" obiektu — gracz musi wówczas zmienić kierunek ruchu.

Implementacja przesuwania obiektu wzdłuż innego obiektu jest stosunkowo prosta. Cały potrzebny kod umieściliśmy w klasie `CollisionDetectionWithSliding`, która jest podklasą klasy `CollisionDetection`. Metoda odpowiadająca za taką obsługę kolizji jest przedstawiona na listingu 11.7.

Listing 11.7. *Przesuwanie obiektu wzdłuż innego obiektu (CollisionDetectionWithSliding.java)*

```
/**
    Obsługuje kolizję obiektu. Obiekt A jest obiektem poruszającym
    się, natomiast obiekt B jest tym obiektem, z którym obiekt A koliduje.
    Obiekt A przesuwa się wzdłuż krawędzi obiektu B lub (jeśli to
    możliwe) przekracza ten obiekt.
```

```
*/
protected boolean handleObjectCollision(GameObject objectA, GameObject objectB,
float distSq, float minDistSq, Vector3D oldLocation)
{
    objectA.notifyObjectCollision(objectB);

    if (objectA.isFlying()) {
        return true;
    }

    float stepSize = objectA.getBounds().getTopHeight() / 6;
    Vector3D velocity = objectA.getTransform().getVelocity();

    // Jeśli to możliwe, stań na obiekcie statycznym.
    float objectABottom = objectA.getY() + objectA.getBounds().getBottomHeight();
    float objectBTop = objectB.getY() + objectB.getBounds().getTopHeight();
    if (objectABottom + stepSize > objectBTop && objectBTop + objectA.getBounds().
    getTopHeight() < objectA.getCeilHeight())
    {
        objectA.getLocation().y = (objectBTop - objectA.getBounds().
        getBottomHeight());
        if (velocity.y < 0) {
            objectA.setJumping(false);
            // Uwzględnij działanie siły grawitacji.
            velocity.y = -.01f;
        }
        return false;
    }

    if (objectA.getX() != oldLocation.x || objectA.getZ() != oldLocation.z)
    {
        // Przesuwaj obiekt wzdłuż krawędzi obiektu statycznego.
        float slideDistFactor = (float)Math.sqrt(minDistSq / distSq) - 1;
        scratch.setTo(objectA.getX(), 0, objectA.getZ());
        scratch.subtract(objectB.getX(), 0, objectB.getZ());
        scratch.multiply(slideDistFactor);
        objectA.getLocation().add(scratch);

        // Jeśli obiekt koliduje ze ścianą, przywróć go do pierwotnego położenia.
        if (super.checkWalls(objectA, oldLocation, 0) != null)
        {
            return true;
        }

        return false;
    }

    return true;
}
```

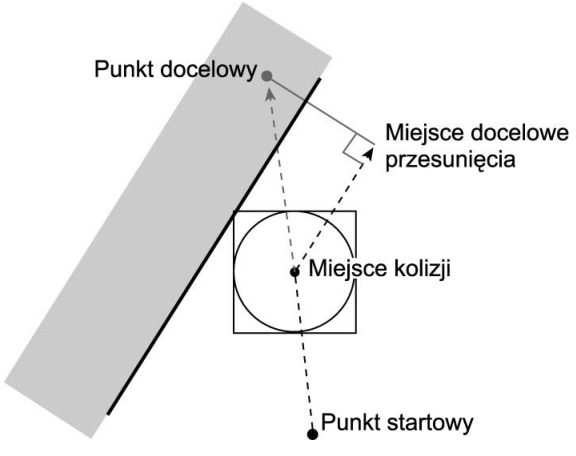
W powyższym kodzie przesłaniamy metodę `handleObjectCollision()`. Nowa metoda w pierwszej kolejności sprawdza, czy poruszający się obiekt nie może przejść nad obiektem statycznym. Jeśli nie, obiekt jest przesuwany wzdłuż krawędzi obiektu statycznego. Jeśli przesuwanie powoduje kolizję ze ścianą, obiekt jest przywracany na pierwotną pozycję.

Skoro mówimy o kolizjach ze ścianami, warto się zastanowić, jak zaimplementować poruszanie się obiektu wzdłuż ściany.

Przesuwanie obiektu wzdłuż ściany

Obsługa płynnego ruchu obiektu wzdłuż ściany jest z pozoru skomplikowanym zadaniem, jednak w rzeczywistości wymaga jedynie zastosowania kilku prostych obliczeń matematycznych. Jeśli znamy docelowe położenie obiektu (miejsce, do którego obiekt dotarłby, gdyby na jego drodze nie było ściany), możemy łatwo znaleźć miejsce, w którym obiekt powinien się znaleźć, poruszając się wzdłuż ściany (patrz rysunek 11.14).

Rysunek 11.14. *Przesuwanie obiektu wzdłuż ściany: obiekt porusza się wzdłuż ściany*



Na powyższym rysunku szarym odcinkiem oznaczyliśmy prostą prostopadłą do ściany. Jeśli znamy długość tej prostej, możemy łatwo obliczyć docelowe miejsce poruszającego się obiektu. Mamy do czynienia z prostym trójkątem prostokątnym. Rozważmy wektor prowadzący do miejsca kolizji od docelowego położenia obiektu:

```
vector.setTo(actualX, 0, actualZ);
vector.subtract(goalX, 0, goalZ);
```

Długość interesującego nas odcinka jest wówczas iloczynem skalarnym tego wektora i wektora prostopadłego do wielokąta pełniącego rolę ściany:

```
float length = vector.getDotProduct(wall.getNormal());
```

Możemy teraz obliczyć położenie punktu docelowego obiektu poruszającego się wzdłuż ściany

```
float slideX = goalX + length + wall.getNormal().x;
float slideZ = goalZ + length + wall.getNormal().z;
```

Tak wygląda rozwiązanie tego problemu. Cały kod (włącznie z dodatkowymi instrukcjami warunkowymi) obsługujący przesuwanie obiektu wzdłuż ściany znajduje się na listingu 11.8.

Listing 11.8. *Przesuwanie obiektu wzdłuż ściany (CollisionDetectionWithSliding.java)*

```
private Vector3D scratch = new Vector3D();
private Vector3D originalLocation = new Vector3D();

...

/**
    Sprawdza, czy obiekt w grze nie koliduje ze ścianami reprezentowanymi
    w drzewie BSP. Zwraca pierwszą ścianę, dla której występuje kolizja, lub
    wartość null, jeśli obiekt nie koliduje z żadną ścianą. Jeśli metoda wykryje
    kolizję, obiekt przemieszcza się wzdłuż ściany i sprawdza, czy nie
    występują kolejne kolizje. Jeśli w czasie ruchu wzdłuż ściany nastąpi
    nowa kolizja, obiekt jest przywracany na swoją pierwotną pozycję.
*/
public BSPPolygon checkWalls(GameObject object, Vector3D oldLocation, long elapsedTime)
{

    float goalX = object.getX();
    float goalZ = object.getZ();

    BSPPolygon wall = super.checkWalls(object, oldLocation, elapsedTime);
    // Jeśli wykryto kolizję i obiekt się sam nie zatrzymał:
    if (wall != null && object.getTransform().isMoving()) {
        float actualX = object.getX();
        float actualZ = object.getZ();

        // Iloczyn skalarny wektora prostopadłego do ściany i linii prowadzącej
           do celu.
        scratch.setTo(actualX, 0, actualZ);
        scratch.subtract(goalX, 0, goalZ);
        float length = scratch.getDotProduct(wall.getNormal());

        float slideX = goalX + length * wall.getNormal().x;
        float slideZ = goalZ + length * wall.getNormal().z;

        object.getLocation().setTo(slideX, object.getY(), slideZ);
        originalLocation.setTo(oldLocation);
        oldLocation.setTo(actualX, oldLocation.y, actualZ);

        // Użyj mniejszego promienia otoczenia podczas ruchu wzdłuż ściany.
        PolygonGroupBounds bounds = object.getBounds();
        float originalRadius = bounds.getRadius();
        bounds.setRadius(originalRadius-1);

        // Sprawdź występowanie kolizji w czasie ruchu wzdłuż ściany.
        BSPPolygon wall2 = super.checkWalls(object, oldLocation, elapsedTime);

        // Przywróć zmienione parametry.
        oldLocation.setTo(originalLocation);
        bounds.setRadius(originalRadius);

        if (wall2 != null) {
            object.getLocation().setTo(
                actualX, object.getY(), actualZ);
            return wall2;
        }
    }

    return wall;
}
```

W powyższym kodzie metoda `checkWalls()` przykrywa metodę zdefiniowaną w klasie `CollisionDetection`. Metoda odpowiada za obsługę ruchu wzdłuż ściany oraz sprawdzanie, czy po wykonaniu tego ruchu nie występują inne kolizje z dowolnymi innymi ścianami. Jeśli taka kolizja ma miejsce, obiekt jest cofany do punktu, w którym nastąpiła kolizja.

Oczywiście zawsze możemy wyłączyć mechanizm przesuwania obiektu wzdłuż ściany. Przykładowo metody obsługujące kolizje obiektów reprezentujących pociski powinny powodować natychmiastowe zatrzymywanie pocisku po uderzeniu w ścianę, podłogę lub sufit:

```
public void notifyWallCollision() {
    transform.getVelocity().setTo(0,0,0);
}

public void notifyFloorCollision() {
    transform.getVelocity().setTo(0,0,0);
}

public void notifyCeilingCollision() {
    transform.getVelocity().setTo(0,0,0);
}
```

Za pomocą powyższych metod uzyskujemy efekt „wbijania" się pocisków w ściany i inne obiekty, co oznacza, że możemy emulować omawianą wcześniej technikę obsługi kolizji.

Zaimplementowaliśmy już przemieszczanie się obiektów wzdłuż obiektów statycznych i ścian. Naszym następnym celem jest płynne poruszanie się obiektów na schodach i uwzględnienie siły grawitacji.

Grawitacja i płynny ruch na schodach (przesuwanie obiektu wzdłuż podłogi)

Kolejnym powszechnie stosowanym efektem jest umożliwienie graczowi i innym obiektom występującym w grze płynne poruszanie się na schodach. Bez odpowiedniej obsługi takiego ruchu gracz będzie w nienaturalny sposób skakał, chodząc po schodach (tak było w naszym pierwszym programie demonstrującym mechanizm wykrywania kolizji), ponieważ położenie obiektu w osi *y* będzie się z każdym kolejnym stopniem skokowo zmieniało.

Także kiedy gracz będzie schodził z wyższego poziomu na niższy, będzie momentalnie spadał na odpowiednią wysokość zamiast płynnie spadać na skutek działania grawitacji.

Wpływ siły grawitacji na obiekty w grze będziemy obsługiwali tak samo, jak w rozdziale 5., gdzie z czasem zwiększała się szybkość opadania obiektu. Siłę grawitacji możemy stosować dla obiektu, który znajduje się na większej wysokości niż znajdująca się pod nim podłoga.

Dokładnie odwrotnie jest w przypadku wchodzenia gracza po schodach: jeśli obiekt znajduje się na mniejszej wysokości niż podłoga pod nim, wówczas należy zastosować przyspieszenie w górę.

Obsługę wszystkich własności fizycznych związanych z ruchem obiektu umieściliśmy w klasie `Physics` (patrz listing 11.9). To oczywiście dopiero początek implementacji tych zjawisk — w przyszłości dodamy kolejne elementy.

Listing 11.9. *Grawitacja i wchodzenie po schodach (plik Physics.java)*

```
/**
    Domyślne przyspieszenie ziemskie w jednostkach na milisekundę kwadratową.
*/
public static final float DEFAULT_GRAVITY_ACCEL = -.002f;

/**
    Domyślne przyspieszenie dla ruchu w górę schodów w jednostkach na
    milisekundę kwadratową.
*/
public static final float DEFAULT_SCOOT_ACCEL = .006f;

private float gravityAccel;
private float scootAccel;
private Vector3D velocity = new Vector3D();

/**
    Stosuje przyspieszenie grawitacyjne dla określonego obiektu klasy
    GameObject według czasu, który minął od ostatniej aktualizacji.
*/
public void applyGravity(GameObject object, long elapsedTime) {
    velocity.setTo(0, gravityAccel * elapsedTime, 0);
    object.getTransform().addVelocity(velocity);
}


/**
    Stosuje przyspieszenie dla ruchu po schodach w górę dla określonego
    obiektu klasy GameObject według czasu, który minął od ostatniej
    aktualizacji.
*/
public void scootUp(GameObject object, long elapsedTime) {
    velocity.setTo(0, scootAccel * elapsedTime, 0);
    object.getTransform().addVelocity(velocity);
}
```

Znacznie bardziej skomplikowanym zadaniem jest uzyskanie wiedzy o tym, kiedy należy stosować zaimplementowane powyżej efekty w różnych sytuacjach. Przykładowo działania siły grawitacji nie powinniśmy uwzględniać w przypadku latających obiektów. Nie powinniśmy też obsługiwać płynnego wchodzenia po schodach w przypadku, gdy gracz znajduje się w powietrzu, lecz należy ten efekt stosować w przypadku podskakującego obiektu (do samego skakania przejdziemy za chwilę).

Wszystkie takie sytuacje obsługujemy w kodzie przedstawionym na listingu 11.10.

Listing 11.10. *Sprawdzanie podłogi i sufitu (CollisionDetectionWithSliding.java)*

```
/**
    Sprawdza, czy obiekt koliduje z podłogą i sufitem.
    Używa metod object.getFloorHeight() i object.getCeilHeight()
```

```
    do pozyskania wartości definiujących podłogę i sufit. Stosuje
    przyspieszenie ziemskie dla obiektu znajdującego się w
    powietrzu i płynne przyspieszenie w górę dla gracza
    znajdującego się na wysokości mniejszej niż wynosi wysokość
    podłogi (celem uzyskania efektu płynnego wchodzenia po schodach).
*/
protected void checkFloorAndCeiling(GameObject object, long elapsedTime)
{
    float floorHeight = object.getFloorHeight();
    float ceilHeight = object.getCeilHeight();
    float bottomHeight = object.getBounds().getBottomHeight();
    float topHeight = object.getBounds().getTopHeight();
    Vector3D v = object.getTransform().getVelocity();
    Physics physics = Physics.getInstance();

    // Sprawdza, czy obiekt stoi na podłodze.
    if (object.getY() + bottomHeight == floorHeight) {
        if (v.y < 0) {
            v.y = 0;
        }
    }
    // Sprawdza, czy obiekt nie znajduje się poniżej poziomu podłogi.
    else if (object.getY() + bottomHeight < floorHeight) {

        if (!object.isFlying()) {
            // Jeśli opada:
            if (v.y < 0) {
                object.notifyFloorCollision();
                v.y = 0;
                object.getLocation().y = floorHeight - bottomHeight;
            }
            else if (!object.isJumping()) {
                physics.scootUp(object, elapsedTime);
            }
        }
        else {
            object.notifyFloorCollision();
            v.y = 0;
            object.getLocation().y = floorHeight - bottomHeight;
        }
    }
    // Sprawdza, czy obiekt nie uderza w sufit.
    else if (object.getY() + topHeight > ceilHeight) {
        object.notifyCeilingCollision();
        if (v.y > 0) {
            v.y = 0;
        }
        object.getLocation().y = ceilHeight - topHeight;
        if (!object.isFlying()) {
            physics.applyGravity(object, elapsedTime);
        }
    }
    // Nad podłogą:
    else {
        if (!object.isFlying()) {
            // Zatrzymaj ruch po schodach w górę (jeśli obiekt jest w takim ruchu).
```

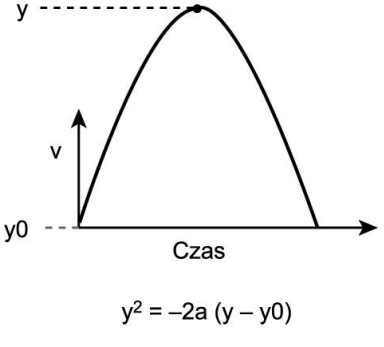
```
                if (v.y > 0 && !object.isJumping()) {
                    v.y = 0;
                    object.getLocation().y = floorHeight - bottomHeight;
                }
                else {
                    physics.applyGravity(object, elapsedTime);
                }
            }
        }
    }
```

Powyższy fragment kodu przykrywa zdefiniowaną w klasie `CollisionDetection` metodę `checkFloorAndCeiling()`. Pozostało nam już tylko zaimplementowanie skakania (głównie dla zabawy).

Skakanie

Skakanie powinno wyglądać bardzo podobnie, jak mechanizm zrealizowany w rozdziale 5. — należy po prostu właściwie obsłużyć wektor skierowany w górę. Początkowa wartość tego wektora zmniejsza się na skutek działania siły grawitacji.

W niektórych sytuacjach trzeba jednak określić, jak wysoko dany obiekt może skoczyć. Odpowiednie równanie prezentujemy na rysunku 11.15.



Rysunek 11.15. *Znajdowanie szybkości skoku dla określonej wysokości*

Na rysunku 11.15 szybkość skoku (v) można wyznaczyć na podstawie wysokości skoku ($y - y0$) oraz przyspieszenia ziemskiego (a). To standardowe równanie dla szybkości i przyspieszenia jest powszechnie stosowane w podręcznikach do fizyki.

Możemy teraz dodać do naszej klasy `Physics` kilka nowych metod implementujących skakanie (patrz listing 11.11).

Listing 11.11. *Skakanie (Physics.java)*

```
/**
    Ustawia dla danego obiektu GameObject wektor ruchu w górę, który
    umożliwi skok tego obiektu na określoną wysokość. Wywołuje metodę
    getJumpVelocity(), która oblicza szybkość ruchu za pomocą funkcji
    Math.sqrt().
```

```
*/
public void jumpToHeight(GameObject object, float jumpHeight) {
    jump(object, getJumpVelocity(jumpHeight));
}


/**
    Ustawia określoną szybkość ruchu pionowego obiektu GameObject dla
    określonej szybkości skoku.
*/
public void jump(GameObject object, float jumpVelocity) {
    velocity.setTo(0, jumpVelocity, 0);
    object.getTransform().getVelocity().y = 0;
    object.getTransform().addVelocity(velocity);
}


/**
    Zwraca początkową szybkość skoku potrzebną do osiągnięcia
    określonej wysokości (po uwzględnieniu bieżącej grawitacji).
    Wykorzystuje w obliczeniach funkcję Math.sqrt().
*/
public float getJumpVelocity(float jumpHeight) {
    // Użyj wzoru dla szybkości i przyspieszenia: v*v = -2 * a(y-y0)
    // (v jest szybkością skoku, a to przysp., y-y0 to maksymalna wys.)
    return (float)Math.sqrt(-2*gravityAccel*jumpHeight);
}
```

Powyższe metody umożliwiają nam zarówno znalezienie prędkości początkowej (w pionie) potrzebnej do osiągnięcia danej wysokości (w obliczeniach wykorzystujemy wartość przyspieszenia ziemskiego), jak i obsługę samego skoku.

Program demonstracyjny obsługujący kolizje z przesuwaniem

Zakończyliśmy omawianie zagadnień związanych z wykrywaniem kolizji. W kodach opracowanych dla tej książki znajduje się klasa `CollisionTestSliding`, która różni się od wcześniejszego programu demonstracyjnego jedynie implementacją obsługi ruchu wzdłuż innych obiektów. Oznacza to, że w jednym programie zawarliśmy obsługę ruchu gracza wzdłuż ścian, obsługę ruchu gracza wzdłuż innych obiektów, przekraczanie małych obiektów, wchodzenie na obiekty, płynne poruszanie się po schodach, stosowanie siły grawitacji i podskakiwanie.

Efektem ubocznym zaproponowanego rozwiązania jest możliwość stawania na pociskach. Ponieważ pociski wbijają się w ściany, możesz wystrzelić taką ich liczbę, by „narysować" z nich dodatkowy poziom przy samej ścianie. Następnie możesz wejść na ten nietypowy poziom!

Oczywiście jest to tylko jeden (zapewne nie najbardziej realistyczny) ze sposobów obsługi kolizji pocisków. Zwykle będziemy starali się zapewnić zniszczenie (nie wbijanie się) pocisków niezależnie od tego, w co trafiają.

Rozszerzenia

Opracowany przez nas mechanizm wykrywania i obsługi kolizji całkiem nieźle sprawdza się w praktyce, stworzone rozwiązanie nie jest jednak doskonałe i wymaga dalszych ulepszeń. Oto kilka pomysłów na przyszłe rozszerzenia:

- Implementacja drzew sfer, umożliwiających bardziej precyzyjne wykrywanie kolizji typu obiekt-obiekt.
- Wykonywanie dodatkowych testów pozwalających określić, czy obiekt nie przebył pomiędzy klatkami na tyle dużej odległości, by możliwe było wystąpienie kolizji pomiędzy punktem startowym a punktem docelowym.
- Umożliwienie graczom skradania i czołgania się, co pozwoli im na wchodzenie do niższych pomieszczeń.
- Implementacja wahań kamery oddających ruch głowy gracza podczas chodzenia i biegania.

Podsumowanie

Mechanizm wykrywania kolizji jest nieodłącznym elementem niemal każdej współczesnej gry (dwu- i trójwymiarowej).

Na początku tego rozdziału omówiliśmy zagadnienie izolowania obiektów za pomocą siatki i — tym samym — ograniczania liczby niezbędnych testów na występowanie kolizji. Następnie skupiliśmy się na różnych mechanizmach wykrywania kolizji, opartych na sferach otaczających, drzewach sfer, walcach otaczających i prostopadłościanach otaczających. Zaimplementowaliśmy algorytmy wykrywania kolizji zarówno z innymi obiektami, jak i ze ścianami, podłogami oraz sufitami reprezentowanymi w drzewie BSP. Zaimplementowaliśmy także lepszy mechanizm obsługi kolizji, który umożliwił graczom (i innym obiektom) poruszanie się wzdłuż krawędzi innych obiektów, ścian oraz po schodach. Na końcu, w celu zwiększenia realizmu gry, dodaliśmy wpływ siły grawitacji i możliwość skakania.

Program demonstracyjny umożliwia niszczenie robotów, co nie jest jednak zbyt trudnym zadaniem. Nad utrudnieniem gry będziemy pracować w kolejnym rozdziale: dodamy naszym przeciwnikom sztuczną inteligencję, dzięki czemu gra stanie się znacznie ciekawsza.