

Rozdział 7.

Grafika trójwymiarowa

W tym rozdziale:

- ◆ Typy renderowania grafiki trójwymiarowej.
- ◆ Nie zapominajmy o matematyce!
- ◆ Podstawy grafiki trójwymiarowej.
- ◆ Algebra trzech wymiarów.
- ◆ Wielokąty.
- ◆ Przekształcenia przestrzeni trójwymiarowej.
- ◆ Prosty potok tworzenia grafiki 3D.
- ◆ Ruch kamery.
- ◆ Bryły i usuwanie niewidocznych powierzchni.
- ◆ Rysowanie wielokątów za pomocą konwertera skanującego.
- ◆ Przycinanie w trzech wymiarach.
- ◆ Ostateczny potok renderowania.
- ◆ Podsumowanie.

Spotkałem kiedyś człowieka, który twierdził, że nigdy nie powstaną samochody latające w powietrzu.

Oczywiście nie zgodziłem się z nim i nadal uważam, że nie miał racji. Człowiek ten twierdził, że prawdziwym powodem, dla którego nigdy nie powstaną latające samochody, nie są problemy techniczne, ale ograniczenia ludzkiego umysłu. Większość ludzi nie będzie po prostu w stanie kontrolować pojazdu, który oprócz poruszania się do przodu, do tyłu oraz w prawo lub w lewo może jeszcze dodatkowo wznowić się lub obniżać. Nasz umysł jest po prostu w stanie kontrolować tylko ruch w dwóch wymiarach (wjeżdżanie samochodem na szczyt wzgórza lub jazda w dolinę się tutaj nie liczą, bo cały czas poruszamy się po powierzchni ziemi) i ruch w trzecim wymiarze będzie po prostu zbyt skomplikowany, aby taki samochód mógł się stać popularnym środkiem komunikacji.

To prawda, że sprawy związane z przestrzenią trójwymiarową są skomplikowane. To samo dotyczy grafiki trójwymiarowej. Mam jednak nadzieję, że informacje zawarte w tym rozdziale dostarczą Ci wystarczającej wiedzy na temat podstaw grafiki trójwymiarowej, abyś był w stanie tworzyć własne latające samochody. Przynajmniej w grze.

Ways renderowania grafiki trójwymiarowej

Najczęściej stosuje się dwa sposoby renderowania trójwymiarowej scenerii: śledzenie promieni światła i modelowanie wielokątów.

Technika *śledzenia promieni światła* (ang. *ray tracing*) naśladuje zjawiska fizyczne z rzeczywistego świata, przynajmniej w tym sensie, że modeluje bieg promieni światła — tyle że w odwrotną stronę. Zamiast modelować bieg promienia światła od źródła światła do oka wykonujemy odwrotną operację: badamy przebieg linii promieni światła od oka do światła. Jak łatwo sobie uświadomić, modelowanie promienia światła dla każdego piksela obrazu na ekranie wymaga wykonania zbyt wielu obliczeń naraz. Dlatego też technika śledzenia biegu promieni światła bardzo rzadko wykorzystywana jest w grafice trójwymiarowej działającej w czasie rzeczywistym. Niemniej technika ta pozwala na tworzenie bardzo realistycznych obrazów, które mieliśmy okazję oglądać na przykład w filmie *Epoka lodowcowa*.

Techniką renderowania trójwymiarowego, stosowaną w grach prezentujących trójwymiarowe obrazy zmieniające się w czasie rzeczywistym i w większości komputerowo generowanych obrazach trójwymiarowych, jest modelowanie wielokątów. W *modelowaniu wielokątów* (ang. *polygon modeling*) wirtualny świat trójwymiarowy interpretowany jest jako zestaw obiektów składających się z płaskich wielokątów. Pojedynczy wielokąt jest to płaska, ograniczona figura geometryczna o trzech lub więcej bokach (taka jak np. trójkąt, czworokąt czy ośmiokąt). Modelowanie wielokątów jest szybsze, bowiem wykonywane jest tutaj znacznie mniej obliczeń niż w przypadku korzystania z techniki śledzenia promieni światła; ponadto można jeszcze uzyskać znaczące przyspieszenie, korzystając z karty akceleratora 3D. Używanie akceleratora 3D leży w najlepszym interesie projektanta gier, ponieważ urządzenie te można obecnie znaleźć w prawie każdym komputerze.

W języku Java dostępne są dwa sposoby korzystania z możliwości karty akceleratora 3D: albo za pomocą interfejsu Java3D, albo za pomocą wiązania interfejsu OpenGL.

Java3D to graficzny interfejs API wysokiego poziomu, który obsługuje takie typowe zagadnienia związane z programowaniem grafiki, jak usuwanie niewidocznych powierzchni, proste wykrywanie kolizji czy zarządzanie kolejnymi scenami. Programiści korzystający z interfejsu Java3D są w stanie szybko przygotować grafikę gry bez konieczności zagłębiania się w szczegóły różnych technik programowania obrazów trójwymiarowych. Choć interfejs Java3D jest napisany głównie w języku Java, to jego jądro korzysta z interfejsu OpenGL lub interfejsu DirectX (interfejsy te renderują obrazy trójwymiarowe).

Interfejs Java3D dostępny jest dla systemu Windows i wielu odmian systemu Unix, nie jest jednak dołączany do zestawu Java 1.4.1 SDK. Ponadto, z powodów związanych z kosztami licencji, interfejs Java3D nie jest dostępny dla wszystkich systemów operacyjnych. Obecnie na przykład nie można z niego korzystać w systemie operacyjnym Mac OS X.

Z kolei wiązania interfejsu OpenGL można zazwyczaj wykorzystywać za darmo w komercyjnych programach; współdziałały one także z szerszą bazą sprzętową, w tym z Mac OS X. Interfejs OpenGL jest interfejsem graficznym niskiego poziomu, wymaga więc od programisty znacznie większej wiedzy na temat tworzenia trójwymiarowej grafiki. Wiązanie z interfejsem OpenGL umożliwia wykorzystywanie funkcji tego interfejsu w kodzie napisanym w języku Java.

Dwa z popularnych wiązań OpenGL to GL4Java (<http://www.jausoft.com/gl4java.html>) oraz LWJGL (<http://java-game-lib.sourceforge.net>). W momencie pisania tej książki wiązanie GL4Java było szerzej rozpowszechnione wśród różnych systemów operacyjnych i pozwalało na jednocześnie korzystanie z narzędzi takich, jak AWT czy Swing. Niemniej wiązanie to nie było aktualizowane od co najmniej roku, więc pewne najnowsze funkcje i możliwości interfejsu OpenGL mogą nie być w nim dostępne. Z kolei biblioteka LWJGL (ang. *Light-Weight Java Game Library* — prosta biblioteka tworzenia gier dla języka Java) podczas łączenia się z interfejsem OpenGL korzysta z buforów natywnych interfejsu NIO, tak więc działa odróżnionie szybciej. Biblioteka ta jest regularnie modyfikowana i aktualizowana, niemniej nie jest kompatybilna ani z AWT, ani ze Swing.

W tym oraz w następnym rozdziale (poświęconym mapowaniu tekstur i oświetleniu) zajmiemy się tworzeniem prostego mechanizmu renderowania, korzystając wyłącznie z technik programistycznych. Renderowanie to nie będzie tak szybkie, jakie byłoby renderowanie wykorzystujące techniki sprzętowe, dlatego też renderowanie programowe nie jest najlepszym rozwiązaniem w przypadku bardziej rozbudowanych gier. Niemniej przyjrzenie się procesowi programowania przygotowanemu za pomocą technik programistycznych pozwala lepiej zrozumieć podstawowe techniki grafiki 3D, a co za tym idzie, również zasady działania tych programów renderujących, które korzystają z technik sprzętowych. Przeważającą część prezentowanych tutaj metod renderowania można również wykorzystać w programach używających sprzętowych technik renderowania.

Ponadto renderowanie oparte na technikach programistycznych będzie idealnym rozwiązaniem w sytuacjach, gdy potrzebna nam będzie niewielka, możliwa do załadowania z sieci gra, w przypadku której od użytkownika nie będzie wymagane ładowanie czy instalowanie ani interfejsu Java3D, ani żadnego z wiązań OpenGL.

Nie zapominajmy o matematyce

Zanim przejdziemy do programowania grafiki 3D, powinniśmy zatrzymać się na moment i powiedzieć najpierw kilka słów o trygonometrii i wektorach. Czytelników uprasza się jednak o niezasypywanie — mówię poważnie, wiedza tu przypomniana będzie nam niezbędna. Tworzenie grafiki trójwymiarowej wymagać będzie częstego korzystania z wzorów trygonometrycznych i podstaw matematyki wektorów.

Trygonometria i trójkąty prostokątne

Większość z nas pamięta jeszcze zapewne ze szkoły definicje podstawowych pojęć trygonometrycznych, przedstawione na rysunku 7.1. Definicje te określają wzajemne relacje między kątami i długościami różnych boków trójkąta prostokątnego. Niektóre osoby znają te definicje na pamięć, dla wszystkich pozostałych przypominam je na rysunku 7.1.

Rysunek 7.1.

Definicje podstawowych funkcji trygonometrycznych



Większość problemów z zakresu grafiki trójwymiarowej można rozwiązać za pomocą trójkątów prostokątnych. Trójkąt prostokątny, jak sama nazwa wskazuje, jest to taki trójkąt, którego jeden z kątów jest kątem prostym, czyli ma 90 stopni. Aby „rozwiązać” trójkąt prostokątny lub, innymi słowy, poznać długość wszystkich jego boków i miary wszystkich jego kątów, wystarczy znać albo długość dwóch z jego boków, albo długość jednego boku i miarę jednego z dwóch kątów ostrych.

Wszystkim znane jest zapewne równanie $a^2 + b^2 = c^2$. W tym równaniu, zwanym twierdzeniem Pitagorasa, liczby a , b i c są długościami boków trójkąta prostokątnego. Przy czym długość przeciwprostokątnej, czyli boku położonego naprzeciw kąta prostego, oznaczana jest literą c . Jeśli więc znamy długości dwóch z trzech boków trójkąta prostokątnego i chcielibyśmy poznać długość trzeciego, wystarczy rozwiązać to równanie.

Jeśli jednak znamy tylko jeden bok i jeden z kątów ostrych, przydatne okazują się definicje trygonometryczne zestawione na rysunku 7.1. Definicje sinusa, cosinusa i tangensa określają relacje między bokami trójkąta prostokątnego a jego kątami i umożliwiają w ten sposób „rozwiązywanie” trójkąta prostokątnego.

Wektory

Podczas tworzenia grafiki trójwymiarowej pozycję punktu w przestrzeni określa się za pomocą trzech współrzędnych (x , y , z). Najważniejszą i może najtrudniejszą do zrozumienia sprawą jest fakt, że współrzędne te można traktować zarówno jako punkt w przestrzeni, jak i jako wektor. *Wektor* jest po prostu połączeniem kierunku i wielkości (tak jak w przypadku wektora prędkości) i podstawą całego rachunku wektorowego. Przykładowo może istnieć wektor prędkości skierowany na wschód o wartości 90 km/h. Jeśli kierunek wschodni wyznacza oś x , to wektor $(90, 0, 0)$ będzie interpretowany jako prędkość 90 km/h w kierunku wschodnim. Podobnie jeśli oś y wskazywać będzie na południe, to wektor $(0, 0, 90)$ będzie interpretowany jako prędkość 90 km/h skierowana na południe.

W tej książce zajmować się będziemy tylko wektorami trójwymiarowymi, niemniej należy mieć świadomość, że wektory występują nie tylko w przestrzeni trójwymiarowej. Wektory mogą mieć równie dobrze zarówno dwa wymiary, jak i więcej niż trzy. W tekście wektory oznaczać będziemy dużymi literami (w odróżnieniu od liczb, które oznaczamy małymi literami). Składowe komponenty wektora oznaczać będziemy za pomocą odpowiednich subskryptów:

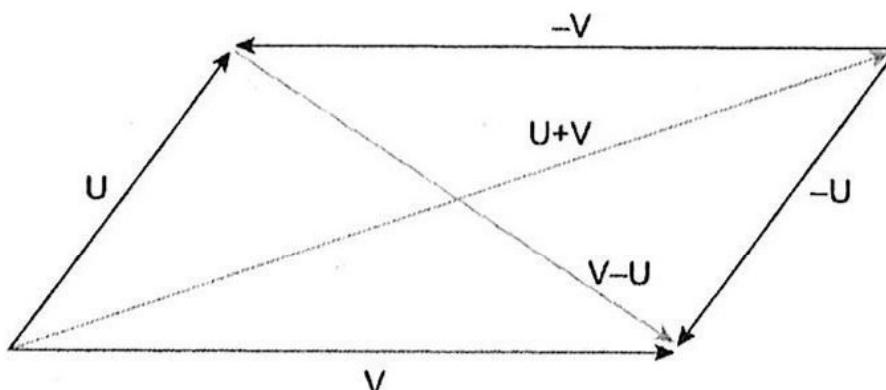
$$V = (V_x, V_y, V_z)$$

Należy pamiętać, że wektor to wielkość i kierunek, więc w jego przypadku nie istnieje żaden konkretny punkt zaczepienia — nie ma znaczenia, gdzie jest jego początek. Innymi słowy, podróż z prędkością 90 km/h z Los Angeles na wschód będzie z perspektywy wektorów dokładnie tym samym, co podróż z prędkością 90 km/h na wschód z Paryża (oczywiście z tą tylko różnicą, że w Stanach Zjednoczonych prędkości liczy się w milach na godzinę, ale mam nadzieję, że zrozumieлиście, o co mi chodzi). W obu przypadkach będzie to ten sam wektor.

Teraz zajmijmy się pewnymi elementami rachunku wektorowego. Najpierw proste dodawanie: wektory dodaje się do siebie tak, jak to zostało pokazane na rysunku 7.2. Na tym rysunku widzimy, jaki jest efekt dodania do siebie wektorów U i V (wektor $U + V$) oraz odjęcia wektora U od V (wektor $V - U$). Pokazane zostały również wektory $-U$ i $-V$: mają one te same wielkości, ale skierowane są w przeciwnym kierunku niż wektory U i V .

Rysunek 7.2.

Wektory można do siebie dodawać



Matematycznie dodawanie wektorów trójwymiarowych wykonuje się poprzez dodanie do siebie odpowiednich komponentów tych wektorów:

$$U + V = (U_x + V_x, U_y + V_y, U_z + V_z)$$

Dla przykładu, jeśli obserwujemy ptaka lecącego na wschód z prędkością 90 km/h, którego wiatr zwiewa na zachód z prędkością 15 km/h, to po zsumowaniu obu wektorów przekonamy się, że porusza się on na wschód z prędkością 75 km/h. Jak łatwo zauważycie, dodanie do siebie dwóch wektorów daje zupełnie nowy wektor z własną wielkością i kierunkiem. Oczywiście to samo dotyczy odejmowania wektorów. Kolejnym działaniem jest pomnożenie wektora przez skalar (czyli liczbę). Operacja ta zmieni wielkość wektora, nie zmieniając jednak jego kierunku — wykonuje się ją, mnożąc przez tę liczbę każdy z komponentów wektora:

$$s \cdot V = (s \cdot V_x, s \cdot V_y, s \cdot V_z)$$

Tak więc po pomnożeniu wektora prędkości 90 km/h skierowanego na wschód przez 2 otrzymamy również skierowany na wschód wektor prędkości 180 km/h. Tylko pamiętaj, aby nie próbować takich manewrów w domu.

Wielkość wektora lub też jego długość oznaczamy przez ujęcie go w pionowe kreski. Długość wektora wyliczamy, korzystając z trójwymiarowej wersji równania Pitagorasa:

$$|V| = (V_x^2 + V_y^2 + V_z^2)^{1/2}$$

Wektor o długości 1 nazywany jest wektorem jednostkowym lub wektorem znormalizowanym. Normalizację wektora wykonuje się, dzieląc go po prostu przez jego długość. Wektor znormalizowany oznaczamy przez umieszczenie ponad nim trójkątnego daszka (^):

$$\hat{U} = U / |U|$$

Wektory mogą się też kryć pod daszkiem, gdy zaczyna padać.

I to już właściwie koniec naszego krótkiego wprowadzenia do rachunku wektorowego. Będziemy ten temat zgłębiać jeszcze w dalszej części rozdziału. Niemniej to, co już wiemy, wystarczy nam w zupełności do przygotowania klasy dla wektorów, która przyda nam się w dalszej pracy. Klasa Vector3D, zaprezentowana na listingu 7.1, opisuje wektor za pomocą liczb zmiennoprzecinkowych. Większość obliczeń związanych z grafiką trójwymiarową wykonujemy właśnie na zwykłych liczbach zmiennoprzecinkowych (typ float), a nie na liczbach zmiennoprzecinkowych podwójnej precyzji (typ double), ponieważ obliczenia mniej wtedy obciążają komputer i są w większości przypadków wystarczająco precyzyjne.

g 7.1. Vector3D.java

```
package com.brackeen.javagamebook.math3D;

/**
 * Klasa Vector3D implementuje trójwymiarowy wektor, definiowany
 * przez liczby zmiennoprzecinkowe x, y oraz z. Taką reprezentację można
 * traktować jako punkt o współrzędnych (x,y,z) lub jako wektor wiodący
 * od punktu (0,0,0) do punktu (x,y,z).
 */
public class Vector3D implements Transformable {

    public float x;
    public float y;
    public float z;

    /**
     * Tworzy nowy Vector3D z (0,0,0).
     */
    public Vector3D() {
        this(0,0,0);
    }

}
```

```
/** Tworzy nowy Vector3D o tych samych wartościach, jakie
 * ustalono w innym wektorze Vector3D.
 */
public Vector3D(Vector3D v) {
    this(v.x, v.y, v.z);
}

/** Tworzy nowy Vector3D o podanych wartościach (x, y, z).
 */
public Vector3D(float x, float y, float z) {
    setTo(x, y, z);
}

/** Sprawdza, czy ten Vector3D jest równy podanemu obiektem (Object).
 * Będą równe tylko jeśli obiekt jest również wektorem typu Vector3D,
 * a współrzędne x, y oraz z obu wektorów są równe.
 */
public boolean equals(Object obj) {
    Vector3D v = (Vector3D)obj;
    return (v.x == x && v.y == y && v.z == z);
}

/** Sprawdza, czy dany Vector3D ma określone współrzędne
 * x, y oraz z.
 */
public boolean equals(float x, float y, float z) {
    return (this.x == x && this.y == y && this.z == z);
}

/** Przypisuje wektorowi takie same wartości, jakie ma
 * inny Vector3D.
 */
public void setTo(Vector3D v) {
    setTo(v.x, v.y, v.z);
}

/** Przypisuje temu wektorowi ustalone wartości (x, y, z).
 */
public void setTo(float x, float y, float z) {
    this.x = x;
    this.y = y;
    this.z = z;
}
```

```
/**  
 * Dodaje do tego wektora określone wartości(x, y, z).  
 */  
public void add(float x, float y, float z) {  
    this.x+=x;  
    this.y+=y;  
    this.z+=z;  
}  
  
/**  
 * Odejmuje od tego wektora określone wartości (x, y, z).  
 */  
public void subtract(float x, float y, float z) {  
    add(-x, -y, -z);  
}  
  
/**  
 * Dodaje do tego wektora inny, ustalony wektor.  
 */  
public void add(Vector3D v) {  
    add(v.x, v.y, v.z);  
}  
  
/**  
 * Odejmuje od tego wektora inny, ustalony wektor.  
 */  
public void subtract(Vector3D v) {  
    add(-v.x, -v.y, -v.z);  
}  
  
/**  
 * Mnoży ten wektor przez podaną wartość s. Nowa długość  
 * wektora będzie się równać length()*s.  
 */  
public void multiply(float s) {  
    x*=s;  
    y*=s;  
    z*=s;  
}  
  
/**  
 * Dzieli ten wektor przez określoną wartość s. Nowa długość  
 * wektora będzie się równać length()/s.  
 */  
public void divide(float s) {  
    x/=s;  
    y/=s;  
    z/=s;  
}
```

```

    /**
     * Zwraca długość wektora jako liczbę zmiennoprzecinkową typu float.
    */
    public float length() {
        return (float) Math.sqrt(x*x + y*y + z*z);
    }

    /**
     * Przekształca ten Vector3D na wektor jednostkowy lub,
     * innymi słowy, na wektor o długości 1. Równoważna wywołaniu
     * v.divide(v.length()).
    */
    public void normalize() {
        divide(length());
    }

    /**
     * Przekształca ten Vector3D na reprezentacjęłańcuchową.
    */
    public String toString() {
        return "(" + x + ", " + y + ", " + z + ")";
    }
}

```

Nie ma w tej klasie niczego trudnego czy bardzo skomplikowanego. Klasa Vector3D zawiera specjalne metody, pozwalające na dodawanie, odejmowanie wektorów, mnożenie i dzielenie wektorów przez skalar, wyliczanie ich długości oraz normalizowanie.

Klasy Vector3D będziemy używać zarówno w przypadku wektorów, jak i punktów w przestrzeni, ponieważ niektóre obliczenia matematyczne związane z grafiką trójwymiarową wymagają, by pewne wartości raz były traktowane jako punkty, a innym razem jako wektory. Na przykład możemy potrzebować wektora łączącego dwa punkty w przestrzeni. W tym przypadku, jeśli potraktujemy oba punkty jako wektory Vector3D, odjęcie jednego od drugiego da nam poszukiwany wektor łączący te punkty.

Jak już wcześniej mówiłem, to, co pokazałem do tej pory, to jeszcze nie cała matematyka związana z wektorami. Do klasy Vector3D trzeba będzie dodać jeszcze kilka metod. Zajmiemy się tym w dalszej części rozdziału. W tej chwili mamy już jednak podstawowe niezbędne narzędzia, by zająć się podstawami grafiki trójwymiarowej.

Podstawy grafiki trójwymiarowej

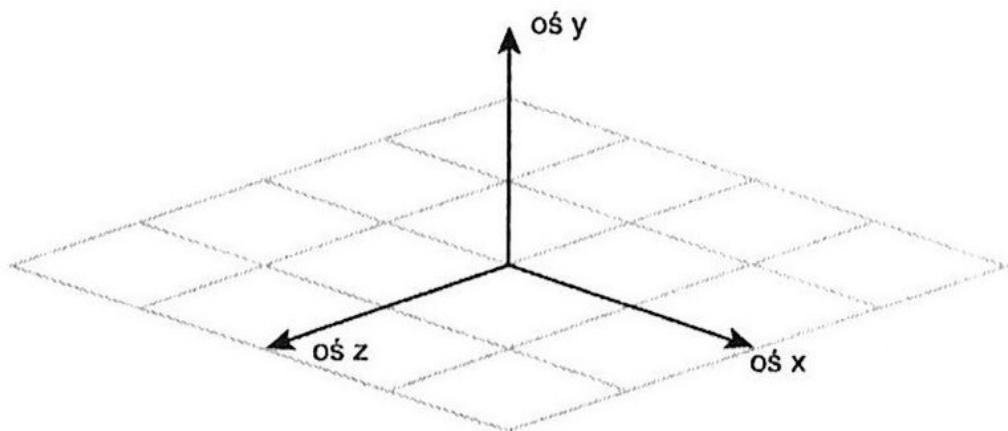
Co właściwie opisują wartości x , y oraz z w wektorze trójwymiarowym?

Podczas tworzenia grafiki dwuwymiarowej zazwyczaj korzysta się z dwuwymiarowego układu współrzędnych, przystającego do współrzędnych ekranowych: początek układu współrzędnych znajduje się w lewym górnym rogu i wartości x rosną z lewej do prawej,

a wartości y z góry do dołu. W grafice trójwymiarowej jednak trzy osie współrzędnych nie będą przekładać się wprost na współrzędne urządzenia. Tak więc zachodzi konieczność „przetłumaczenia” wyliczonego świata trójwymiarowego na dwuwymiarowy system współrzędnych ekranu. W tej książce stosować będziemy system współrzędnych trójwymiarowych zaczerpnięty z moich podręczników do matematyki, zwany „prawoskrętnym” układem współrzędnych, w którym osa x wskazuje „w prawo”, osa y wskazuje „w górę”, a osa z „w tył” (za plecy obserwatora). Jest to także system wykorzystywany przez interfejs OpenGL. Prawoskrętny system współrzędnych został przedstawiony na rysunku 7.3.

Rysunek 7.3.

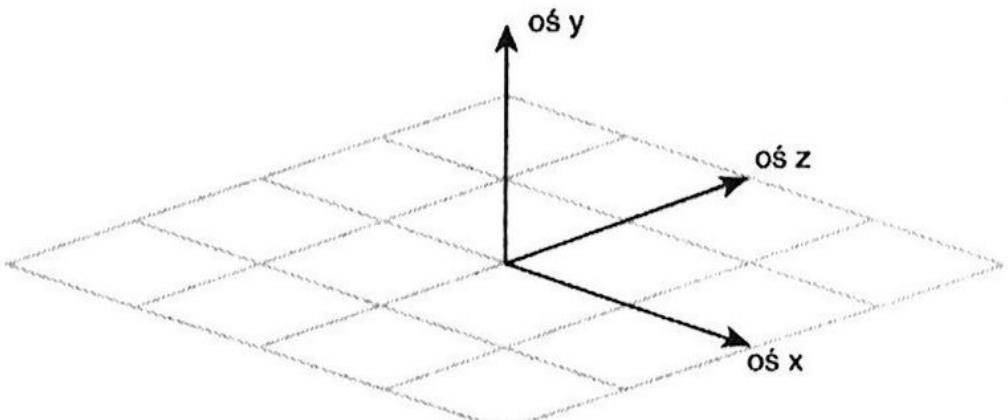
Prawoskrętny układ współrzędnych powszechnie stosuje się w grafice komputerowej i właśnie takiego układu współrzędnych będziemy używać w naszej książce



Jeśli w naszej prawej ręce skierujemy palec wskazujący zgodnie z kierunkiem osi y , a kciuk zgodnie z kierunkiem osi x , to środkowy palec będzie wskazywał osię z . Jeśli to samo zrobimy, używając lewej ręki, otrzymamy lewośkrętny układ współrzędnych, przedstawiony na rysunku 7.4.

Rysunek 7.4.

Lewoskrętny układ współrzędnych jest alternatywnym sposobem opisywania przestrzeni trójwymiarowej

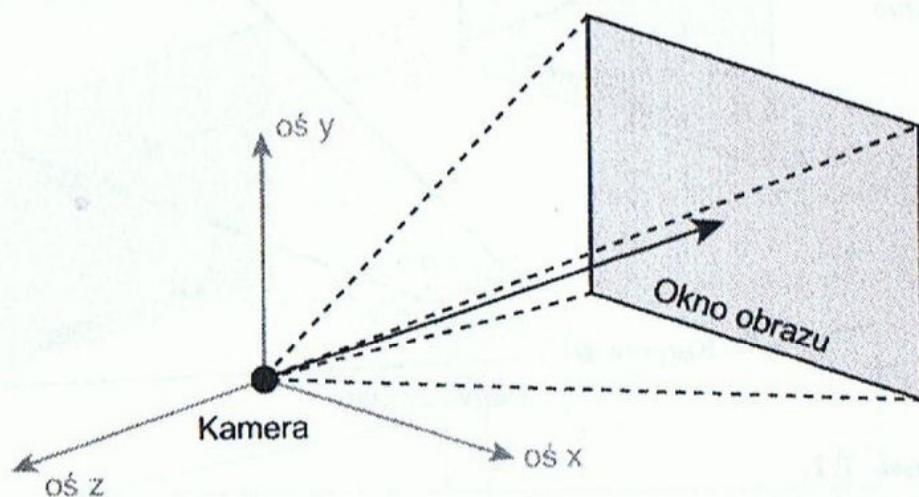


Inne trójwymiarowe układy współrzędnych są niczym więcej niż obróceniem prawoskrętnego układu współrzędnych w taki sposób, aby osa z wskazywała kierunek „do góry”. My jednak korzystać będziemy tylko i wyłącznie z prawoskrętnego układu współrzędnych, takiego jak zaprezentowany na rysunku 7.3. Teraz pora zapoznać się z pojęciami takimi, jak kamera i okno obrazu. Kamera jest miejscem, w którym zlokalizowany jest obserwator względem pozostałych elementów świata. Z kolei *okno obrazu* (ang. *view window*) jest to okno w trójwymiarowej przestrzeni, które ma taki sam rozmiar, jak okno, przez które oglądamy trójwymiarową scenę na ekranie. W grach komputerowych okno to ma zazwyczaj takie same wymiary, jak ekran.

Umieścimy kamerę w centrum układu współrzędnych $(0, 0, 0)$, a okno obrazu w kierunku odwrotnym niż wskazuje oś z , tak jak to zostało pokazane na rysunku 7.5. W ten sposób oś z będzie malała wraz ze wzrostem głębi. Jest to typowy sposób umiejscawiania okna obrazu.

Rysunek 7.5.

Kamera zlokalizowana jest w punkcie $(0, 0, 0)$ i skierowana jest w kierunku środka okna obrazu, znajdującego się w kierunku przeciwnym od wskazywanego przez oś z . Okno obrazu zajmuje zazwyczaj cały ekran



Umieszczenie kamery w punkcie $(0, 0, 0)$ i skierowanie jej w kierunku ujemnych wartości osi z może się wydawać dziwne. Można by przecież przesunąć kamerę albo zmienić jej kierunek, prawda? Moglibyśmy na przykład obrócić kamerę o 90 stopni w prawo, tak aby „patrzyła” w kierunku wskazywanym przez oś x . Dlaczego więc wybraliśmy właśnie takie ustawienie?

Odpowiedź brzmi: dla uproszczenia rachunków. Skierowanie kamery w kierunku wyznaczanym przez oś z i umieszczenie jej właśnie w punkcie $(0, 0, 0)$ bardzo upraszcza wiele obliczeń, stąd jest to typowa orientacja kamery stosowana w grafice trójwymiarowej. W dalszej części rozdziału opowiemy o sposobach symulowania ruchu kamery, na razie jednak skoncentrujmy się na rysowaniu trójwymiarowej grafiki dla nieruchomej kamery.

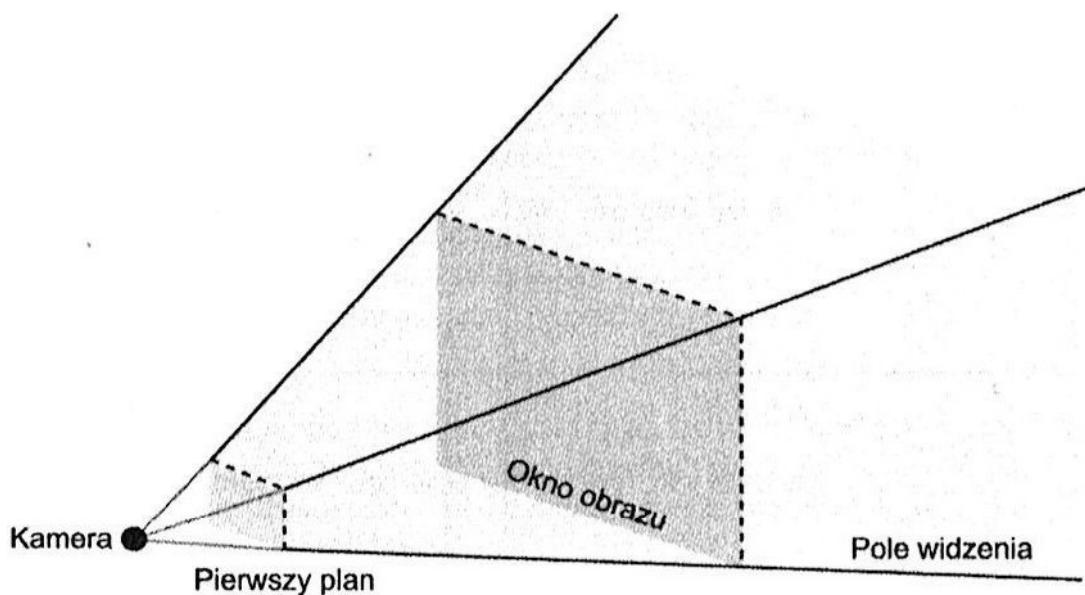
Warto zauważyć, że wszystko, co jest widoczne dla kamery, znajduje się w obrębie tego, co nazywamy *polem widzenia* (ang. *view frustum*, inaczej – ostrosłup widzenia), tak jak to zostało pokazane na rysunku 7.6. Pole widzenia ma zazwyczaj kształt czterobocznego ostrosłupa, którego wierzchołek jest czasami ścięty przez prostokąt pierwszego planu. Tworząc grafikę, należy narysować tylko to, co znajduje się w obrębie pola widzenia.

Zatrzymajmy się teraz na chwilę i zastanówmy się, co chcemy zrobić. Mamy trójwymiarowy świat i chcielibyśmy wyświetlić te obiekty z naszego świata, które znajdują się w polu widzenia, w formie dwuwymiarowego obrazu, na którym dalsze obiekty będą się wydawać mniejsze niż obiekty bliższe. W tym celu konieczne będzie rzutowanie trójwymiarowych obiektów na okno obrazu tak, jak to zostało pokazane na rysunku 7.7.

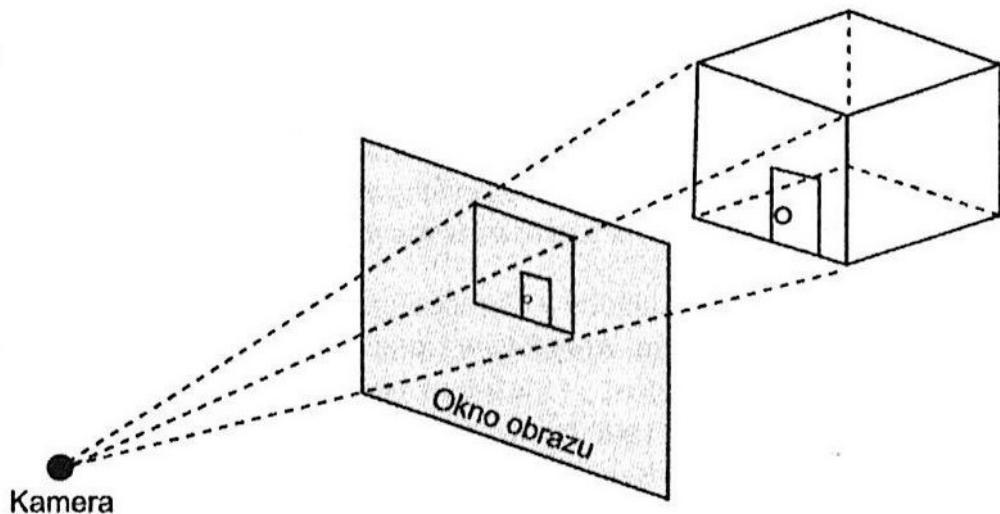
Rzutowanie obiektu trójwymiarowego na okno obrazu polega na wykreśleniu linii biegących od rysowanych punktów obrazu trójwymiarowego do kamery (przerywane linie na rysunku 7.7). Tam, gdzie linie te przetną się z oknem obrazu, rzutowany punkt będzie wyświetlany na ekranie. Ponieważ kamerę traktujemy jako punkt, obiekty bardziej

Rysunek 7.6.

Wszystkie widoczne elementy sceny znajdują się w obrębie pola widzenia

**Rysunek 7.7.**

Obiekty z trójwymiarowego świata są rzutowane na okno obrazu. Dzięki temu obiekty znajdujące się dalej będą wyglądać na mniejsze



oddalone od kamery będą po rzutowaniu mniejsze niż te znajdujące się bliżej. W ten sposób przechodzimy do kolejnego podrozdziału, który omawiać będzie wzory matematyczne wykorzystywane do rzutowania punktu na okno obrazu.

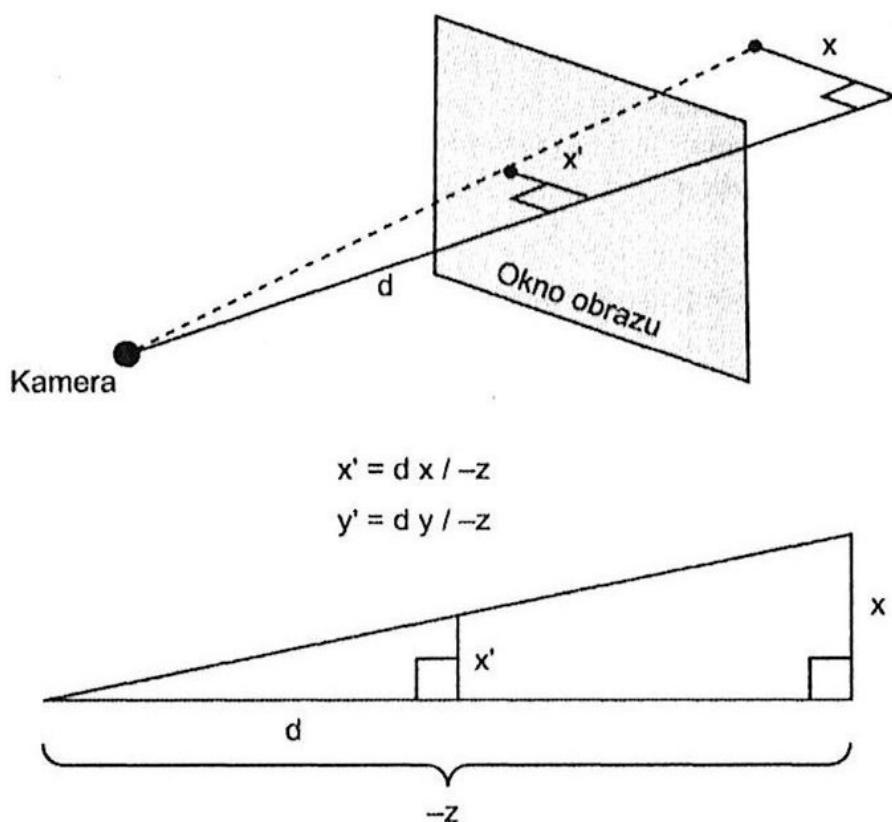
Algebra trzech wymiarów

Problem rzutowania punktu na okno widoku daje się w prosty sposób rozwiązać za pomocą trójkątów prostokątnych, tak jak to zostało przedstawione na rysunku 7.8.

Na rysunku 7.8 widzimy, jak w wyniku rzutowania trójwymiarowego punktu (x, y, z) na okno obrazu powstaje punkt dwuwymiarowy (x', y') . Należy pamiętać, że współrzędna z zmniejsza się w kierunku, w którym zwrócona jest kamera, tak więc odległość od punktu położenia kamery do rzutowanego punktu trójwymiarowego wynosi $-z$. W przykładzie przedstawionym na tym rysunku przeliczamy tylko współrzędną x , niemniej współrzędną y wylicza się dokładnie w ten sam sposób.

Rysunek 7.8.

Rzutowanie punktu (x, y, z)
na okno widoku



Trójkąt uformowany z linii x i $-z$ jest trójkątem podobnym do trójkąta utworzonego przez linie x' i d (wszystkie kąty obu trójkątów są takie same). Aby jednak rozwiązać ten problem, należy znać wartość d , czyli odległość od kamery do okna obrazu, którą nazywać będziemy odległością obrazu.

Dobrą wiadomością jest to, że możemy wybrać dowolną wartość d , czyli odległości obrazu, która w trakcie całej gry będzie zawsze taka sama. Odległość ta będzie zależeć od tego, jak szeroki kąt widzenia chcemy prezentować na ekranie. Dla przykładu, jeśli kamera będzie bardzo blisko okna obrazu (odległość obrazu będzie bardzo mała), to kąt widzenia kamery będzie bardzo szeroki i jej pole widzenia będzie obejmować znaczącą część naszego trójwymiarowego świata. I odwrotnie, jeśli kamera będzie dość daleko od okna obrazu (odległość obrazu będzie duża), to kamera będzie miała bardzo wąskie pole widzenia, ponieważ jej kąt widzenia będzie niewielki.

Kąt widzenia kamery zależy wyłącznie od nas. Zazwyczaj w grach kąt widzenia kamery w poziomie wybiera się pomiędzy wartościami 45 a 90 stopni. W prezentowanych w tej książce przykładach będziemy przyjmować kąt widzenia równy 75 stopniom.

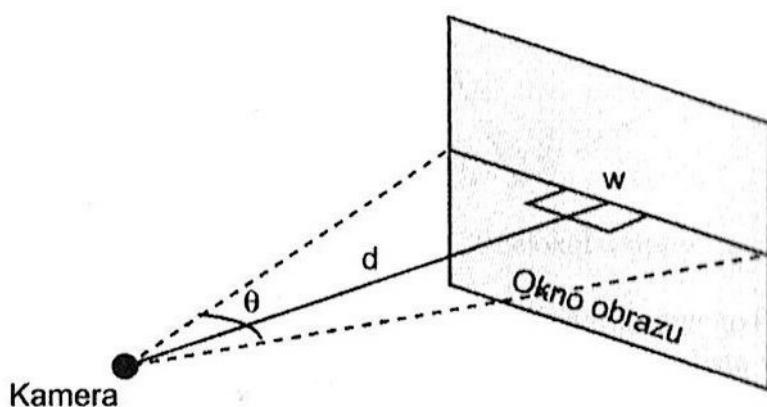
Dla poziomego kąta widzenia odległość obrazu wylicza się tak, jak na rysunku 7.9.

Warto zauważyć, że odległość obrazu zależy od rozmiarów okna. Jeśli pozwolimy graczowi zmieniać rozmiary okna na ekranie w trakcie gry, to konieczne będzie wtedy ponowne wyliczenie odległości obrazu.

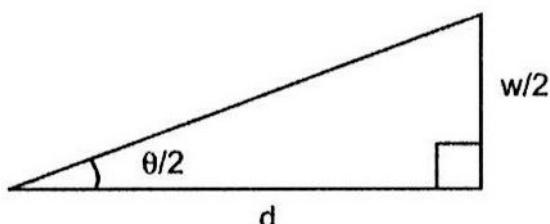
Gdy już znamy tę odległość, możemy wykonać rzutowanie punktu na okno obrazu, wykorzystując wzory zaprezentowane na rysunku 7.8. Niemniej należy pamiętać, że dwuwymiarowy system współrzędnych okna obrazu nie pokrywa się z systemem współrzędnych ekranu, co zostało pokazane na rysunku 7.10. Trzeba wykonać jeszcze jedno przekształcenie, by przekonwertować współrzędne okna obrazu na współrzędne ekranowe.

Rysunek 7.9.

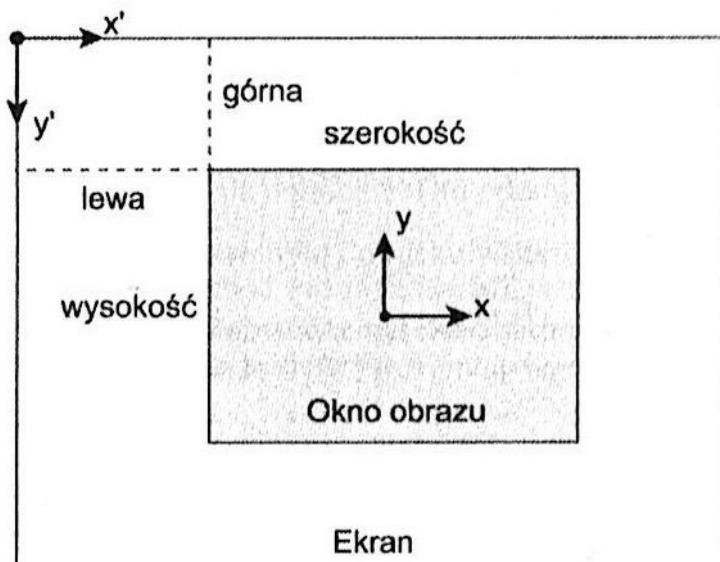
Wyliczanie odległości d między kamerą a oknem dla danego kąta pola widzenia θ i szerokości okna obrazu w



$$d = (w/2) / \tan(\theta/2)$$

**Rysunek 7.10.**

Przekształcanie współrzędnych punktu w oknie obrazu (x, y) na współrzędne punktu na ekranie (x', y')



$$\begin{aligned}x' &= \text{lewa} + \text{szerokość}/2 + x \\y' &= \text{górna} + \text{wysokość}/2 - y\end{aligned}$$

I to już cała matematyka. Przedstawione tu równania umożliwiają wyliczenie położenia obrazowanego punktu z przestrzeni trójwymiarowej na płaskim ekranie komputera. Okno obrazu i proces rzutowania umieściliśmy w klasie ViewWindow, przedstawionej na listingu 7.2.

Listing 7.2. Klasa ViewWindow.java

```
import java.awt.Rectangle;

/**
 * Klasa ViewWindow, reprezentująca okno widoku, na które
 * rzutowane są obiekty trójwymiarowe.
 */
```

```
/*
public class ViewWindow {

    private Rectangle bounds;
    private float angle;
    private float distanceToCamera;

    /**
     * Tworzy nowe okno ViewWindow o określonych granicach na ekranie
     * i ustalonym kącie widzenia w poziomie.
     */
    public ViewWindow(int left, int top, int width, int height,
                      float angle)
    {
        bounds = new Rectangle();
        this.angle = angle;
        setBounds(left, top, width, height);
    }

    /**
     * Definiuje granice tego okna ViewWindow na ekranie.
     */
    public void setBounds(int left, int top, int width,
                          int height)
    {
        bounds.x = left;
        bounds.y = top;
        bounds.width = width;
        bounds.height = height;
        distanceToCamera = (bounds.width/2) /
            (float)Math.tan(angle/2);
    }

    /**
     * Definiuje poziomy kąt widzenia dla tego okna ViewWindow.
     */
    public void setAngle(float angle) {
        this.angle = angle;
        distanceToCamera = (bounds.width/2) /
            (float)Math.tan(angle/2);
    }

    /**
     * Pobiera kąt widzenia w poziomie dla tego okna obrazu.
     */
    public float getAngle() {
        return angle;
    }

    /**
     * Pobiera szerokość tego okna obrazu.
     */
    public int getWidth() {
        return bounds.width;
    }
}
```

```
/***
 * Pobiera wysokość tego okna obrazu.
 */
public int getHeight() {
    return bounds.height;
}

/***
 * Pobiera przesunięcie y tego okna widoku względem ekranu.
 */
public int getTopOffset() {
    return bounds.y;
}

/***
 * Pobiera przesunięcie x tego okna widoku względem ekranu.
 */
public int getLeftOffset() {
    return bounds.x;
}

/***
 * Pobiera odległość od kamery do tego okna obrazu.
 */
public float getDistance() {
    return distanceToCamera;
}

/***
 * Przekształca współrzędną x z tego okna obrazu
 * na odpowiadającą jej współrzędną x na ekranie.
 */
public float convertFromViewXToScreenX(float x) {
    return x + bounds.x + bounds.width/2;
}

/***
 * Przekształca współrzędną y z tego okna obrazu
 * na odpowiadającą jej współrzędną y na ekranie.
 */
public float convertFromViewYToScreenY(float y) {
    return -y + bounds.y + bounds.height/2;
}

/***
 * Przekształca współrzędną x z ekranu na odpowiadającą
 * jej współrzędną x w oknie obrazu.
 */
public float convertFromScreenXToViewX(float x) {
    return x - bounds.x - bounds.width/2;
}
```

```

    /**
     * Przekształca współrzędną y z ekranu na odpowiadającą
     * jej współrzędną y w oknie obrazu.
    */
    public float convertFromScreenYToViewY(float y) {
        return -y + bounds.y + bounds.height/2;
    }

    /**
     * Rzutuje podany wektor na ekran.
    */
    public void project(Vector3D v) {
        // rzutuj na okno obrazu
        v.x = distanceToCamera * v.x / -v.z;
        v.y = distanceToCamera * v.y / -v.z;

        // konwertuj na współrzędne ekranowe
        v.x = convertFromViewXToScreenX(v.x);
        v.y = convertFromViewYToScreenY(v.y);
    }
}

```

W klasie ViewWindow rzutowanie wykonywane jest wewnątrz metody `project()`, która wykonuje rzutowanie wektora `Vector3D` na okno obrazu `ViewWindow`, a następnie tłumaczy współrzędne okna obrazu na współrzędne ekranu. Po wywołaniu metody `project()` przekształcone komponenty `x` i `y` wektora `Vector3D` staną się odpowiednimi współrzędnymi ekranowymi, a komponent `z` można będzie zupełnie zignorować.

Hura! Wiemy, jak rzutować punkty z trójwymiarowej przestrzeni na ekran komputera! Zaraz, zaraz, to jeszcze nie wszystko. Wiemy, jak rzutować punkty, ale potrzebujemy przecież jakiegoś sposobu modelowania wielokątów. Pora więc teraz zająć się wielokątami.

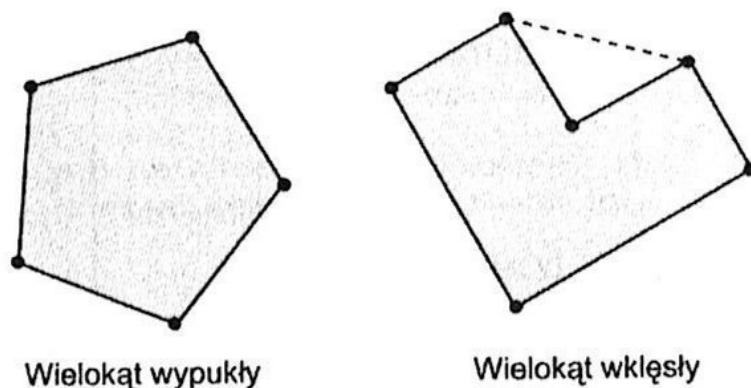
Wielokąty

Wypełnione wielokąty można zasadniczo podzielić na dwie kategorie: wielokąty wypukłe i wielokąty wklęsłe. Pokazane zostało to na rysunku 7.11. Wielokąty wklęsłe mają na obwodzie zagłębienie, jakby niszę. W przykładach przedstawionych w tym rozdziale będziemy zajmować się wyłącznie wielokątami wypukłymi, ponieważ z wielokątami wklęsłymi wiąże się konieczność rozpatrywania pewnych sytuacji szczególnych, przez co praca z nimi jest znacznie trudniejsza. Oczywiście każdy wielokąt wklęsły można rozbić na kilka wielokątów wypukłych, tak więc nie będziemy mieli żadnych problemów z symulowaniem wielokątów wklęsłych w naszych grach.

Jeśli przyjrzeć się rysunkowi 7.11, widać od razu, że wielokąt można bardzo łatwo opisać matematycznie. Wielokąty są po prostu zbiorem punktów lub, inaczej mówiąc, wierzchołków. Wielokąty będą wypełnione dopiero wówczas, gdy narysujemy je na ekranie.

Rysunek 7.11.

Dwa typy wielokątów:
wypukłe i wklęsłe.
Wielokąt jest wklęsły,
gdy jego obwód zagina
się do środka



Dlatego też zaczniemy od przygotowania klasy wielokąta trójwymiarowego Polygon3D, która została przedstawiona na listingu 7.3. Klasa ta po prostu zarządza listą wierzchołków wielokąta.

Listing 7.3. Klasa Polygon3D.java

```
package com.brackeen.javagamebook.math3D;

/**
 * Klasa Polygon3D przedstawia wielokąt jako zbiór
 * wierzchołków.
 */
public class Polygon3D implements Transformable {

    // Tymczasowe wektory wykorzystywane do obliczeń.
    private static Vector3D temp1 = new Vector3D();
    private static Vector3D temp2 = new Vector3D();

    private Vector3D[] v;
    private int numVertices;
    private Vector3D normal;

    /**
     * Tworzy pusty wielokąt, który można wykorzystywać jako
     * wielokąt „roboczy” dla potrzeb transformacji, rzutowania itd.
     */
    public Polygon3D() {
        numVertices = 0;
        v = new Vector3D[0];
        normal = new Vector3D();
    }

    /**
     * Tworzy nowy wielokąt Polygon3D o podanych wierzchołkach.
     */
    public Polygon3D(Vector3D v0, Vector3D v1, Vector3D v2) {
        this(new Vector3D[] { v0, v1, v2 });
    }

    /**
     * Tworzy nowy wielokąt Polygon3D o podanych wierzchołkach. Zakładamy,
     * że wszystkie wierzchołki położone są w tej samej płaszczyźnie.
     */
}
```

```
/*
public Polygon3D(Vector3D v0, Vector3D v1, Vector3D v2,
    Vector3D v3)
{
    this(new Vector3D[] { v0, v1, v2, v3 });
}

/** Tworzy nowy wielokąt Polygon3D o podanych wierzchołkach. Zakładamy,
 * że wszystkie wierzchołki położone są w tej samej płaszczyźnie.
 */
public Polygon3D(Vector3D[] vertices) {
    this.v = vertices;
    numVertices = vertices.length;
    calcNormal();
}

/** Przypisuje temu wielokątowi takie same wierzchołki, jakie ma
 * podany wielokąt.
 */
public void setTo(Polygon3D polygon) {
    numVertices = polygon.numVertices;
    normal.setTo(polygon.normal);

    ensureCapacity(numVertices);
    for (int i=0; i<numVertices; i++) {
        v[i].setTo(polygon.v[i]);
    }
}

/** Upewnia się, że wielokąt ma wystarczającą pojemność,
 * żeby przechowywać dane o wszystkich swoich wierzchołkach.
 */
protected void ensureCapacity(int length) {
    if (v.length < length) {
        Vector3D[] newV = new Vector3D[length];
        System.arraycopy(v, 0, newV, 0, v.length);
        for (int i=v.length; i<newV.length; i++) {
            newV[i] = new Vector3D();
        }
        v = newV;
    }
}

/** Pobiera liczbę wierzchołków, które ma ten wielokąt.
 */
public int getNumVertices() {
    return numVertices;
}
```

```

    /**
     * Pobiera wierzchołek o podanym indeksie.
    */
    public Vector3D getVertex(int index) {
        return v[index];
    }

    /**
     * Rzutuje ten wielokąt na okno obrazu.
    */
    public void project(ViewWindow view) {
        for (int i=0; i<numVertices; i++) {
            view.project(v[i]);
        }
    }
}

```

Klasa `Polygon3D` korzysta z tablicy wektorów `Vector3D`, by za jej pomocą reprezentować wielokąt. Podobnie jak klasa `Vector3D`, przedstawiona tu klasa `Polygon3D` jest tylko wyjściowym szkieletem wielokąta — będziemy ją jeszcze rozwijać w dalszej części tego rozdziału.

Domyślny konstruktor, niewymagający przekazania żadnych argumentów, tworzy prosty wielokąt, który nie posiada żadnych wierzchołków. Wielokąt utworzony za pomocą tego konstruktora można wykorzystywać w charakterze wielokąta „roboczego”. Wielokąt roboczy przydawać się będzie wszędzie tam, gdzie potrzebna nam będzie kopia wielokąta, którą chcielibyśmy zmodyfikować, by nie była konieczna zmiana oryginalnego wielokąta. Aby przypisać wielokątowi roboczemu dane z wielokąta, który chcielibyśmy w ten sposób modyfikować, wystarczy wywołać metodę `setTo()`.

Jak łatwo zauważyc, klasa `Polygon3D` opisuje kształt wielokąta, nic natomiast nie wiadomo o jego kolorze czy tekście. Teksturami wielokątów zajmiemy się w następnym rozdziale, a na razie będziemy rysować tylko wielokąty wypełnione jednolitym kolorem. Bez trudu można przygotować klasę `SolidPolygon3D`, będącą podklassą klasy `Polygon3D`, która będzie zawierała dodatkowe metody umożliwiające ustawianie lub pobieranie koloru wielokąta.

W tym momencie moglibyśmy już przygotować prosty przykład grafiki trójwymiarowej, jednak opowiemy o transformacjach w przestrzeni trójwymiarowej, które pozwolą nam uatrakcyjnić odrobinę nasz przykładowy program.

Przekształcenia przestrzeni trójwymiarowej

Aby móc poruszać wielokąty w przestrzeni trójwymiarowej (lub też wykonywać inne triki graficzne), należy zastosować na nich odpowiednie przekształcenie przestrzeni trójwymiarowej. Trzy najbardziej typowe przekształcenia wykonywane na trójwymiarowych wielokątach to translacja, skalowanie i rotacja (obracanie).

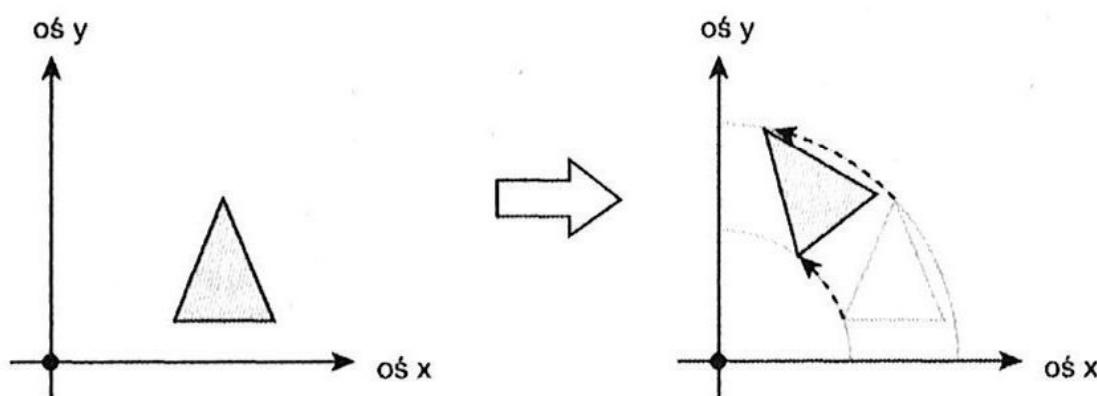
- ♦ Translację lub, mówiąc prościej, przesuwanie wielokąta w przestrzeni wykonuje się, dodając po prostu do każdego wierzchołka wielokąta wektor translacji.
- ♦ Skalowanie lub inaczej powiększanie, ewentualnie zmniejszanie wielokąta (przy zachowaniu jego kształtu) jest dość proste, jeśli środek wielokąta znajduje się w początku układu współrzędnych. W takiej sytuacji wielokąt skaluje się, po prostu mnożąc każdy z jego wierzchołków przez współczynnik skalowania. Gdyby jednak środek wielokąta nie znajdował się w początku układu współrzędnych, to w ten sposób nie tylko skalowalibyśmy wielokąt, ale również przesunelibyśmy go w pewnym kierunku. Ponieważ nie będziemy potrzebować skalowania w naszym mechanizmie grafiki trójwymiarowej, pominiemy to przekształcenie.
- ♦ Rotacja polega na obróceniu wielokąta wokół osi x , osi y , osi z lub dwóch albo wszystkich trzech osi naraz.

Rotacje

Najpierw przyjrzyjmy się równaniom matematycznym związanym z rotacją — dokładniej obrotowi dookoła osi z . Gdy obracamy punkt wokół osi z to jego współrzędna z nie ulega zmianie. Obrót taki wygląda więc tak samo, jak obrót wokół punktu $(0, 0)$ w przestrzeni dwuwymiarowej, co zostało pokazane na rysunku 7.12. Każdy z punktów wielokąta obraca się wokół punktu $(0, 0)$, więc zmienia się tylko jego pozycja, a nie odległość względem punktu $(0, 0)$.

Rysunek 7.12.

Obrót wokół osi z
jest niczym więcej
niż obrotem w dwóch
wymiarach wokół
punktu $(0, 0)$



Jeśli potraktujemy x i y jako wartości wyrażające długość promienia r , a kąt obrotu wokół osi z nazwiemy φ , to współrzędne x i y punktu, który chcemy obrócić, będą spełniały następujące równania:

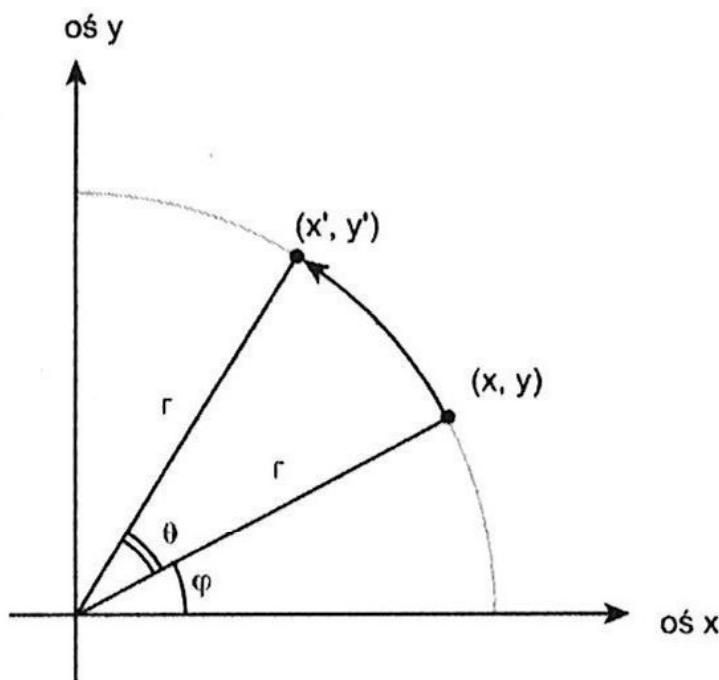
$$\begin{aligned}x &= r \cos \varphi \\y &= r \sin \varphi\end{aligned}$$

Jeśli wykonamy obrót w kierunku odwrotnym do ruchu wskazówek zegara o kąt θ (co zostało pokazane na rysunku 7.13), to nowe wartości współrzędnych x' i y' będą wyrażane następującymi wzorami:

$$\begin{aligned}x' &= r \cos (\varphi + \theta) \\y' &= r \sin (\varphi + \theta)\end{aligned}$$

Rysunek 7.13.

Obracanie punktu (x, y) o kąt θ . Kąt oryginalnego punktu (x, y) to φ , a kąt punktu (x', y') po obrocie to $(\varphi + \theta)$



Równania te możemy przekształcić do następującej postaci:

$$\begin{aligned}x' &= r \cos \varphi \cos \theta - r \sin \varphi \sin \theta \\y' &= r \sin \varphi \cos \theta + r \cos \varphi \sin \theta\end{aligned}$$

Po podstawieniu x za $r \cos \varphi$, a y za $r \sin \varphi$ otrzymamy:

$$\begin{aligned}x' &= x \cos \theta - y \sin \theta \\y' &= x \sin \theta + y \cos \theta\end{aligned}$$

I to już cała matematyka. Otrzymane wzory umożliwiają nam obracanie punktu dookoła osi z w kierunku przeciwnym do ruchu wskazówek zegara o kąt θ .

Obracanie dookoła osi x i osi y wylicza się podobnie. Oto odpowiednie wzory:

- ♦ Aby obrócić punkt w kierunku odwrotnym do ruchu wskazówek zegara dookoła osi x :

$$\begin{aligned}\♦ \quad x' &= x \\ \♦ \quad y' &= y \cos \theta - z \sin \theta \\ \♦ \quad z' &= y \sin \theta + z \cos \theta\end{aligned}$$

- ♦ Aby obrócić punkt w kierunku odwrotnym do ruchu wskazówek zegara dookoła osi y :

$$\begin{aligned}\♦ \quad x' &= z \sin \theta + x \cos \theta \\ \♦ \quad y' &= y \\ \♦ \quad z' &= z \cos \theta - x \sin \theta\end{aligned}$$

- ♦ Oraz dla przypomnienia, w kierunku odwrotnym do ruchu wskazówek zegara dookoła osi z :
 - ♦ $x' = x \cos \theta - y \sin \theta$
 - ♦ $y' = x \sin \theta + y \cos \theta$
 - ♦ $z' = z$

W przypadku bardziej złożonej rotacji konieczne może być wyliczenie trzech obrotów: dla osi x , y i z . Aby obrócić wielokąt, należy po prostu zastosować odpowiednie wzory na każdym z punktów tego wielokąta.

Hermetyzacja przekształceń rotacji i translacji

Mimo iż zaprezentowane tutaj wzory na wykonanie rotacji wyglądają prosto, należy pamiętać, że matematyczne funkcje `Math.sin()` i `Math.cos()` są pewnym obciążeniem dla komputera. Implementuje się je za pomocą wielu obliczeń i wymagają one stosowania odwołań do funkcji interfejsu JNI, tak więc wywoływanie tych funkcji dla każdego wektora `Vector3D`, który chcielibyśmy przekształcić, nie jest zbyt wydajną techniką. Zamiast tego lepiej jest wyliczyć i zapamiętać wartość sinusa i cosinusa dla kąta obrotu, a potem w przekształceniach konkretnych wektorów używać już tych wyliczonych wcześniej wartości. Ponieważ przekształcenia wykonuje się zazwyczaj jednokrotnie w jednej klatce, zredukujemy w ten sposób znacznie liczbę wywołań kosztownych funkcji `Math.sin()` i `Math.cos()`.

Klasa `Transform3D`, zaprezentowana na listingu 7.4, implementuje przedstawione tutaj przekształcenia. Zawiera wszelkie informacje niezbędne do wykonywania translacji i rotacji, sama jednak nie wykonuje tych przekształceń. Później, gdy będziemy chcieli wykonywać za jej pomocą przekształcenia, będziemy mogli skorzystać z wyliczonych wcześniej wartości sinusa i cosinusa, by obracać wszystkie niezbędne wektory `Vector3D`.

Listing 7.4. *Transform3D.java*

```
package com.brackeen.javagamebook.math3D;

/**
 * Klasa Transform3D reprezentuje przekształcenia rotacji i translacji.
 */
public class Transform3D {

    protected Vector3D location;
    private float cosAngleX;
    private float sinAngleX;
    private float cosAngleY;
    private float sinAngleY;
    private float cosAngleZ;
    private float sinAngleZ;

    /**
     * Tworzy nową transformację Transform3D bez określania, czy to
     * translacja, czy też rotacja.
     */
```

```
/*
public Transform3D() {
    this(0.0,0);
}

/** Tworzy nową transformację Transform3D bez określania konkretnej translacji lub rotacji.
*/
public Transform3D(float x, float y, float z) {
    location = new Vector3D(x, y, z);
    setAngle(0.0,0);
}

/** Tworzy nową transformację Transform3D.
*/
public Transform3D(Transform3D v) {
    location = new Vector3D();
    setTo(v);
}

public Object clone() {
    return new Transform3D(this);
}

/** Przypisuje tej transformacji Transform3D inną transformację Transform3D.
*/
public void setTo(Transform3D v) {
    location.setTo(v.location);
    this.cosAngleX = v.cosAngleX;
    this.sinAngleX = v.sinAngleX;
    this.cosAngleY = v.cosAngleY;
    this.sinAngleY = v.sinAngleY;
    this.cosAngleZ = v.cosAngleZ;
    this.sinAngleZ = v.sinAngleZ;
}

/** Pobiera lokację (translację) z tej transformacji.
*/
public Vector3D getLocation() {
    return location;
}

public float getCosAngleX() {
    return cosAngleX;
}

public float getSinAngleX() {
    return sinAngleX;
}
```

```
public float getCosAngleY() {
    return cosAngleY;
}

public float getSinAngleY() {
    return sinAngleY;
}

public float getCosAngleZ() {
    return cosAngleZ;
}

public float getSinAngleZ() {
    return sinAngleZ;
}

public float getAngleX() {
    return (float)Math.atan2(sinAngleX, cosAngleX);
}

public float getAngleY() {
    return (float)Math.atan2(sinAngleY, cosAngleY);
}

public float getAngleZ() {
    return (float)Math.atan2(sinAngleZ, cosAngleZ);
}

public void setAngleX(float angleX) {
    cosAngleX = (float)Math.cos(angleX);
    sinAngleX = (float)Math.sin(angleX);
}

public void setAngleY(float angleY) {
    cosAngleY = (float)Math.cos(angleY);
    sinAngleY = (float)Math.sin(angleY);
}

public void setAngleZ(float angleZ) {
    cosAngleZ = (float)Math.cos(angleZ);
    sinAngleZ = (float)Math.sin(angleZ);
}

public void setAngle(float angleX, float angleY, float angleZ)
{
    setAngleX(angleX);
    setAngleY(angleY);
    setAngleZ(angleZ);
}

public void rotateAngleX(float angle) {
    if (angle != 0) {
        setAngleX(getAngleX() + angle);
    }
}
```

```

public void rotateAngleY(float angle) {
    if (angle != 0) {
        setAngleY(getAngleY() + angle);
    }
}

public void rotateAngleZ(float angle) {
    if (angle != 0) {
        setAngleZ(getAngleZ() + angle);
    }
}

public void rotateAngle(float angleX, float angleY,
    float angleZ)
{
    rotateAngleX(angleX);
    rotateAngleY(angleY);
    rotateAngleZ(angleZ);
}

```

Stosowanie transformacji

Same transformacje (przekształcenia) są wykonywane dopiero w specjalnych metodach, które dodamy do klasy wektora Vector3D. Prezentujemy je na listingu 7.5.

Listing 7.5. Metody wykonujące transformacje w klasie Vector3D.java

```

/**
 * Obróć ten wektor naokoło osi x o określony kąt.
 * Kąt podawany jest w radianach. Funkcja Math.toRadians()
 * pozwala na konwersję ze stopni na radiany.
 */
public void rotateX(float angle) {
    rotateX((float)Math.cos(angle), (float)Math.sin(angle));
}

/**
 * Obróć ten wektor naokoło osi y o określony kąt.
 * Kąt podawany jest w radianach. Funkcja Math.toRadians()
 * pozwala na konwersję ze stopni na radiany.
 */
public void rotateY(float angle) {
    rotateY((float)Math.cos(angle), (float)Math.sin(angle));
}

/**
 * Obróć ten wektor naokoło osi z o określony kąt.
 * Kąt podawany jest w radianach. Funkcja Math.toRadians()
 * pozwala na konwersję ze stopni na radiany.
 */

```

```
/*
public void rotateZ(float angle) {
    rotateZ((float)Math.cos(angle), (float)Math.sin(angle));
}

/***
Obróć ten wektor dookoła osi x o określony kąt,
wykorzystując już wyliczone wartości cosinusa i sinusa kąta,
o który obracamy wektor.
*/
public void rotateX(float cosAngle, float sinAngle) {
    float newY = y*cosAngle - z*sinAngle;
    float newZ = y*sinAngle + z*cosAngle;
    y = newY;
    z = newZ;
}

/***
Obróć ten wektor dookoła osi y o określony kąt,
wykorzystując już wyliczone wartości cosinusa i sinusa kąta,
o który obracamy wektor.
*/
public void rotateY(float cosAngle, float sinAngle) {
    float newX = z*sinAngle + x*cosAngle;
    float newZ = z*cosAngle - x*sinAngle;
    x = newX;
    z = newZ;
}

/***
Obróć ten wektor dookoła osi x o określony kąt,
wykorzystując już wyliczone wartości cosinusa i sinusa kąta,
o który obracamy wektor.
*/
public void rotateZ(float cosAngle, float sinAngle) {
    float newX = x*cosAngle - y*sinAngle;
    float newY = x*sinAngle + y*cosAngle;
    x = newX;
    y = newY;
}

/***
Dodaje do tego wektora określoną transformację. Wektor
jest najpierw obracany, a dopiero potem wykonywana jest translacja.
*/
public void add(Transform3D xform) {

    // obróć
    addRotation(xform);

    // wykonaj translację
    add(xform.getLocation());
}
```

```

    /**
     * Odejmuje do tego wektora określoną transformację. Na wektorze
     * najpierw wykonywana jest translacja, a dopiero potem jest on obracany.
     */
    public void subtract(Transform3D xform) {
        // wykonaj translację
        subtract(xform.getLocation());

        // obróć
        subtractRotation(xform);
    }

    /**
     * Obraca wektor o kąt podany
     * w transformacji.
     */
    public void addRotation(Transform3D xform) {
        rotateX(xform.getCosAngleX(), xform.getSinAngleX());
        rotateZ(xform.getCosAngleZ(), xform.getSinAngleZ());
        rotateY(xform.getCosAngleY(), xform.getSinAngleY());
    }

    /**
     * Obraca wektor o kąt przeciwny do podanego
     * w transformacji.
     */
    public void subtractRotation(Transform3D xform) {
        // Zauważcie, że  $\sin(-x) == -\sin(x)$  i  $\cos(-x) == \cos(x)$ .
        rotateY(xform.getCosAngleY(), -xform.getSinAngleY());
        rotateZ(xform.getCosAngleZ(), -xform.getSinAngleZ());
        rotateX(xform.getCosAngleX(), -xform.getSinAngleX());
    }
}

```

W zaprezentowanym tu kodzie, jak łatwo zauważyc, znajdują się metody pozwalające zarówno „dodać”, jak i „odjąć” transformację. Przy dodawaniu transformacji stosowana jest po prostu transformacja na wektorze. Z kolei odejmowanie transformacji polega na zastosowaniu na wektorze odwrotności transformacji: zamiast obracania wektora o kąt θ i dodawania lokacji V wektor jest obracany o kąt $-\theta$, a następnie odejmowana jest od niego lokacja V . Metody odejmujące transformację służą do symulowania swobodnego ruchu kamery, o czym opowiemy w dalszej części tego rozdziału.

Metody dodawania `add(Transform3D)` i odejmowania transformacji `subtract (Transform3D)` zostały również dołączone do klasy wielokąta `Polygon3D`. Tam metody te przekształcają po prostu każdy z wierzchołków wielokąta.

Porządek rotacji

Ponieważ rotację złożoną wykonujemy, obracając punkt po kolej dookoła każdej z osi, niezmiernie ważne jest ustalenie, w jakiej kolejności będą wykonywane obroty dookoła kolejnych osi. Jeśli obroty dookoła różnych osi zostaną wykonane w innej kolejności,

otrzymamy inny wynik końcowy. W naszym kodzie podczas dodawania transformacji rotacje wykonywane są w kolejności: wokół osi x , potem z i wreszcie y . Odejmowanie transformacji wymaga z kolei odwrotnego porządku rotacji (y , z i x). Nasz wybór jest całkowicie arbitralny i dostarcza przewidywalnych wyników w przypadku gier polegających na przedstawianiu scenerii 3D z punktu widzenia wędrującego po niej bohatera (tak, jak w grze *Doom*).

Prosty potok tworzenia grafiki 3D

W tym momencie możemy już naszkicować prosty potok tworzenia grafiki 3D, podsumowujący kolejne operacje, które będą konieczne, by narysować wielokąt:

1. Zastosuj transformację
2. Rzutuj ją na okno obrazu.
3. Rysuj.

Ten potok został zaimplementowany w klasie `Simple3DTest1`, zaprezentowanej na listingu 7.6. Klasa ta po kolei: wykonuje transformacje, rzutuje i rysuje dwa wielokąty tworzące drzewo.

Listing 7.6. *Simple3DTest1.java*

```
import com.brackeen.javagamebook.test.GameCore;

import java.awt.*;
import java.awt.event.KeyEvent;
import java.awt.geom.GeneralPath;

import com.brackeen.javagamebook.input.*;
import com.brackeen.javagamebook.math3D.*;

/**
 * Test grafiki 3D, ilustrujący rysowanie wielokątów.
 */
public class Simple3DTest1 extends GameCore {

    public static void main(String[] args) {
        new Simple3DTest1().run();
    }

    // Utwórz wielokąty wypełnione jednolitym kolorem.
    private SolidPolygon3D treeLeaves = new SolidPolygon3D(
        new Vector3D(-50, -35, 0),
        new Vector3D(50, -35, 0),
        new Vector3D(0, 150, 0));

    private SolidPolygon3D treeTrunk = new SolidPolygon3D(
        new Vector3D(-5, -50, 0),
        new Vector3D(5, -50, 0),
        new Vector3D(5, -35, 0),
        new Vector3D(-5, -35, 0));
}
```

```
private Transform3D treeTransform = new Transform3D(0.0,-500);
private Polygon3D transformedPolygon = new Polygon3D();
private ViewWindow viewWindow;

private GameAction exit = new GameAction("exit");
private GameAction zoomIn = new GameAction("zoomIn");
private GameAction zoomOut = new GameAction("zoomOut");

public void init() {
    super.init();

    InputManager inputManager = new InputManager(
        screen.getFullScreenWindow());
    inputManager.setCursor(InputManager.INVISIBLE_CURSOR);
    inputManager.mapToKey(exit, KeyEvent.VK_ESCAPE);
    inputManager.mapToKey(zoomIn, KeyEvent.VK_UP);
    inputManager.mapToKey(zoomOut, KeyEvent.VK_DOWN);

    // Niech okno obrazu zajmuje cały ekran.
    viewWindow = new ViewWindow(0, 0,
        screen.getWidth(), screen.getHeight(),
        (float)Math.toRadians(75));

    // Wypełnij wielokąty kolorami.
    treeLeaves.setColor(new Color(0x008000));
    treeTrunk.setColor(new Color(0x714311));
}

public void update(long elapsedTime) {
    if (exit.isPressed()) {
        stop();
        return;
    }

    // Przechwyć czas elapsedTime.
    elapsedTime = Math.min(elapsedTime, 100);

    // Obróć naokoło osi y.
    treeTransform.rotateAngleY(0.002f*elapsedTime);

    // Pozwól użytkownikowi powiększać i zmniejszać obraz.
    if (zoomIn.isPressed()) {
        treeTransform.getLocation().z += 0.5f*elapsedTime;
    }
    if (zoomOut.isPressed()) {
        treeTransform.getLocation().z -= 0.5f*elapsedTime;
    }
}

public void draw(Graphics2D g) {
    // Usuń tło.
    g.setColor(Color.black);
    g.fillRect(0, 0, screen.getWidth(), screen.getHeight());
```

```
// Wyświetl komunikat.  
g.setColor(Color.white);  
g.drawString("Klawisze up/down to regulacja powiększenia. Esc zamyka program.",  
5, fontSize);  
  
// Narysuj wielokąty drzewa.  
transfromAndDraw(g, treeTrunk);  
transfromAndDraw(g, treeLeaves);  
}  
  
/**  
 * Rzutuje i rysuje wielokąt na oknie widoku.  
 */  
private void transfromAndDraw(Graphics2D g,  
SolidPolygon3D poly)  
{  
    transformedPolygon.setTo(poly);  
  
    // Wykonaj translację i obrót wielokąta.  
    transformedPolygon.add(treeTransform);  
  
    // Rzutuj wielokąt na ekran.  
    transformedPolygon.project(viewWindow);  
  
    // Konwertuj wielokąt na Java2D GeneralPath i narysuj go.  
    GeneralPath path = new GeneralPath();  
    Vector3D v = transformedPolygon.getVertex(0);  
    path.moveTo(v.x, v.y);  
    for (int i=1; i<transformedPolygon.getNumVertices(); i++) {  
        v = transformedPolygon.getVertex(i);  
        path.lineTo(v.x, v.y);  
    }  
    g.setColor(poly.getColor());  
    g.fill(path);  
}
```

Program Simple3DTest1 rysuje drzewo cały czas obracające się dookoła osi y. Użytkownik może przybliżyć drzewo lub oddalić je od kamery, wciskając klawisze *up/down*. Efekt działania programu został pokazany na rysunku 7.14.

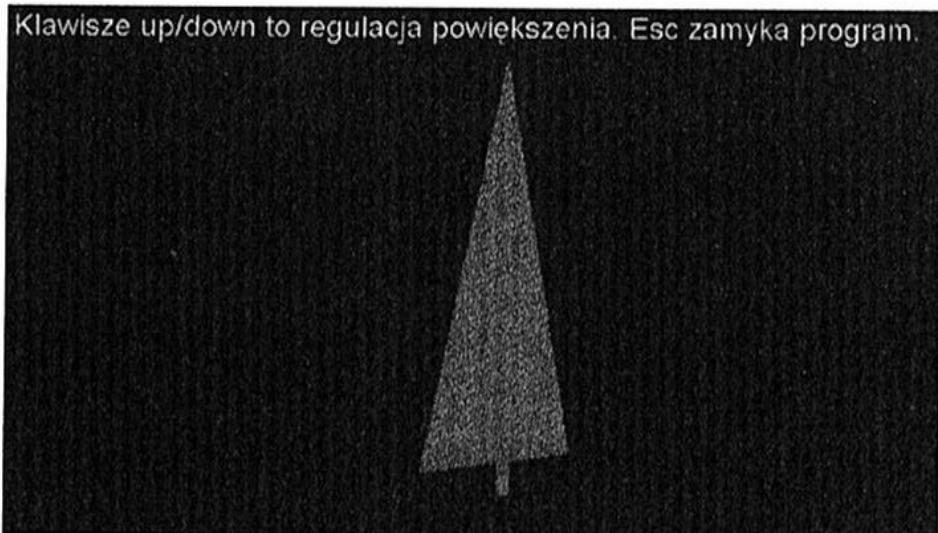
W programie Simple3DRest1 zamiast klasy wielokąta trójwymiarowego `Polygon3D` używamy jej podklasy `SolidPolygon3D`.

Wielokąty trójwymiarowe rysowane są za pomocą metody `transfromAndDraw()`. Metoda ta zajmuje się transformowaniem i rysowaniem wielokąta.

Aby narysować wielokąt, należy najpierw przekonwertować go na obiekt `GeneralPath` (część pakietu `java.awt.geom`). Interfejs `Graphics2D` dostarcza metod umożliwiających rysowanie obiektów `GeneralPath`, tak więc wielokąt zostanie narysowany automatycznie.

Rysunek 7.14.

Ekrany programu
Simple3DTest1



Warto zauważyć, że rysowane tu wielokąty drzewa nie są tak naprawdę transformowane. Są po prostu kopiowane do wielokąta roboczego i dopiero on jest transformowany. Gdybyśmy przekształcali oryginalne wielokąty tworzące drzewo, z czasem zapewne „rozjechałyby się” one z powodu stopniowego kumulowania się błędów pojawiających się podczas zaokrąglania liczb zmiennoprzecinkowych. Jedno przekształcenie nie powoduje zauważalnych błędów, niemniej po wielu transformacjach błędy mogą znacznie zdeformować kształt wielokąta. Dlatego należy skopiować wielokąt do wielokąta roboczego i wykonywać transformację na wielokącie roboczym.

W programie Simple3DTest1 pojawia się jednak parę problemów. Być może dostrzegłeś niektóre z nich:

- ◆ Kamera jest umieszczona na stałe w punkcie $(0, 0, 0)$. Mimo iż mamy możliwość przybliżania i oddalania drzewa przesuwanego w osi z , lepiej by było, gdyby dało się przesuwać kamerę swobodnie w różne strony.
- ◆ Gdy drzewo jest bardzo blisko kamery, wielokąty rysowane są w bardzo dziwny sposób. Ponadto, jeśli przesadzimy z przybliżeniem drzewa i kamera znajdzie się po drugiej stronie drzewa, to drzewo zostanie narysowane do góry nogami! Dzieje się tak dlatego, że wielokąty nie są przycinane, aby mieściły się w polu widzenia.
- ◆ Tego prawie nie widać, ale czasami wielokąty nie stykają się dokładnie ze sobą — można czasami dostrzec między nimi szczeliny. Dzieje się tak z dwóch powodów: po pierwsze, interfejs Graphics2D domyślnie zapisuje w obiekcie GeneralPath wierzchołki, których współrzędne są zaokrąglane do najbliższej liczby całkowitej, zamiast zachowywać współrzędne wierzchołków wyrażone w liczbach zmiennoprzecinkowych. Po drugie, dwa stykające się ze sobą wielokąty tworzą rozgałęzienia, bowiem nie posiadają wspólnych wierzchołków. Na razie nie będziemy się zajmować problemem rozgałęzień. Naprawimy tę niedoróbkę w rozdziale 10., w podrozdziale poświęconym zarządzaniu sceną za pomocą drzew BSP.
- ◆ Na koniec nasze drzewo jest płaskie, przez co nie wygląda zbyt ładnie. Drzewa utworzone z brył i inne kształty bryłowe są znacznie ciekawsze.

Zajmijmy się teraz usuwaniem tych niedoróbek! Na początek: swobodny ruch kamery.

Ruch kamery

W programie Simple3DTest1 wykonywaliśmy przekształcenia trzech wielokątów, aby uzyskać efekt przybliżania bądź oddalania drzewa od kamery. Szczerze powiedziawszy, nie wiadomo, co tu się tak naprawdę porusza: kamera czy drzewo?

Oczywiście nie ma dla nas znaczenia, który obiekt tak naprawdę się porusza, bo z punktu widzenia obserwatora liczy się tylko subiektywnie postrzegana przez niego odległość od drzewa.

Zamiast więc zmieniać wykorzystywane techniki programistyczne, by umożliwić kamierze poruszanie się po całym świecie, należy skorzystać z obiektu transformacji Transform3D, który reprezentuje położenie kamery i zastosować na każdym wielokącie tworzącym scenę przekształcenie odwrotne do przechowywanego w tym obiekcie (odając przekształcenie). Jak widać, poruszamy raczej cały świat, a nie przesuwamy kamery. W ten sposób kamera zawsze znajdować się będzie w punkcie (0, 0, 0), co znacznie uprości wszystkie obliczenia, a my uzyskamy wrażenie, że kamera porusza się tam, gdzie chcemy.

Dla przykładu powiedzmy, że wirtualna kamera znajduje się w punkcie (10, 50, 500) i „spogląda” pod kątem 5 stopni w lewo. Aby zastosować przekształcenie symulujące takie ustawienie kamery, należy po prostu przesunąć każdy z wielokątów sceny o (-10, -50, -500) i obrócić je o 5 stopni w prawo. Innymi słowy, trzeba po prostu wywołać dla każdego wielokąta metodę subtract (Transform3D).

Obrót naokoło każdej z osi dostarcza następujących efektów symulujących ruch kamery w scenerii 3D:

- ◆ Obrót naokoło osi x daje wrażenie spoglądania w góre lub w dół.
- ◆ Obrót naokoło osi y daje wrażenie obracania głowy w prawo lub w lewo.
- ◆ Obrót naokoło osi z daje wrażenie przechylania głowy na boki.

W kodzie obsługującym kamerę trzeba będzie zapewne ograniczyć możliwości spoglądania w górę lub w dół. Podobnie, mechanizm przechylania kamery na boki powinien służyć raczej tylko celom demonstracyjnym. W grze komputerowej zazwyczaj nie ma potrzeby udostępniania graczom możliwości przechylania głowy bohatera na boki.

Bryły i usuwanie niewidocznych powierzchni

W naszym pierwszym przykładzie — SimpleDemo3DTest1 — wielokąty są rysowane w takiej kolejności, w jakiej występują na liście wielokątów. W tym przykładzie taka uproszczona technika nie spowoduje żadnych nieprzyjemnych efektów ubocznych, ponieważ drzewo jest płaskie, a wielokąty nie nakładają się na siebie.

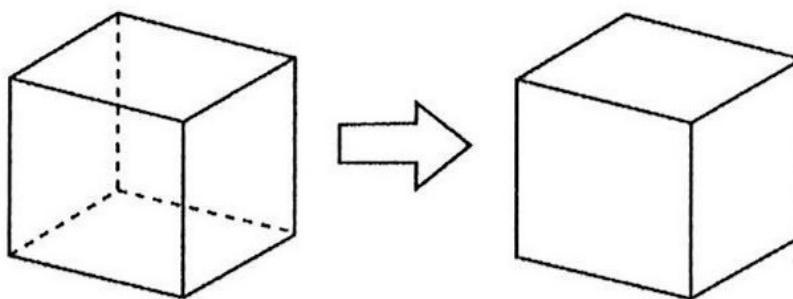
Jeśli natomiast zamierzamy narysować bryłę — czyli grupę wielokątów posiadających przód, tył i boki — to potrzebny nam będzie lepszy algorytm ustalania, które wielokąty powinny być rysowane i w jakim porządku. W przeciwnym bowiem razie wielokąty, które znajdują się z tyłu, mogłyby się pojawiać przed wielokątami, które są bliżej. Nie trzeba mówić, że takie efekty zniszczą całą scenę trójwymiarową.

Porządkowanie widocznych wielokątów omówione zostanie w kolejnych rozdziałach, jednakże najpierw jednak pomówmy o algorytmach zajmujących się usuwaniem zakrytych powierzchni. Opierają się one na pomyśle, by nie rysować wielokątów, które znajdują się za innymi wielokątami lub z jakichś innych powodów nie są widoczne dla kamery.

Pierwsza technika usuwania niewidocznych powierzchni, z której będziemy korzystać, nazywana jest *usuwaniem tylnych powierzchni* (ang. *backface removal*) lub czasami likwidowaniem tylnych powierzchni. Rysunek 7.15 pokazuje podstawową zasadę działania tej techniki: rysujemy tylko te wielokąty, które ustawione są przodem do kamery.

Rysunek 7.15.

Usuwanie tylnych powierzchni. Zamiast wszystkich sześciu ścian rysowanego sześciokąta rysowane są tylko trzy ściany, skierowane przodem do kamery



Usuwanie tylnych powierzchni jest bardzo sprytnym trikiem i, statystycznie rzecz biorąc, zmniejsza o połowę liczbę wielokątów, które trzeba narysować. Ponadto technika ta perfekcyjnie i bez żadnego dodatkowego wysiłku z naszej strony rysuje wszystkie wieloboki wypukłe. Wielobok wypukły to taka trójwymiarowa bryła, która nie ma żadnych niszczy ani zagłębień.

Niemniej technika usuwania tylnych powierzchni nie sprawdza się w przypadku wszystkich bez wyjątku wielokątów. Jeśli na przykład mamy wklęsły wielobok lub kilka brył, potrzebny nam będzie kolejny algorytm, taki jak na przykład z-buforowanie, które omówimy w rozdziale 9., poświęconym obiektom trójwymiarowym.

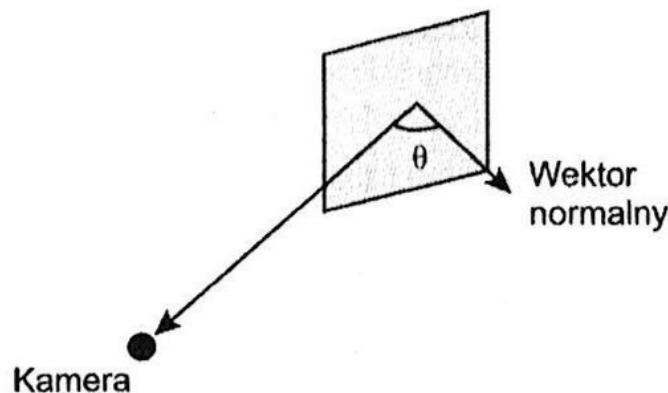
Aby ustalić, który wielokąt jest tylną powierzchnią, należy wprowadzić pojęcie wektora normalnego wielokąta. Wektor normalny wielokąta jest ustawiony względem wielokąta pod kątem prostym. Tak się wspaniale składa, że wektor normalny wskazuje w kierunku, w którym wielokąt jest skierowany.

Jeśli wektor normalny jest obrócony względem kierunku kamery o nie więcej niż 90 stopni, oznaczać to będzie, że wielokąt jest skierowany przodem do kamery, co pokazaliśmy na rysunku 7.16.

Teraz wystarczy po prostu wyliczyć dwie wartości: wektor normalny i kąt między dwoma wektorami. Zaczniemy od wyliczenia kąta między wektorami.

Rysunek 7.16.

Ustalanie, czy wielokąt jest zwrócony przodem do kamery

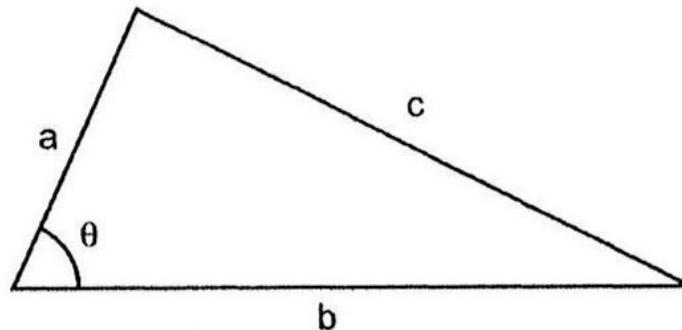


Iloczyn skalarny wektorów

Aby obliczyć kąt między wektorami, należy skorzystać z prawa cosinusów, które przy поминаем na rysunku 7.17.

Rysunek 7.17.

Prawo cosinusów



$$c^2 = a^2 + b^2 - 2 a b \cos \theta$$

Aby zastosować prawo cosinusów do wektorów, wystarczy zamiast długości a , b i c użyć wektorów U , V oraz $(U - V)$. W ten sposób prawo cosinusów przyjmie następującą postać:

$$|U - V|^2 = |U|^2 + |V|^2 - 2 |U| |V| \cos \theta$$

W tym przypadku nie jest nam potrzebna znajomość dokładnej wartości kąta. Należy tylko wiedzieć, czy kąt ten nie jest mniejszy niż 90 stopni. Jeśli kąt jest mniejszy niż 90 stopni (czyli jest kątem ostrym), cosinus tego kąta będzie większy od 0. Innymi słowy, $\cos \theta > 0$. stąd też $2 |U| |V| \cos \theta > 0$. Wiedząc o tym, rozwiążemy wzór z twierdzenia cosinusów.

$$\begin{aligned} 2 |U| |V| \cos \theta &= |U|^2 + |V|^2 - |U - V|^2 \\ &= |U|^2 + |V|^2 - (U_x - V_x)^2 - (U_y - V_y)^2 - (U_z - V_z)^2 \\ &= |U|^2 + |V|^2 - U_x^2 - V_x^2 - U_y^2 - V_y^2 - U_z^2 - V_z^2 + 2U_xV_x + 2U_yV_y + 2U_zV_z \\ &= 2U_xV_x + 2U_yV_y + 2U_zV_z \end{aligned}$$

$$|U| |V| \cos \theta = U_xV_x + U_yV_y + U_zV_z$$

W ten sposób równanie znacznie się upraszcza, prawda? Aby ustalić, czy kąt między dwoma wektorami jest kątem ostrym, wystarczy wykonać tylko trzy mnożenia i dwa dodawania. I, co więcej, okazuje się (co nie powinno nikogo dziwić), że ktoś już wpadł

na to wcześniej. Przedstawione równanie nosi nazwę iloczynu skalarnego wektorów i jest jedną z podstaw rachunku wektorów. Zazwyczaj iloczyn skalarny wektorów oznacza się za pomocą kropki:

$$U \bullet V = U_x V_x + U_y V_y + U_z V_z = |U| |V| \cos \theta$$

Dzięki temu, uzbrojeni w iloczyn skalarny wektorów, jesteśmy w stanie bez trudu powiedzieć, kiedy kąt między dwoma wektorami jest kątem ostrym, a kiedy kątem prostym lub rozwartym:

- Jeśli $(U \bullet V < 0)$ to $(\theta > 90^\circ)$
- Jeśli $(U \bullet V = 0)$ to $(\theta = 90^\circ)$
- Jeśli $(U \bullet V > 0)$ to $(\theta < 90^\circ)$

Iloczyn skalarny wektorów będziemy wielokrotnie wykorzystywać w grafice trójwymiarowej, dlatego dodamy do klasy wektora Vector3D specjalną metodę getDotProduct(), umożliwiającą wyliczanie iloczynu skalarnego dwóch wektorów:

```
/*
    Zwraca iloczyn skalarny tego wektora i innego podanego
    wektora.
*/
public float getDotProduct(Vector3D v) {
    return x*v.x + y*v.y + z*v.z;
}
```

Wracając do naszego problemu usuwania tylnych powierzchni brył: teraz jesteśmy w stanie ustalić, czy kąt między wektorem normalnym wielokąta a kamerą jest kątem ostrym. Potrzebujemy tylko jakiegoś sposobu wyliczania wektora normalnego wielokąta.

Iloczyn wektorowy wektorów

Korzystając z iloczynu skalarnego wektorów, możemy ustalić, czy dwa wektory są względem siebie ustawione pod kątem ostrym lub kątem prostym, co ma miejsce, gdy ich iloczyn skalarny równy jest 0. Wiedza ta ułatwia nam również znajdowanie wektora normalnego wielokąta.

W porządku, a więc szukamy wektora normalnego dla pewnego określonego wielokąta. W tym celu należy wziąć dwa dowolne wektory wyznaczające ten wielokąt (nazwiemy je U i V). Iloczyn skalarny między każdym z tych wektorów a wektorem normalnym wielokąta, N , będzie równy 0:

$$U \bullet N = 0$$

$$V \bullet N = 0$$

Stąd:

$$U_x N_x + U_y N_y + U_z N_z = 0$$

$$V_x N_x + V_y N_y + V_z N_z = 0$$

Ten układ równań ma nieskończoną liczbę rozwiązań, jedno z nich można jednak bardzo łatwo znaleźć, korzystając z wyznaczników (definicję wyznacznika znaleźć można w każdym podręczniku algebry liniowej):

$$\begin{aligned}N_x &= U_y V_z - U_z V_y \\N_y &= U_z V_x - U_x V_z \\N_z &= U_x V_y - U_y V_x\end{aligned}$$

I w ten właśnie sposób otrzymujemy wektor normalny. To rozwiązanie nazywane jest iloczynem wektorowym wektorów, który matematycy oznaczają za pomocą krzyżyka:

$$U \times V = (U_y V_z - U_z V_y, U_z V_x - U_x V_z, U_x V_y - U_y V_x)$$

Podobnie jak w przypadku iloczynu skalarnego, iloczyn wektorowy również będzie nieustannie wykorzystywany w grafice trójwymiarowej, dlatego dodamy do klasy wektora Vector3D metodę pozwalającą wyliczyć i ten iloczyn:

```
/**
 * Przypisuje temu wektorowi wynik iloczynu wektorowego dwóch
 * podanych wektorów. Jeden z podanych wektorów może być
 * tym wektorem (this).
 */
public void setToCrossProduct(Vector3D u, Vector3D v) {
    // Przypisz najpierw do zmiennych lokalnych.
    // bo u lub v mogą być wektorem 'this'.
    float x = u.y * v.z - u.z * v.y;
    float y = u.z * v.x - u.x * v.z;
    float z = u.x * v.y - u.y * v.x;
    this.x = x;
    this.y = y;
    this.z = z;
}
```

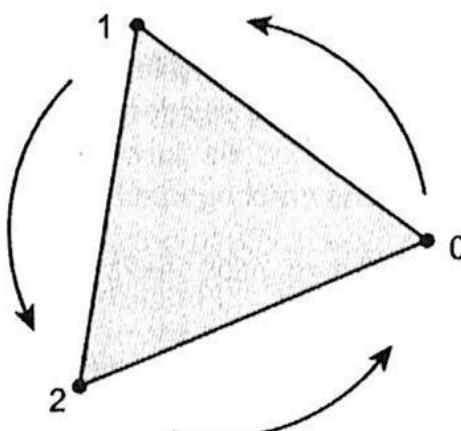
W tym rozwiążaniu znajduje się pewien haczyk, na który zapewne zwróciłeś uwagę. Wektor normalny jest prostopadły do wielokąta, jak jednak ustalić, w którą z dwóch możliwych stron wskazuje? Może przecież wskazywać albo w kierunku, w którym wielokąt jest zwrócony przodem, albo w kierunku odwrotnym — oba wektory będą prostopadłe do wielokąta. I jak w ogóle ustalić, która strona jest „przodem”, a która „tyłem” wielokąta?

Odpowiedź brzmi: wszystko zależy od sposobu, w jaki definiowane są wierzchołki wielokąta. W przypadku prawostronnego układu współrzędnych przód wielokąta będzie po tej stronie, z której patrząc, zobaczymy wierzchołki wielokąta ustawione w porządku odwrotnym do kierunku ruchu wskazówek zegara, tak jak to zostało pokazane na rysunku 7.18. Jeśli popatrzylibyśmy na wielokąt z drugiej strony, to wierzchołki ustawione byłyby w kierunku zgodnym z ruchem wskazówek zegara.

Dlatego należy upewnić się, że nasze wielokąty są definiowane w ten sposób, aby — gdy patrzymy na nie z przodu — ich wierzchołki ustawione były w porządku odwrotnym do ruchu wskazówek zegara.

Rysunek 7.18.

„Przodem” wielokąta będzie ta strona, z której wierzchołki wielokąta widać będącymi jako ustawione w kierunku odwrotnym do ruchu wskazówek zegara



Teraz należy dodać do klasy wielokąta `Polygon3D` kilka metod, które umożliwią wyliczanie jego wektora normalnego. Wektor normalny będzie zmienną składową klasy, dzięki czemu nie będzie konieczne wyliczanie go za każdym razem od nowa. Ponieważ każdy wierzchołek wielokąta jest punktem, użyjemy wektorów `temp1` i `temp2`, by wyliczyć dwa wektory definiujące wielokąt.

/**

Wylicza jednostkowy wektor normalny dla tego wielokąta.

Metoda ta wykorzystuje pierwszy, drugi i trzeci wierzchołek, by wyliczyć wektor normalny, więc jeśli wierzchołki te leżą w jednej linii, metoda ta nie będzie działać. W tym przypadku można otrzymać wektor normalny z boków wielokąta.

Metoda `setNormal()` pozwala ręcznie ustawić wektor normalny.

Wykorzystuje ona do obliczeń statyczne obiekty w klasie `Polygon3D`, więc nie gwarantuje bezpieczeństwa obliczeń w przypadku wielu równolegle wykorzystywanych instancji klasy `Polygon3D`.

*/

```
public Vector3D calcNormal() {
    if (normal == null) {
        normal = new Vector3D();
    }
    temp1.setTo(v[2]);
    temp1.subtract(v[1]);
    temp2.setTo(v[0]);
    temp2.subtract(v[1]);
    normal.setToCrossProduct(temp1, temp2);
    normal.normalize();
    return normal;
}
```

/**

Pobiera wektor normalny tego wielokąta. Jeśli wierzchołki się zmieniają, należy użyć `calcNormal()`.

*/

```
public Vector3D getNormal() {
    return normal;
}
```

```

 $\ast\ast$ 
    Definiuje wektor normalny dla tego wielokąta.
 $\ast\ast$ 
public void setNormal(Vector3D n) {
    if (normal == null) {
        normal = new Vector3D(n);
    }
    else {
        normal.setTo(n);
    }
}

```

Metoda `calcNormal()` wykorzystuje pierwszy, drugi i trzeci wierzchołek wielokąta do wyliczenia jego wektora normalnego. Oczywiście metoda ta nie działa, jeśli wszystkie trzy wierzchołki znajdują się w jednej linii (czyli są współliniowe). W kolejnym rozdziale opowiem o krawędziach wielokątów. W przypadkach takich jak ten możemy wykorzystać wektory normalne, wyliczone przy użyciu krawędzi wielokąta.

Na koniec możemy wreszcie ustalić, czy wielokąt znajduje się przodem do kamery. W tym celu trzeba dodać do klasy wielokąta `Polygon3D` kolejną metodę o nazwie `isFacing()`, która będzie sprawdzać, czy wielokąt skierowany jest przodem do wybranego punktu:

```

 $\ast\ast$ 
Sprawdza, czy wielokąt zwrócony jest przodem do określonego punktu.
Metoda ta wykorzystuje do obliczeń statyczne obiekty w klasie Polygon3D,
więc nie gwarantuje bezpieczeństwa obliczeń w przypadku wielu
równolegle wykorzystywanych instancji klasy Polygon3D.
 $\ast\ast$ 
public boolean isFacing(Vector3D u) {
    temp1.setTo(u);
    temp1.subtract(v[0]);
    return (normal.getDotProduct(temp1) >= 0);
}

```

Metody `calcNormal()` i `isFacing()` wykorzystują do podręcznych obliczeń obiekty `Vector3D`, co sprawia, że klasy `Polygon3D` nie można bezpiecznie wykorzystywać w równoległych wątkach. W tym przypadku niczym to jednak nie grozi, ponieważ wywołujemy te metody tylko w głównym wątku gry.

Dodatkowe właściwości iloczynu skalarnego i wektorowego

Dla Twojej wygody prezentuję tutaj jeszcze kilka właściwości iloczynu skalarnego i iloczynu wektorowego wektorów. Niektóre z nich zostaną wykorzystane w następnym rozdziale. W przedstawionych tu wzorach litery A , B i C oznaczają wektory, natomiast mała litera s — liczbę rzeczywistą.

Właściwości iloczynu skalarnego:

$$\begin{aligned} A \bullet B &= B \bullet A \\ (sA) \bullet B &= s(A \bullet B) \\ A \bullet B + A \bullet C &= A \bullet (B + C) \end{aligned}$$

Właściwości iloczynu wektorowego:

$$A \times B = -B \times A$$

$$A \times (B + C) = A \times B + A \times C$$

Przemienność iloczynu wektorowego względem skalarnego dla trzech wektorów:

$$(A \times B) \bullet C = (C \times A) \bullet B = (B \times C) \bullet A$$

Właściwości iloczynu wektorowego trzech wektorów:

$$A \times (B \times C) = B (A \bullet C) - C (A \bullet B)$$

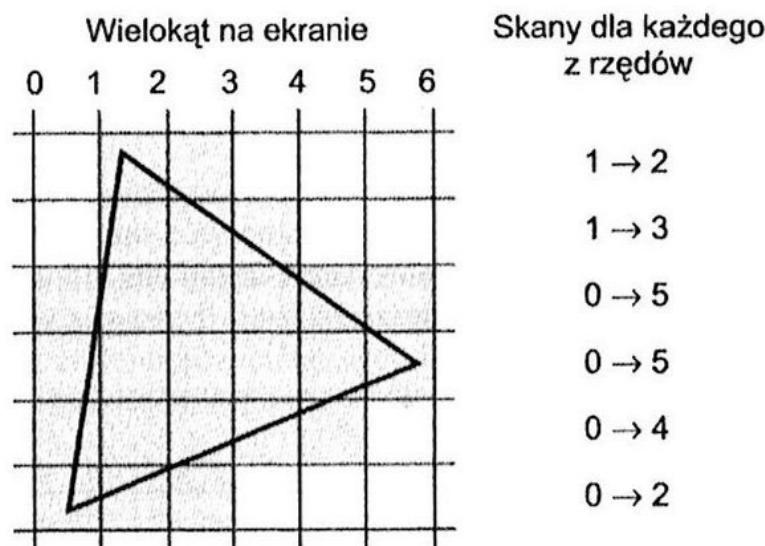
Rysowanie wielokątów za pomocą konwertera skanującego

W przykładzie Simple3DTest1 rysowaliśmy wielokąty, konwertując rzutowane wielokąty na obiekty GeneralPath. Jak można się było przekonać, technika ta nie daje najlepszych rezultatów, ponieważ współrzędne wierzchołków wielokątów są zaokrąglane do najbliższej liczby całkowitej. Aby naprawić tę niedoróbkę, przygotujemy własny konwerter skanujący, który będzie przechowywał wierzchołki o współrzędnych wyrażonych liczbami zmiennoprzecinkowymi. *Konwerter skanujący* (ang. *scan converter*) konwertuje wielokąty na poziome skany, w wyniku czego wielokąt zostaje wykreślony jako seria poziomych linii. Z takiego własnego konwertera skanującego będziemy korzystać również w następnym rozdziale podczas mapowania tekstur.

Przechowywanie współrzędnych wierzchołków w postaci liczb zmiennoprzecinkowych może się wydawać bezsensowne, skoro, jak pamiętamy, w przypadku ekranu wykorzystywane są i tak tylko współrzędne wyrażone liczbami całkowitymi. W końcu położenie wielokąta będzie przecież i tak opisywane za pomocą liczb całkowitych. Trick polega jednak na tym, aby przeliczenie liczb zmiennoprzecinkowych na liczby całkowite wykonać później. Domyślnie klasa GeneralPath zaokrąglą każdy punkt do najbliższej liczby całkowitej przed narysowaniem krawędzi wielokąta. My natomiast przekonwertujemy liczby zmiennoprzecinkowe na liczby całkowite dopiero po narysowaniu krawędzi. W ten sposób wielokąt zostanie narysowany znacznie dokładniej.

Rysowanie obwodu wielokąta przypomina zabawę w „połącz kropki”: rysujemy po prostu po kolejni każdy bok lub, inaczej mówiąc, linie łączące kolejne wierzchołki. Natomiast wypełnianie wielokąta kolorem może się na początku wydawać dość złożonym problemem. Aby to uprościć, rozbijemy ten proces na mniejsze kroki. Rysunek 7.19 przedstawia wielokąt podzielony na serię poziomych skanów, po jednym dla każdego rzędu. Aby wypełnić wielokąt kolorem, wystarczy dla każdego skanu narysować poziomą linię odpowiedniej długości.

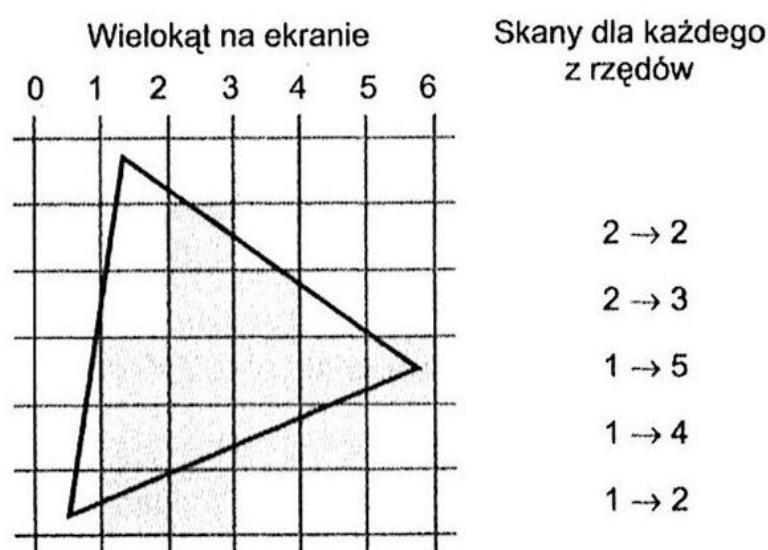
Rysunek 7.19.
Konwertowanie wielokąta na serię poziomych skanów



Na rysunku 7.19 pokazaliśmy, że w ten sposób rysowany będzie każdy z pikseli tworzących wielokąt. Co jednak stanie się, jeśli dwa wielokąty będą znajdować się tuż obok siebie? W tym wariantie wielokąty będą miały wspólne krawędzie, tak więc niektóre piksele zostaną narysowane dwukrotnie. Nie będzie dokładnie wiadomo, który piksel należy, do którego wielokąta.

Aby to naprawić, wprowadzimy rozróżnienie między prawą a lewą krawędzią każdego z rysowanych wielokątów. Lewe krawędzie będziemy zaokrąglać za pomocą funkcji `ceil()`, a prawe krawędzie za pomocą `ceil()-1`. Nowy sposób rysowania został przedstawiony na rysunku 7.20.

Rysunek 7.20.
Bardziej rygorystyczna metoda konwertowania wielokąta na serię poziomych skanów



Po zastosowaniu techniki zilustrowanej na rysunku 7.20 dwa sąsiadujące ze sobą wielokąty nie będą już na siebie nachodzić.

Ponadto w klasie MoreMath korzystać będziemy z przygotowanej przez nas funkcji `ceil()`:

```
public static int ceil(float f) {
    if (f > 0) {
        return (int)f + 1;
    }
}
```

```

        else {
            return (int)f;
        }
    }
}

```

Funkcja ta konwertuje liczby zmiennoprzecinkowe typu float na liczby całkowite, podczas gdy oryginalna funkcja Math.ceil() pobiera jako parametr liczbę zmiennoprzecinkową typu double i zwraca liczbę całkowitą, ale też typu double. Dzięki zastosowaniu tej własnej funkcji unikniemy przechłonnego konwertowania z i na typ double.

Teraz możemy przygotować prostą klasę skanu Scan, przedstawioną na listingu 7.7.

Listing 7.7. Klasa Scan

```


    /**
     * Pozioma linia skanu.
    */
    public static class Scan {
        public int left;
        public int right;

        /**
         * Wyznacza lewą i prawą granicę tego skanu, jeśli
         * wartość x jest na zewnątrz bieżącej granicy.
        */
        public void setBoundary(int x) {
            if (x < left) {
                left = x;
            }
            if (x-1 > right) {
                right = x-1;
            }
        }

        /**
         * Oczyszcza tę linię skanu.
        */
        public void clear() {
            left = Integer.MAX_VALUE;
            right = Integer.MIN_VALUE;
        }

        /**
         * Ustala, czy skan jest poprawny (t.j. czy left <= right).
        */
        public boolean isValid() {
            return (left <= right);
        }

        /**
         * Definiuje ten skan.
        */
        public void setTo(int left, int right) {
            this.left = left;
            this.right = right;
        }
    }
}


```

```
/**  
 * Sprawdza, czy ten skan jest równy zdefiniowanym wartościom.  
 */  
public boolean equals(int left, int right) {  
    return (this.left == left && this.right == right);  
}
```

Klasa Scan jest podklassą klasy ScanConverter.

Dla każdej linii skanu tworzony jest nowy obiekt Scan, a konwerter skanujący skanuje każdą krawędź wielokąta. Za każdym razem, gdy bada pojawienie się krawędzi w kolejnej linii skanu, wywołuje metodę setBoundary() obiektu Scan. Wywołanie tej meto-

poziomych linii w oknie obrazu ViewWindow oraz informacje o dolnej (bottom) i górnej (top) granicy ostatnio skanowanego wielokąta. Oto lista zmiennych składowych klasy ScanConverter:

```
ViewWindow view;  
Scan[] scans;  
int top;  
int bottom;
```

Metody wykorzystywane w klasie ScanConverter do skanowania wielokąta przedstawione zostały na listingu 7.8:

Listingu 7.8 Metody programu konwertera skanującego

```
int numVertices = polygon.getNumVertices();  
for (int i=0; i<numVertices; i++) {  
    Vector3D v1 = polygon.getVertex(i);  
    Vector3D v2;
```

```
        if (i == numVertices - 1) {
            v2 = polygon.getVertex(0);
        }
        else {
            v2 = polygon.getVertex(i+1);
        }

        // Sprawdź, czy v1.y < v2.y:
        if (v1.y > v2.y) {
            Vector3D temp = v1;
            v1 = v2;
            v2 = temp;
        }
        float dy = v2.y - v1.y;

        // Ignoruj linie poziome.
        if (dy == 0) {
            continue;
        }

        int startY = Math.max(MoreMath.ceil(v1.y), minY);
        int endY = Math.min(MoreMath.ceil(v2.y)-1, maxY);
        float dx = v2.x - v1.x;

        float gradient = dx / dy;

        // Wykrywanie krawędzi (z równania prostej).

        for (int y=startY; y<=endY; y++) {
            int x = MoreMath.ceil(v1.x + (y - v1.y) * gradient);
            // Upewnij się, że x znajduje się wewnątrz ram obrazu.
            x = Math.min(maxX+1, Math.max(x, minX));
            scans[y].setBoundary(x);
        }
    }
}
```

Klasa ScanConverter konwertuje rzutowany wielokąt na ciąg poziomych skanów. Sprawdza również, czy skany te mieścią się w oknie obrazu.

Warto zauważyć, że konwerter ScanConverter działać będzie tylko w przypadku wielokątów wypukłych. Wielokąty wklęsłe mogą mieć „zagłębienia”, które konwerter przeoczy.

Oto pętla, w której wykonywane są po kolej i wszystkie skany:

```
int y = scanConverter.getTopBoundary();
while (y<=scanConverter.getBottomBoundary()) {
    ScanConverter.Scan scan = scanConverter.getScan(y);
    if (scan.isValid()) {
        g.drawLine(scan.left, y, scan.right, y);
    }
    y++;
}
```

Optymalizowanie konwertera skanującego za pomocą liczb stałoprzecinkowych

Przedstawiony tu prosty konwerter skanujący co prawda działa, ale mógłby działać szybciej. Jego powolność będzie szczególnie widoczna, gdy na ekranie pojawi się wiele wielokątów.

Jednym z powodów, dla którego kod ten nie jest wystarczająco szybki, jest konieczność wykonywania wielu obliczeń na liczbach zmiennoprzecinkowych dla każdej poziomej linii skanu. Aby przyspieszyć działanie konwertera skanującego, można użyć liczb całkowitych zamiast zmiennoprzecinkowych i spróbować zmniejszyć liczbę obliczeń wykonywanych dla każdej linii skanu.

Oczywiście wykonywanie obliczeń na liczbach całkowitych spowoduje, że dokładność będzie mniejsza od tej, której byśmy pragnęli — trzeba więc podejść do korzystania z liczb całkowitych w twórczy sposób. W tym celu należy pomnożyć każdą z liczb zmiennoprzecinkowych przez odpowiedni współczynnik, by otrzymać wartość całkowitą. Dla przykładu, jeśli pomnożymy liczbę 10,5 przez współczynnik 1 000, otrzymamy liczbę całkowitą 10 500.

Istotny jest oczywiście wybór właściwego współczynnika. Jeśli użyjemy współczynnika będącego potęgą liczby 2, to konwertowanie między wartościami zmiennoprzecinkowymi a odpowiadającymi im liczbami całkowitymi można będzie szybko wykonać po prostu za pomocą odpowiedniego przesuwania bitów.

W prezentowanych tu przykładach współczynnikiem jest 2^{16} . Ponieważ liczby całkowite mają 32 bity długości, użycie tego współczynnika spowoduje, że na część całkowitą przeznaczone będzie 16 pierwszych bitów, a na część ułamkową 16 ostatnich. Ta technika przedstawiania liczb ułamkowych nazywana jest często techniką liczb stałoprzecinkowych, ponieważ dla części dziesiętnej ułamka przeznaczona jest zawsze ścisłe określona liczba bitów w liczbie całkowitej.

Zacznijmy od stałych, które definiują skalowane przez współczynnik i zapisane w liczbach całkowitych liczby stałoprzecinkowe:

```
final int SCALE_BITS = 16;  
final int SCALE = 1 << SCALE_BITS;  
final int SCALE_MASK = SCALE - 1;
```

Operator `<<` wykonuje przesuwanie bitów, a `1 << 16` wykonuje przesunięcie odpowiadające mnożeniu przez współczynnik 2^{16} . Dysponując tymi stałymi, możemy przygotować kod konwertujący liczby zmiennoprzecinkowe na stałoprzecinkowe i na odwrót.

Konwertowanie liczby zmiennoprzecinkowej na stałoprzecinkową:

```
int fixed = (int) (value * SCALE);
```

Konwertowanie liczby stałoprzecinkowej na zmiennoprzecinkową:

```
float value = (float)fixed / SCALE;
```

Konwertowanie liczby całkowitej na liczbę stałoprzecinkową:

```
int fixed = value << SCALE_BITS;
```

Konwertowanie liczby stałoprzecinkowej na całkowitą:

```
int value = fixed >> SCALE_BITS;
```

Pobieranie części ułamkowej z liczby stałoprzecinkowej:

```
float frac = (float)(fixed & SCALE_MASK) / SCALE;
```

Dodawanie i odejmowanie liczb stałoprzecinkowych wykonywane jest tak samo, jak zwykłe dodawanie i odejmowanie. Mnożenie liczby stałoprzecinkowej oraz zwykłej liczby całkowitej również będzie działać w ten sam sposób.

Niemniej mnożenie lub dzielenie przez siebie dwóch liczb stałoprzecinkowych wykonuje się trochę inaczej niż zwykłe operacje mnożenia i dzielenia. Pomnożenie przez siebie dwóch liczb 32-bitowych da w rezultacie liczbę 64-bitową. Tak więc w przypadku dwóch liczb stałoprzecinkowych mnożenie takie podwoi liczbę bitów znajdujących się w części ułamkowej wyniku. Aby to naprawić, należy obliczenia wykonywać za pomocą liczb 64-bitowych (typ long) — wówczas nie będzie konieczne przycinanie bitów, które mogą być nam potrzebne. Po wykonaniu obliczenia należy wykonać przesunięcie bitów, które przekonwertuje wynik z powrotem na standardową liczbę stałoprzecinkową.

```
int fixed = (int) (((long)fixed1 * fixed2) >> SCALE_BITS);
```

Podobnie dzieje się w przypadku dzielenia, co również daje się łatwo naprawić:

```
int fixed = (int) (((long)fixed1 << SCALE_BITS) / fixed2);
```

Prawdę powiedziawszy, w naszym kodzie nie będzie potrzeby dzielenia lub mnożenia przez siebie liczb stałoprzecinkowych, niemniej pokazałem, jak to uczynić, by zaspokoić Twoją ewentualną ciekawość.

Skoro już wyjaśniliśmy, jak działają liczby stałoprzecinkowe, oto zoptymalizowany konwerter skanujący:

```
int minX = view.getLeftOffset();
int maxX = view.getLeftOffset() + view.getWidth() - 1;
int minY = view.getTopOffset();
int maxY = view.getTopOffset() + view.getHeight() - 1;

int numVertices = polygon.getNumVertices();
for (int i=0; i<numVertices; i++) {
    Vector3D v1 = polygon.getVertex(i);
    Vector3D v2;
    if (i == numVertices - 1) {
        v2 = polygon.getVertex(0);
    }
    else {
        v2 = polygon.getVertex(i+1);
    }
```

```
// Upewnij się, że v1.y < v2.y:  
if (v1.y > v2.y) {  
    Vector3D temp = v1;  
    v1 = v2;  
    v2 = temp;  
}  
float dy = v2.y - v1.y;  
  
// Ignoruj linie poziome.  
if (dy == 0) {  
    continue;  
}  
  
int startY = Math.max(MoreMath.ceil(v1.y), minY);  
int endY = Math.min(MoreMath.ceil(v2.y)-1, maxY);  
top = Math.min(top, startY);  
bottom = Math.max(bottom, endY);  
float dx = v2.x - v1.x;  
  
// Specjalny przypadek: linia pionowa.  
if (dx == 0) {  
    int x = MoreMath.ceil(v1.x);  
    // Upewnij się, że x mieści się w granicach obrazu.  
    x = Math.min(maxX+1, Math.max(x, minX));  
    for (int y=startY; y<=endY; y++) {  
        scans[y].setBoundary(x);  
    }  
}  
else {  
    // Wykryj krawędź podczas skanu (używając równania prostej).  
    float gradient = dx / dy;  
  
    // Przytnij początek linii.  
    float startX = v1.x + (startY - v1.y) * gradient;  
    if (startX < minX) {  
        int yInt = (int)(v1.y + (minX - v1.x) /  
                      gradient);  
        yInt = Math.min(yInt, endY);  
        while (startY <= yInt) {  
            scans[startY].setBoundary(minX);  
            startY++;  
        }  
    }  
    else if (startX > maxX) {  
        int yInt = (int)(v1.y + (maxX - v1.x) /  
                      gradient);  
        yInt = Math.min(yInt, endY);  
        while (startY <= yInt) {  
            scans[startY].setBoundary(maxX+1);  
            startY++;  
        }  
    }  
    if (startY > endY) {  
        continue;  
    }
```

```

// Przytnij koniec linii.
float endX = v1.x + (endY - v1.y) * gradient;
if (endX < minX) {
    int yInt = MoreMath.ceil(v1.y + (minX - v1.x) / gradient);
    yInt = Math.max(yInt, startY);
    while (endY >= yInt) {
        scans[endY].setBoundary(minX);
        endY--;
    }
}
else if (endX > maxX) {
    int yInt = MoreMath.ceil(v1.y + (maxX - v1.x) / gradient);
    yInt = Math.max(yInt, startY);
    while (endY >= yInt) {
        scans[endY].setBoundary(maxX+1);
        endY--;
    }
}
if (startY > endY) {
    continue;
}

// Równanie proste z wykorzystaniem liczb całkowitych.
int xScaled = (int)(SCALE * v1.x +
    SCALE * (startY - v1.y) * dx / dy) + SCALE_MASK;
int dxScaled = (int)(dx * SCALE / dy);

for (int y=startY; y<=endY; y++) {
    scans[y].setBoundary(xScaled >> SCALE_BITS);
    xScaled+=dxScaled;
}
}

```

Oczywiście liczby stałoprzecinkowe nie oferują ani tej dokładności, ani zakresu, co liczby zmiennoprzecinkowe. Ponieważ wartości wyliczane podczas skanowania mogą wykracać poza zakres liczb stałoprzecinkowych, w przedstawionym tu kodzie przycinamy początek i koniec skanu, jeśli nie mieści się wewnątrz obrazu. Ponadto, zamiast wyliczać przecięcie z krawędzią w każdej kolejnej linii, użyjemy nachylenia krawędzi (`dxScaled`), by odpowiednio zwiększać wartość x (`xScaled`) w każdej kolejnej linii.

Ostateczna wersja konwertera skanującego ScanConverter nie jest idealnie zoptymalizowana. Pewne elementy mogłyby zapewne działać szybciej. Jedno z rozwiązań optymalizacyjnych mogłoby na przykład polegać na przycinaniu jeszcze przed skanowaniem dwuwymiarowych wielokątów, które nie mieszczą się w oknie obrazu. Dzięki temu nie trzeba byłoby już wykonywać pracochłonnego skanowania tych części wielokątów (i całych wielokątów), które i tak nie są widoczne. Niemniej nawet ta wersja klasy ScanConverter działa całkiem dobrze.

Przycinanie w trzech wymiarach

Ostatnim z udoskonaleń wprowadzanych w tym rozdziale do prostego przykładu Simple3DTest1 jest przycinanie w trzech wymiarach. Jak pamiętamy, wielokąt tworzący drzewo wyglądał bardzo dziwnie, gdy znajdował się tuż przed kamerą. A ponadto, gdy drzewo znajdowało się za kamerą, było wyświetlane do góry nogami. Aby to naprawić, przytniemy wielokąty, usuwając wszystko, co znajduje się poza polem widzenia kamery.

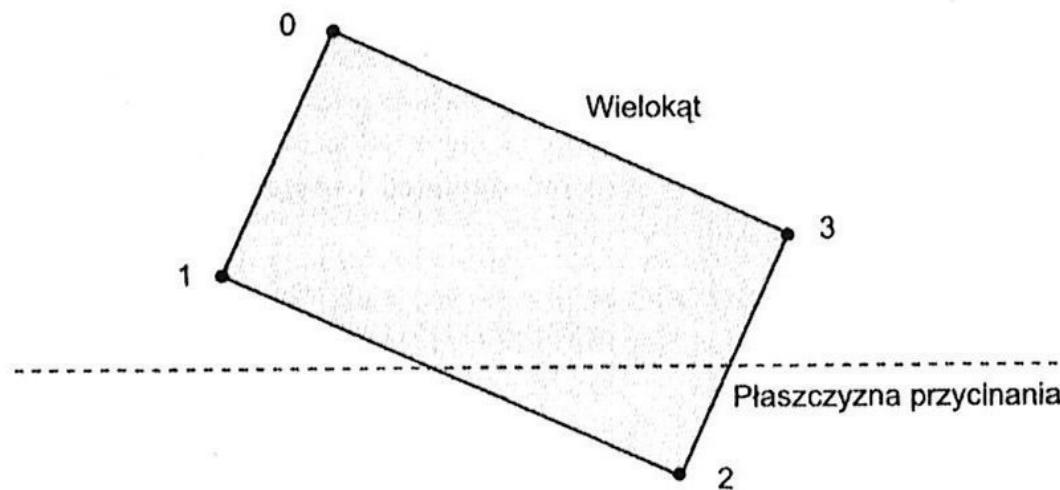
Pole widzenia kamery ma kształt czworobocznej piramidy (lub ostrosłupa). Możemy więc przycinać wielokąty wystające poza cztery płaszczyzny ograniczające pole widzenia. Alternatywnie można też przyciąć tylko wielokąty wystające przed pierwszy plan i pozwolić, aby przycinanie w dwóch wymiarach zajęło się resztą. To rozwiązanie działa całkiem dobrze i z niego właśnie będziemy korzystać w naszym przykładzie.

Gdy przycinamy wielokąt do płaszczyzny pierwszego planu, możliwe są trzy różne sytuacje: albo wielokąt jest w całości przed tą płaszczyzną, albo w całości za nią lub też może przecinać tę płaszczyznę. Jeśli znajduje się za nią, to wielokąt można zignorować. Jeśli przecina płaszczyznę, trzeba będzie przyciąć wielokąt do tej płaszczyzny, tak żeby rysować tylko tę część wielokąta, która znajduje się przed nią. Przycięcie wypukłego wielokąta do płaszczyzny tworzy wielokąt, który również jest wypukły, nie musimy się więc obawiać, że powstaną jakieś wielokąty wklęsłe.

Przyjrzyjmy się najpierw rysunkowi 7.21. Oto wielokąt przecinający płaszczyznę, do której jest przycinany. Na rysunku płaszczyzna ta została zaznaczona linią przerwana.

Rysunek 7.21.

Przycinanie wielokąta,
krok 1.: wielokąt
przecina płaszczyznę
przycinania



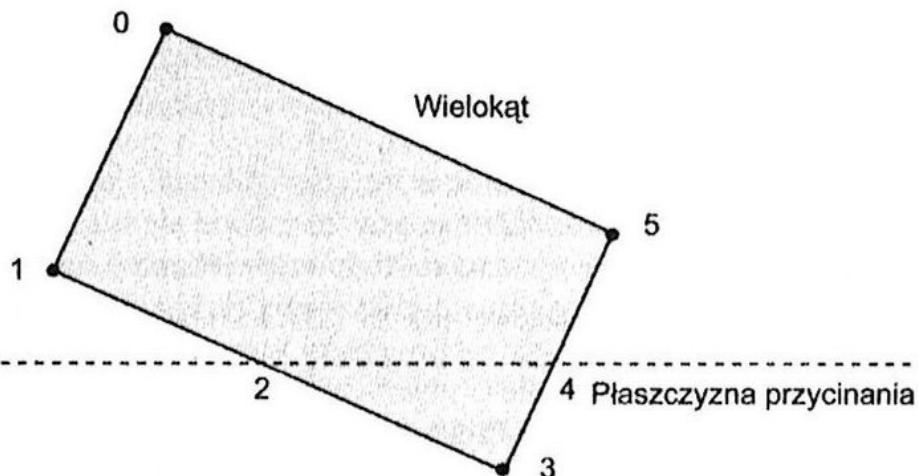
W kolejnym kroku (rysunek 7.22) dodajemy do wielokąta nowe wierzchołki w miejscu, gdzie krawędzie wielokąta przecinają się z płaszczyzną przycinania.

Po wykonaniu tej operacji mamy już tylko krawędzie znajdujące się przed powierzchnią przycinania lub za nią. W ostatnim kroku po prostu usuniemy krawędzie znajdujące się za powierzchnią przycinania lub, mówiąc inaczej, wszystkie wierzchołki znajdujące się za tą powierzchnią — tak, jak to zostało pokazane na rysunku 7.23.

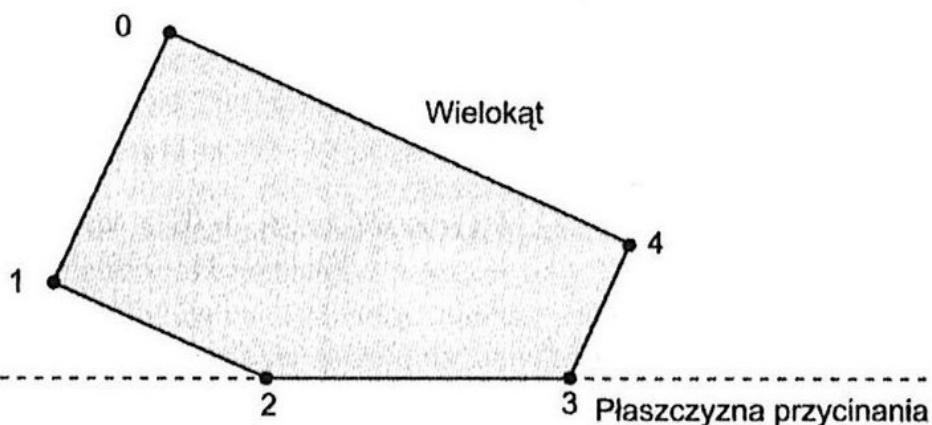
Ponieważ płaszczyzna pierwszego planu, do której wykonujemy przycinanie, jest równoległa do okna obrazu, wszystko, co należy uczynić, sprowadza się będzie do wybrania lokalizacji płaszczyzny przycinania. Na pozór mogłoby się wydawać, że dobrym

Rysunek 7.22.

Przycinanie, krok 2.:
dodajemy nowe
wierzchołki w miejscu,
w którym krawędzie
wielokąta przecinają się
z krawędzią przycinania

**Rysunek 7.23.**

Przycinanie, krok 3.:
usuwamy wszystkie
krawędzie znajdujące
się za płaszczyzną
przycinania



rozwiązaniem jest wybranie płaszczyzny $z = 0$, niemniej taki wybór będzie powodował problemy związane z koniecznością dzielenia przez 0. W prezentowanych tutaj przykładach wykorzystujemy płaszczyznę $z = -1$.

Aby znaleźć punkty przecięcia krawędzi wielokąta z płaszczyzną przycinania, ponownie skorzystamy z równania prostej. Znając położenie płaszczyzny przycinania (czyli jej współrzędną z), otrzymujemy następujące równania, umożliwiające odnajdywanie wspólnych punktów przecięcia:

$$\begin{aligned}x &= (z - z_1)(x_2 - x_1) / (z_2 - z_1) + x_1 \\y &= (z - z_1)(y_2 - y_1) / (z_2 - z_1) + y_1 \\z &= z\end{aligned}$$

Algorytm przycinający zaimplementowany został w metodzie `clip()` klasy wielokąta `Polygon3D`. Oto kod tej metody:

```
/***
 * Przycina ten wielokąt w taki sposób, aby wszystkie jego
 * wierzchołki znalazły się przed płaszczyzną przycinania, clipZ
 * (czyli aby ich współrzędne z spełniały warunek z <= clipZ).
 * Wartość clipZ powinna być różna od 0, co pozwala na uniknięcie
 * problemów związanych z dzieleniem przez 0.
 * Zwraca wartość true (prawda), jeśli wielokąt przynajmniej częściowo
 * znajduje się przed powierzchnią przycinania.
 */
public boolean clip(float clipZ) {
    ensureCapacity(numVertices * 3);

    boolean isCompletelyHidden = true;
```

```
// Wstawia nowe wierzchołki, dzięki którym każda z krawędzi będzie
// albo całkowicie przed, albo całkowicie za płaszczyzną przycinania.
for (int i=0; i<numVertices; i++) {
    int next = (i + 1) % numVertices;
    Vector3D v1 = v[i];
    Vector3D v2 = v[next];
    if (v1.z < clipZ) {
        isCompletelyHidden = false;
    }
    // Sprawdź, czy v1.z < v2.z;
    if (v1.z > v2.z) {
        Vector3D temp = v1;
        v1 = v2;
        v2 = temp;
    }
    if (v1.z < clipZ && v2.z > clipZ) {
        float scale = (clipZ-v1.z) / (v2.z - v1.z);
        insertVertex(next,
                     v1.x + scale * (v2.x - v1.x),
                     v1.y + scale * (v2.y - v1.y),
                     clipZ);
        // Pomiń wierzchołek, który właśnie utworzyliśmy.
        i++;
    }
}
if (isCompletelyHidden) {
    return false;
}

// Usuń wszystkie wierzchołki, dla których z > clipZ.
for (int i=numVertices-1; i>=0; i--) {
    if (v[i].z > clipZ) {
        deleteVertex(i);
    }
}

return (numVertices >= 3);
}

/**
 * Wstawia nowy wierzchołek pod podanym indeksem.
 */
protected void insertVertex(int index, float x, float y,
                            float z)
{
    Vector3D newVertex = v[v.length-1];
    newVertex.x = x;
    newVertex.y = y;
    newVertex.z = z;
    for (int i=v.length-1; i>index; i--) {
        v[i] = v[i-1];
    }
    v[index] = newVertex;
    numVertices++;
}
```

```

    /**
     * Usuwa wierzchołek o podanym indeksie.
    */
    protected void deleteVertex(int index) {
        Vector3D deleted = v[index];
        for (int i=index; i<v.length-1; i++) {
            v[i] = v[i+1];
        }
        v[v.length-1] = deleted;
        numVertices--;
    }
}

```

Zaprezentowana tutaj technika przycinania jest dość prosta, bowiem oparta jest na założeniu, że wykonujemy tylko przycinanie do pionowego pierwszego planu. Opcjonalnie można jeszcze zaprogramować przycinanie do pionowego planu tła (tak, aby wielokąty, które są zbyt daleko, nie były rysowane). Oczywiście przycinanie do prawej, lewej oraz dolnej i górnej płaszczyzny ograniczającej pole widzenia dodaje sporo dodatkowej pracy konwerterowi skanującemu, ale już samo przycięcie wielokątów do pierwszego planu w zupełności wystarcza i sprawia, że grafika wygląda poprawnie.

Ostateczny potok renderowania

Teraz trzeba dodać do potoku renderowania, którego używaliśmy w przykładzie SimpleD3Test1, usuwanie tylnych powierzchni, przycinanie i rysowanie za pomocą konwertera skanującego:

- 1.** Sprawdź, czy wielokąt znajduje się przodem do kamery.
- 2.** Zastosuj transformację.
- 3.** Przytnij.
- 4.** Rzutuj wielokąt na okno obrazu.
- 5.** Konwertuj na linie za pomocą konwertera skanującego.
- 6.** Rysuj wielokąt.

Aby rysować wielokąty, stosując się do tego nowego potoku, przygotujemy abstrakcyjną klasę PolygonRenderer, którą przedstawiłem na listingu 7.9.

Listing 7.9. PolygonRenderer.java

```

package com.brackeen.javagamebook.graphics3D;

import java.awt.Graphics2D;
import java.awt.Color;
import com.brackeen.javagamebook.math3D.*;

/**
 * Klasa PolygonRenderer jest abstrakcyjną klasą, odpowiadającą
 * za transformowanie wielokątów i rysowanie ich na ekranie.
 */

```

```
/*
public abstract class PolygonRenderer {

    protected ScanConverter scanConverter;
    protected Transform3D camera;
    protected ViewWindow viewWindow;
    protected boolean clearViewEveryFrame;
    protected Polygon3D sourcePolygon;
    protected Polygon3D destPolygon;

    /**
     * Tworzy nowy obiekt PolygonRenderer dla podanych obiektów
     * Transform3D (kamery) i ViewWindow (okna obrazu). Obraz jest
     * oczyszczany w momencie wywołania funkcji startFrame().
    */
    public PolygonRenderer(Transform3D camera,
                           ViewWindow viewWindow)
    {
        this(camera, viewWindow, true);
    }

    /**
     * Tworzy nowy obiekt PolygonRenderer dla podanych obiektów
     * Transform3D (kamery) i ViewWindow (okna obrazu). Jeśli
     * clearViewEveryFrame ma wartość true, obraz będzie oczyszczany
     * w momencie wywołania funkcji startFrame().
    */
    public PolygonRenderer(Transform3D camera,
                           ViewWindow viewWindow, boolean clearViewEveryFrame)
    {
        this.camera = camera;
        this.viewWindow = viewWindow;
        this.clearViewEveryFrame = clearViewEveryFrame;
        init();
    }

    /**
     * Tworzy konwerter skanujący i docelowy wielokąt.
    */
    protected void init() {
        destPolygon = new Polygon3D();
        scanConverter = new ScanConverter(viewWindow);
    }

    /**
     * Pobiera kamerę wykorzystywaną przez ten obiekt PolygonRenderer.
    */
    public Transform3D getCamera() {
        return camera;
    }

    /**
     * Wskazuje początek renderowania klatki obrazu. Metoda ta
     * powinna być wywoływana dla każdej klatki, zanim jeszcze
     * zostanie narysowany jakikolwiek wielokąt.
    */
}
```

```

/*
public void startFrame(Graphics2D g) {
    if (clearViewEveryFrame) {
        g.setColor(Color.black);
        g.fillRect(viewWindow.getLeftOffset(),
                   viewWindow.getTopOffset(),
                   viewWindow.getWidth(), viewWindow.getHeight());
    }
}

/***
Informuje o końcu renderowania klatki obrazu. Metoda ta
powinna być wywoływana dla każdej klatki już po narysowaniu
wszystkich wielokątów.
*/
public void endFrame(Graphics2D g) {
    // Tymczasem nie wykonuj żadnych działań.
}

/***
Transformuje i rysuje wielokąt.
*/
public boolean draw(Graphics2D g, Polygon3D poly) {
    if (poly.isFacing(camera.getLocation())) {
        sourcePolygon = poly;
        destPolygon.setTo(poly);
        destPolygon.subtract(camera);
        boolean visible = destPolygon.clip(-1);
        if (visible) {
            destPolygon.project(viewWindow);
            visible = scanConverter.convert(destPolygon);
            if (visible) {
                drawCurrentPolygon(g);
                return true;
            }
        }
    }
    return false;
}

/***
Rysuje bieżący wielokąt. W tym momencie bieżący
wielokąt jest transformowany, przycinany, rzutowany,
konwertowany na linie i rysowany.
*/
protected abstract void drawCurrentPolygon(Graphics2D g);
}

```

Przedstawiona tu klasa `PolygonRender` jest klasą abstrakcyjną, która dostarcza szkieletu renderowania wielokątów. Jej podstawowym przeznaczeniem jest implementacja potoku renderowania pojedynczego wielokąta wewnątrz metody `draw()`. Wszelkie podklasy niezbędne do zaimplementowania metody rysującej bieżący wielokąt `drawCurrentPolygon()` będą zajmować się rysowaniem końcowego wielokąta, konwertowanego na

skany linii. Ponadto algorytm renderujący opcjonalnie czyści jeszcze obraz, jeśli wywołana zostanie metoda startFrame(), zaczynająca nową klatkę. Ekran oczywiście czyścimy dlatego, że rysowane wielokąty nie zapelnią całego ekranu. W przypadku scen trójwymiarowych, w których każdy piksel ekranu jest pokrywany przez jakiś wielokąt, nie ma żadnego powodu, by czyścić obraz.

W kolejnym rozdziale, poświęconym mapowaniu tekstur i oświetleniu, przygotujemy kilka różnych podklas PolygonRender dla różnych typów wielokątów. W obecnej chwili zajmować będziemy się tylko rysowaniem wielokątów jednolicie wypełnionych kolorem, tak więc przygotujemy podkласę SolidPolygonRender, której metoda drawCurrentPolygon() została przedstawiona na listingu 7.10.

Listing 7.10. *SolidPolygonRender.drawCurrentPolygon*

```
protected void drawCurrentPolygon(Graphics2D g) {  
  
    // Ustaw kolor.  
    if (sourcePolygon instanceof SolidPolygon3D) {  
        g.setColor(((SolidPolygon3D)sourcePolygon).getColor());  
    }  
    else {  
        g.setColor(Color.GREEN);  
    }  
  
    // Rysuj linie skanów.  
    int y = scanConverter.getTopBoundary();  
    while (y<=scanConverter.getBottomBoundary()) {  
        ScanConverter.Scan scan = scanConverter.getScan(y);  
        if (scan.isValid()) {  
            g.drawLine(scan.left, y, scan.right, y);  
        }  
        y++;  
    }  
}
```

Klasa SolidPolygonRender implementuje metodę drawCurrentPolygon(), która rysuje wielokąt, tworząc go z listy skanów linii w konwerterze ScanConverter. Warto zauważyc, że metoda ta wywoływana jest tylko wewnątrz metody draw(). Oznacza to, że w momencie jej wywołania bieżący wielokąt destPolygon jest transformowany, przycinany, rzutowany, rozbijany na linie za pomocą konwertera skanującego i w końcu wyświetlany w oknie obrazu. Cały czas dostępny jest również wielokąt źródłowy, na wypadek gdyby potrzebne były jakieś jego dane.

Jeśli wielokąt źródłowy nie będzie instancją klasy SolidPolygon3D (czyli, innymi słowy, nie będzie miał przypisanego żadnego koloru), to rysowany wielokąt zostanie wypełniony domyślnym kolorem zielonym.

Teraz przygotujemy kolejny program testowy, by wypróbować nowy mechanizm renderujący. Ten nowy program, noszący nazwę Simple3DTest2, tworzy trójwymiarową wypukłą bryłę, która ma przypominać dom. Z przodu bryły umieszczone są też dwa wielokąty, które mają imitować okno i drzwi, tak jak to zostało pokazane na rysunku 7.24.

Rysunek 7.24.

Ekrany programu Simple3DTest2, wzbogaconego o mechanizmy przycinania, usuwania tylnych powierzchni, nowy mechanizm renderowania i możliwość poruszania kamery



Program Simple3DTest2 umożliwia użytkownikowi swobodne wędrowanie wokół domu, spoglądanie w górę, w dół, w prawo, w lewo i chodzenie po całym obszarze, nawet wchodzenie do środka domu. Warto zauważyć, że jeśli wejdziemy do domu, to tak naprawdę nie zobaczymy nic, bo każdy z wielokątów tworzących dom będzie zwrócony tyłem do kamery.

Cały kod przykładu Simple3DTest2 jest zbyt długi, by go tutaj prezentować (znaczna jego część dotyczy po prostu tworzenia wielokątów), omówimy jednak tutaj jego najciekawsze elementy. Program Simple3DTest2 wykorzystuje klasę SolidPolygonRenderer w celu rysowania każdego z wielokątów znajdujących się na liście wielokątów:

```
public void draw(Graphics2D g) {
    // Rysuj wielokąty.
    polygonRenderer.startFrame(g);
    for (int i=0; i<polygons.size(); i++) {
        polygonRenderer.draw(g, (Polygon3D)polygons.get(i));
    }
    polygonRenderer.endFrame(g);

    drawText(g);
}
```

Program Simple3DTest2 wyświetla także niektóre instrukcje widoczne na ekranie i może również wyświetlać informacje o tempie renderowania kolejnych klatek:

```
private boolean drawFrameRate = false;
private boolean drawInstructions = true;

private int numFrames;
private long startTime;
private float frameRate;
```

```

    ...
    public void drawText(Graphics g) {
        // Rysuj tekst.
        g.setColor(Color.WHITE);
        if (drawInstructions) {
            g.drawString("Sterowanie ruchem: mysz/klawisze kurSORA. " +
                "Esc zamkna program.", 5, font_size);
        }
        // (Można też wyłączyć BufferStrategy w obiekcie
        // ScreenManager, by uzyskać dokładniejsze testy).
        if (drawFrameRate) {
            calcFrameRate();
            g.drawString(frameRate + " frames/sec", 5,
                screen.getHeight() - 5);
        }
    }

    public void calcFrameRate() {
        numFrames++;
        long currTime = System.currentTimeMillis();

        // Obliczaj szybkość renderowania klatek co 500 milisekund.
        if (currTime > startTime + 500) {
            frameRate = (float)numFrames * 1000 /
                (currTime - startTime);
            startTime = currTime;
            numFrames = 0;
        }
    }
}

```

Tempo wyświetlania klatek jest wyliczane na nowo w odstępie 500 milisekund. Polega to po prostu na podzieleniu liczby narysowanych klatek przez czas, który upłynął. Alternatywnym sposobem wyliczania tempa rysowania klatek jest przechowywanie raz wyliczonej wartości średniej. Ta metoda jednak nie pozwoli na uzyskanie dobrego obrazu tempa wyświetlania klatek, ponieważ może się ono zmieniać w trakcie działania programu — w zależności od tego, jak wiele renderowanych obiektów trzeba będzie narysować.

Transformacje tworzące wrażenie ruchu kamery są aktualizowane w metodzie `update()` programu `Simple3DTest2`:

```

public void update(long elapsedTime) {
    if (exit.isPressed()) {
        stop();
        return;
    }

    // Sprawdź opcje.
    if (largerView.isPressed()) {
        setViewBounds(viewWindow.getWidth() + 64,
            viewWindow.getHeight() + 48);
    }
}

```

```
else if (smallerView.isPressed()) {
    setViewBounds(viewWindow.getWidth() - 64,
                  viewWindow.getHeight() - 48);
}
if (frameRateToggle.isPressed()) {
    drawFrameRate = !drawFrameRate;
}

// Przechwyć czas elapsedTime.
elapsedTime = Math.min(elapsedTime, 100);

float angleChange = 0.0002f*elapsedTime;
float distanceChange = .5f*elapsedTime;

Transform3D camera = polygonRenderer.getCamera();
Vector3D cameraLoc = camera.getLocation();

// Zastosuj ruch.
if (goForward.isPressed()) {
    cameraLoc.x -= distanceChange * camera.getSinAngleY();
    cameraLoc.z -= distanceChange * camera.getCosAngleY();
}
if (goBackward.isPressed()) {
    cameraLoc.x += distanceChange * camera.getSinAngleY();
    cameraLoc.z += distanceChange * camera.getCosAngleY();
}
if (goLeft.isPressed()) {
    cameraLoc.x -= distanceChange * camera.getCosAngleY();
    cameraLoc.z += distanceChange * camera.getSinAngleY();
}
if (goRight.isPressed()) {
    cameraLoc.x += distanceChange * camera.getCosAngleY();
    cameraLoc.z -= distanceChange * camera.getSinAngleY();
}
if (goUp.isPressed()) {
    cameraLoc.y += distanceChange;
}
if (goDown.isPressed()) {
    cameraLoc.y -= distanceChange;
}

// Patrz w górę lub w dół (obrót dookoła osi x).
int tilt = tiltUp.getAmount() - tiltDown.getAmount();
tilt = Math.min(tilt, 200);
tilt = Math.max(tilt, -200);

// Ogranicz zakres spoglądania w górę i w dół.
float newAngleX = camera.getAngleX() + tilt * angleChange;
newAngleX = Math.max(newAngleX, (float)-Math.PI/2);
newAngleX = Math.min(newAngleX, (float)Math.PI/2);
camera.setAngleX(newAngleX);

// Obróć na boki (dookoła osi y).
int turn = turnLeft.getAmount() - turnRight.getAmount();
turn = Math.min(turn, 200);
turn = Math.max(turn, -200);
camera.rotateAngleY(turn * angleChange);
```

```
// Przechylanie głowy w prawo i w lewo (obrót dookoła osi z).
if (tiltLeft.isPressed()) {
    camera.rotateAngleZ(10*angleChange);
}
if (tiltRight.isPressed()) {
    camera.rotateAngleZ(-10*angleChange);
}
}
```

Metoda `update()` przystosowana jest do odbierania poleceń zarówno od myszy, jak i z klawiatury — w zależności od otrzymanych danych porusza odpowiednio kamerą. Korzysta między innymi z funkcji poruszania bohatera po scenerii za pomocą myszy (tzw. *mouselook*, którą można by nażwać żartobliwie „myszkowaniem” po scenerii), by umożliwić użytkownikowi poruszanie się po trójwymiarowym świecie gry, którą przygotowaliśmy w rozdziale 3., w którym omówiono problemy związane z interaktywnością i tworzeniem interfejsu użytkownika gry. Ponadto ogranicza kąt, pod jakim można spojrzeć w górę lub w dół, aby użytkownik nie mógł w ten sposób spojrzeć za plecy i po obróceniu kamery o 360 stopni z powrotem do przodu, jakby wykonywał salta.

Oto wszystkie opcje umożliwiające sterowanie tym przykładem:

Ruch myszy	Obrót w prawo i w lewo; spoglądanie w górę i w dół
Klawisze strzałek; klawisze <i>W, S, A, D</i>	Ruch do przodu, do tyłu, w lewo i w prawo
<i>Page up</i> i <i>Page down</i>	Ruch w górę i w dół
<i>Insert</i> i <i>Delete</i>	Pochylanie głowy w prawo i w lewo
+	Zwiększanie rozmiaru okna obrazu
-	Zmniejszanie rozmiaru okna obrazu
<i>R</i>	Wyświetlanie tempa renderowania klatek
<i>Esc</i>	Wyjście z programu

Klawisze *+/-*, umożliwiające zmianę rozmiarów okna obrazu, są szczególnie przydatne, jeśli program działa zbyt wolno, gdy okno obrazu zajmuje cały ekran. Można je wtedy zmniejszyć, by przyspieszyć działanie przykładu.

```
/**
 * Ustala granice okna obrazu, zawsze wyśrodkowanego względem
 * środka ekranu.
 */
public void setViewBounds(int width, int height) {
    width = Math.min(width, screen.getWidth());
    height = Math.min(height, screen.getHeight());
    width = Math.max(64, width);
    height = Math.max(48, height);
    viewWindow.setBounds((screen.getWidth() - width) / 2,
        (screen.getHeight() - height) / 2, width, height);
}
```

Kiedy uruchomimy ten przykładowy program, zauważymy bez trudu, że niektóre ściany są ciemniejsze niż inne — zostały one przycieniowane. Nie jest to oczywiście żaden sekretny mechanizm cieniający, który ukryłem przed Tobą — wielokątom tym po prostu ręcznie przypisałem ciemniejszy kolor. W kolejnym rozdziale zajmiemy się profesjonalnymi sposobami programowania oświetlenia i cieniowania.

Podsumowanie

Rozdział ten był krótkim wstępem do programowania grafiki trójwymiarowej. Będziemy jeszcze omawiać inne sprawy związane z grafiką trójwymiarową, takie jak mapowanie tekstuury, oświetlenie i zarządzanie sceną, niemniej zajmiemy się nimi już w kolejnych trzech rozdziałach.

Jak zwykle, omawialiśmy w tym rozdziale bardzo wiele spraw — rachunek wektorowy, wykorzystanie liczb stałoprzecinkowych, iloczyn skalarny i wektorowy wektorów, wektory normalne wielokątów, pojęcie okna obrazu, technikę usuwania tylnych powierzchni, transformacje, rzutowanie i przycinanie wielokątów, rysowanie za pomocą konwertera skanującego oraz wypełnianie wielokątów, że wspomnę tylko podstawowe zagadnienia. Ponadto wykorzystywaliśmy kod przygotowany w poprzednich rozdziałach, umożliwiający na przykład tworzenie grafiki na pełnym ekranie lub pobieranie danych z klawiatury oraz myszy (w tym poruszanie się po świecie za pomocą myszy). Przygotowaliśmy również podstawowy szkielet mechanizmu grafiki trójwymiarowej, z którego będziemy korzystać w następnych rozdziałach.

Rozdział 8.

Mapowanie tekstur i oświetlenie

W tym rozdziale:

- ◆ Podstawy mapowania tekstur uwzględniającego perspektywę.
- ◆ Prosty mechanizm mapowania tekstur.
- ◆ Optymalizowanie mapowania tekstur.
- ◆ Prosty mechanizm generowania oświetlenia.
- ◆ Implementowanie podświetlania tekstur.
- ◆ Tworzenie zaawansowanych trików oświetleniowych za pomocą map cieni.
- ◆ Dodatkowe pomysły.
- ◆ Podsumowanie.

Gdy byłem jeszcze małym chłopcem i pojawiły się pierwsze konsole Super Nintendo, wiedziałem, że muszę zdobyć taki zestaw. Konsole Super Nintendo były wielkim krokiem naprzód w stosunku do wcześniejszego standardu Nintendo Entertainment System. Miały lepszą grafikę dwuwymiarową, wspaniałe gry i oferowały praktycznie każdą atrakcję, jakimi mogły w owym czasie kusić gry komputerowe. Wkrótce nadeszły święta i rzeczywiście znalazłem zestaw Super Nintendo pod choinką.

Później wpadł do nas jeden ze znajomych mojej mamy ze swoim synkiem, który był o kilka lat młodszy ode mnie. Oczywiście był zbyt mały, aby być dobrym towarzyszem do zabawy, tak czy siak jednak spadł na mnie obowiązek zabawienia go. Włączyliśmy więc mój nowy zestaw Nintendo i zaczęliśmy grać w standardową grę wykorzystującą technikę przewijania warstw obrazu w dwóch wymiarach.

Wspaniałe efekty uzyskane za pomocą tej techniki sprawiły, że mój mały przyjaciel onieśmiał. Drzewa na pierwszym planie przesuwały się bardzo szybko, podczas gdy góry w tle wędrowały powoli, a przy tym wszystkim zachowane były ostre i realistyczne

tekstury. Nigdy przedtem czegoś takiego nie widział. Pokręcił z zachwytem głową, wółając: „Prawdziwa trójwymiarowa grafika!”.

Oczywiście nie była to tak naprawdę grafika trójwymiarowa. Był to po prostu specjalny trik, który symulował wrażenie głębi w scenie, która w istocie była dwuwymiarowa — było to jednak na tyle zręcznie zrobione, aby wywrzeć piorunujące wrażenie na małym chłopcu. Nie wyjaśniłem mu, że gra tak naprawdę nie była trójwymiarowa, ponieważ uświadomiłem sobie w tym momencie coś ważnego: gra zawsze tylko symuluje scenę trójwymiarową, bowiem obraz wyświetlany jest na dwuwymiarowej powierzchni ekranu komputera lub telewizora. Obraz jest zawsze dwuwymiarowy, rolą gry jest jednak symulowanie świata trójwymiarowego najlepiej, jak to tylko możliwe. Trzeci wymiar jest złudzeniem w oku beholdera¹.

W ten sposób przechodzimy do tematów, które będą poruszane w tym rozdziale: mapowania tekstur i oświetlenia. Przykłady z poprzedniego rozdziału, poświęconego podstawom grafiki trójwymiarowej, pozwalały co prawda tworzyć scenę, która miała głębie, ale wszystkie obiekty widoczne na ekranie tworzone były z prostych wielokątów wypełnionych jednolitym kolorem, które wyglądały dość zgrzebnie. W tym rozdziale wzbogacimy wielokąty o tekstury i cieniowanie, dzięki czemu nasza scena będzie wyglądała znacznie bardziej realistycznie. Pod koniec tego rozdziału przedstawię przykłady, po których uruchomieniu będziesz mógł bez obaw zawałać: „To prawdziwa trójwymiarowa grafika!”

Przede wszystkim zajmiemy się tworzeniem programowego mechanizmu mapowania tekstur i implementowaniu oświetlenia za pomocą mapy cieni, techniki popularyzowanej przez grę *Quake*. Mimo pewnych wad programowy mechanizm renderowania języka Java jest pod pewnymi względami lepszy od sprzętowych technik renderowania. Chodzi między innymi o niezależność od powiązanych z konkretnym sprzętem interfejsów API oraz możliwość zignorowania przestarzałych lub zawierających błędy sterowników grafiki trójwymiarowej. Ponadto własnoręczne przygotowanie mechanizmu mapowania tekstur dla grafiki trójwymiarowej jest czystą przyjemnością. Później być może wykorzystasz w grach sprzętowy mechanizm renderujący, by maksymalnie poprawić szybkość tworzenia grafiki trójwymiarowej, niemniej techniki, które przedstawię w tym rozdziale, pozwalają na dogłębne zrozumienie (i kontrolowanie) zasad działania każdego mechanizmu renderującego, niezależnie od tego, czy jest to mechanizm sprzętowy czy programowy.

Podstawy mapowania tekstur uwzględniającego perspektywę

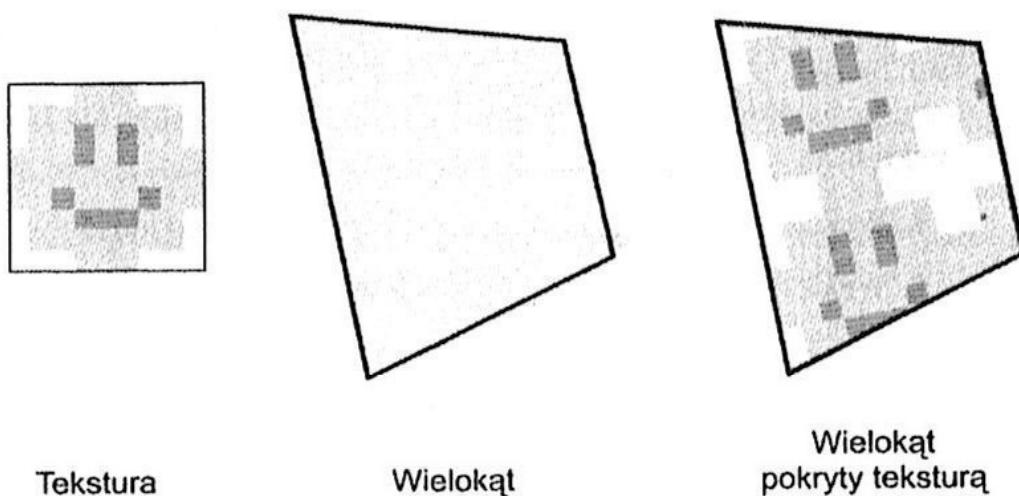
W podrozdziale tym zajmować się będziemy tworzeniem mechanizmu mapowania tekstur, który będzie dawał wyniki uwzględniające wymagania perspektywy, czyli sprawiającego, że zmniejszające się wraz z odległością wielokąty pokryte były również odpowiednio zmniejszonymi teksturami.

¹ Odwołanie do popularnej serii gier „Eye of beholder”. Beholder (po ang. również „strażnik”) to nazwa dziwacznego potwora ze świata gry, strzegącego skarbów przed graczami — przyp. tłum.

Jak wszyscy wiemy, tekstura może być mniejsza od wielokąta, na który jest mapowana. W takim przypadku tekstura zostanie ułożona na wielokącie wielokrotnie, tak jak kafelki na ścianie. Pokazuje to rysunek 8.1.

Rysunek 8.1.

Jeśli mapujemy teksturę mniejszą od wielokąta, zostaje ona wielokrotnie rozłożona na jego powierzchni



Piksel w tekstuze określa się zazwyczaj słowem *tekSEL* (od angielskich słów „texture element” — element tekstuzy). Należy pamiętać, że tekstyty będą skalowane w zależności od tego, w jakiej odległości kamera będzie od wielokąta, a wraz z nimi skalowane są i teksele. Tak więc tekSEL bardzo rzadko ma takie same rozmiary jak piksel w oknie obrazu.

Mapowanie tekstuzy można wykonywać na dwa sposoby: albo mapować każdy tekSEL na piksel, albo odwrotnie — każdy piksel na tekSEL. Mapowanie każdego tekseLA na piksel wymaga wykonania znacznie większej pracy niż wart będzie osiągnięty w ten sposób efekt, ponieważ nie każdy z tekseLI tworzących tekstuzy będzie rysowany (tekstaTA może być na przykład widoczna pod zbyt ostrym kątem lub znajdować się zbyt daleko). Ponadto same tekseLE można obracać i skalować w zależności od orientacji tekstuzy. Zasadniczo, jeśli stosujemy technikę mapowania tekseLI na pikseLE, konieczne jest rysowanie dla każdego tekseLA jednolicie wypełnionego wielokąta; podczas rysowania wykonuje się wtedy sporo nadmiarowej pracy (niektóre pikseLE trzeba rysować więcej niż raz).

Z kolei jeśli mapujemy pikseLE na tekseLE, każdy piksel w oknie obrazu jest z założenia rysowany tylko raz. Innymi słowy: dla każdego piksela obrazu znajdujemy odpowiedni tekSEL, który trzeba w tym miejscu narysować.

Pamiętając o tym wszystkim, zgadniesz bez trudu, że dobrym miejscem na wykonywanie mapowania tekstuzy będzie uczynienie tego w potoku renderowania zaraz po przełożeniu wielokąta na skany linii — podobnie, jak postępowaliśmy w przypadku jednolicie wypełnionych wielokątów w poprzednim rozdziale.

Wprowadzenie równań wykorzystywanych do mapowania tekstuzy

Mogłbym w tym miejscu po prostu przedstawić Ci równania powszechnie używane do mapowania tekstuzy (które można bez problemu znaleźć w internecie). Niemniej zdecydowałem się pokazać również, skąd te równania się wzięły, żeby zaspokoić ciekawość

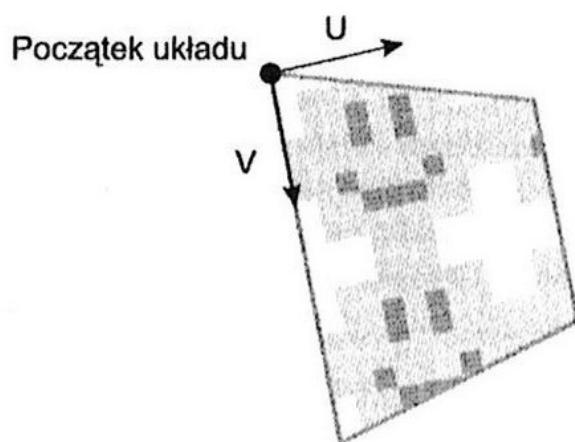
bardziej dociekiwych Czytelników. Zaczniemy najpierw od wprowadzenia pewnych elementów matematyki wektorów. Ci, którzy jej nie znają, powinni przejrzeć krótkie wprowadzenie, które prezentowałem w poprzednim rozdziale.

Po pierwsze zdefiniujmy, jak tekstura będzie zorientowana względem wielokąta. Zależy nam na tym, aby tekstura była odpowiednio przesuwana lub obracana wewnątrz wielokąta, co wykonyuje się za pomocą następujących wektorów (przedstawionych na rysunku 8.2):

- ◆ Wektora O , definiującego początek wewnętrznego układu współrzędnych tekstuury.
- ◆ Wektora U , reprezentującego oś x w układzie współrzędnych tekstuury.
- ◆ Wektora V , reprezentującego oś y w układzie współrzędnych tekstuury.

Rysunek 8.2.

Orientację tekstuury wewnątrz wielokąta ustala się za pomocą wektora początku układu oraz dwóch wektorów kierunku



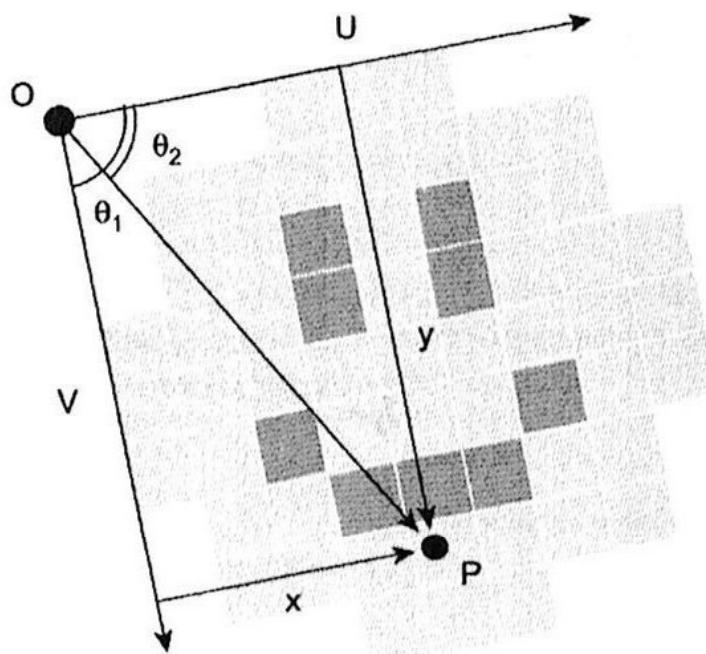
Przy zastosowaniu takiego rozwiązania tekstuura może być zwrócona praktycznie w dowolnym kierunku. Ponadto początek układu współrzędnych tekstuury nie musi być wcale umieszczany w obrębie wielokąta.

Konieczne jest jednak ustalenie kilku zasad. Wektory U i V muszą być względem siebie *prostopadłe* (czyli jeden do drugiego musi być ustawiony pod kątem prostym), bowiem nie zależy nam raczej na nakładaniu na powierzchnię zniekształconych tekstuur. Ponadto należy zadbać, aby wektory U i V były położone w tej samej płaszczyźnie co wielokąt. W przeciwnym bowiem razie wektory U i V „wystawałyby” z wielokąta, a my otrzymalibyśmy dziwne efekty w rodzaju tekstuury nachylonej pod kątem, gdy patrzy się na wielokąt od przodu.

Teraz przyjrzyjmy się wektorom O , U i V bliżej i zobaczymy, jak mają się one do tego, co zamierzamy zrobić. Jak pamiętamy, chcielibyśmy mapować każdy piksel obrazu na odpowiedni teksel tekstuury. Zanim przystąpimy do ustalania, który teksel w danym miejscu narysować, trzeba będzie oczywiście ustalić, jak współrzędne punktu P na powierzchni wielokąta mają się do współrzędnych punktu w oknie obrazu W . Pomińmy na razie ten krok i założmy, że mamy już punkt P na powierzchni wielokąta i chcielibyśmy dla niego znaleźć współrzędne odpowiedniego teksela tekstuury (x, y) — tak, jak to zostało pokazane na rysunku 8.3.

Rysunek 8.3.

Jak punkt P na powierzchni wielokąta ma się do współrzędnych tekstuury



Przyjrzałszy się rysunkowi 8.3, bez trudu możemy przygotować równania umożliwiające wyliczenie współrzędnych x i y wewnątrz tekstuur:

$$x = |P - O| \cos \theta_2$$

$$y = |P - O| \cos \theta_1$$

W przedstawionych tu równaniach $|P - O|$ to odległość od początku układu O do punktu P , θ_1 to kąt między wektorami $(P - O)$ a V , a θ_2 to kąt między wektorami $(P - O)$ i U .

Idźmy krok dalej: jeśli wektory U i V są wektorami jednostkowymi, to obliczenia te można uprościć, korzystając z iloczynu skalarnego wektorów:

$$x = U \bullet (P - O)$$

$$y = V \bullet (P - O)$$

Pamiętaj przy tym, że współrzędne (x, y) tekstuury mogą być ujemne lub mogą nie znajdować się w obrębie tekstuury. Teksturę będziemy bowiem układać na wielokącie jak kafelki.

Teraz wróćmy do poprzedniego kroku i zobaczymy, jak dla określonego punktu P na powierzchni wielokąta odnajduje się wyświetlany na ekranie punkt W na powierzchni okna obrazu (rysunek 8.4). Wiemy już, który z wielokątów znajdzie się w miejscu oznaczonym przez punkt W , ponieważ we wcześniejszej fazie mapowania tekstuury wielokąt został już rzutowany na okno obrazu.

Patrząc na problem od przeciwej strony, można powiedzieć, że punkt W jest po prostu efektem rzutowania punktu P na okno obrazu, który to problem rozwiązuje się za pomocą standardowego zestawu równań przedstawionych w rozdziale 7.:

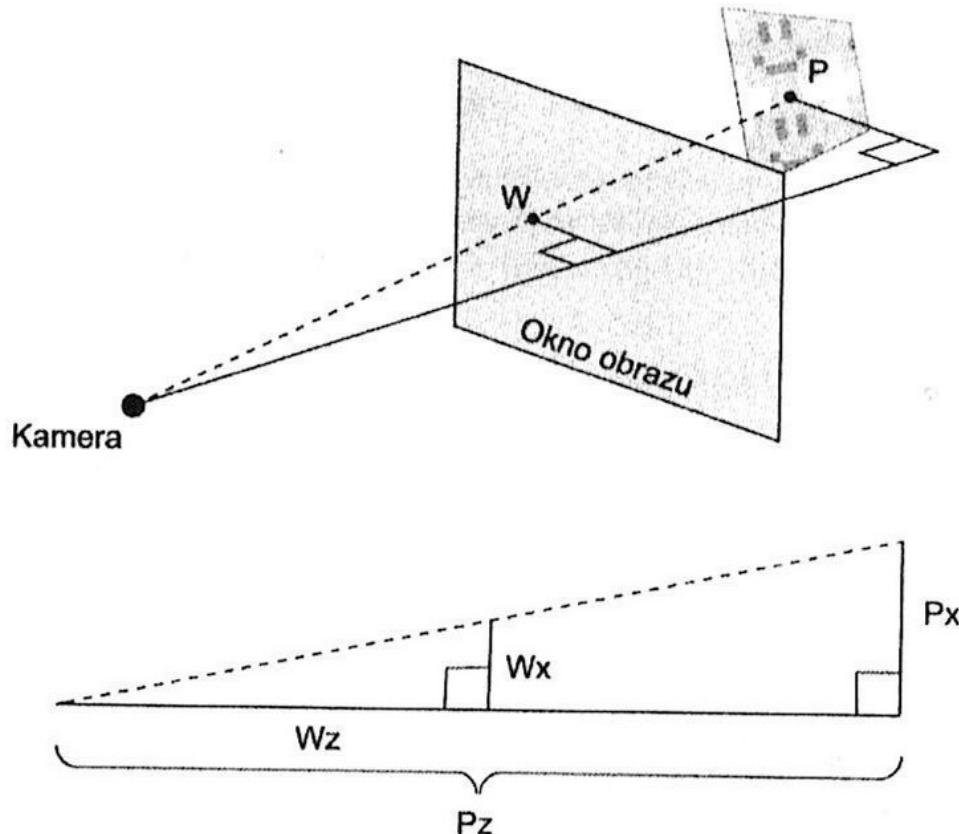
$$W_x = d P_x / P_z$$

$$W_y = d P_y / P_z$$

$$W_z = d P_z / P_z$$

Rysunek 8.4.

Rzutowanie punktu P
z wielokąta na punkt
okna obrazu — W



Współrzędna W_z jest więc równa d , czyli odległości między kamerą a oknem obrazu. Przekształcając to na postać wektorową, otrzymamy:

$$W = (d/P_z) P$$

Ponieważ chcemy znaleźć punkt P , trzeba rozwiązać to równanie dla P :

$$P = (P_z/d) W$$

Tak więc, żeby znaleźć punkt P , potrzebna nam jest znajomość współrzędnej P_z , która jest odlegością od kamery do wielokąta po osi z .

Aby znaleźć odległość P_z , potrzebować będziemy informacji o płaszczyźnie tworzonej przez wielokąt. Należy w tym celu użyć równania płaszczyzny, które zasadniczo informuje po prostu, że wektor normalny N tej płaszczyzny i dowolna linia położona w tej płaszczyźnie będą do siebie prostopadłe. My użyjemy linii $(P - O)$ z rysunku 8.3, ponieważ wektor ten leży w tej samej płaszczyźnie co wielokąt. Tak więc do dzieła:

$$N \bullet (P - O) = 0$$

$$N \bullet O = N \bullet P$$

A oto poprzednie równanie pozwalające wyliczyć P , rozwiążane dla P_z :

$$N \bullet O = N \bullet ((P_z/d) W)$$

$$N \bullet O = (P_z/d) (N \bullet W)$$

$$P_z = d(N \bullet O) / (N \bullet W)$$

Teraz podstawiamy P_z do poprzedniego równania i otrzymujemy wartość P :

$$P = W(N \bullet O) / (N \bullet W)$$

Stąd:

$$x = U \bullet (W(N \bullet O) / (N \bullet W) - O)$$

oraz

$$y = V \bullet (W(N \bullet O) / (N \bullet W) - O)$$

Mamy już równania umożliwiające mapowanie tekstur. Niemniej są to dość złożone równania. Na szczęście można je trochę uprościć, wykorzystując pewne zależności występujące w matematyce wektorów. Zaczniemy od współrzędnej x :

$$x = U \bullet (W(N \bullet O) - O(N \bullet W)) / (N \bullet W)$$

Najpierw zastosujemy prawo mówiące, jaki jest wynik iloczynu wektorowego trzech wektorów: $A \times B \times C = B(A \bullet C) - C(A \bullet B)$:

$$x = U \bullet (N \times W \times O) / (N \bullet W)$$

Teraz skorzystamy z przemienności iloczynu skalarnego względem wektorowego dla trzech wektorów: $A \bullet B \times C = A \times B \bullet C$:

$$x = (U \times N \times W \bullet O) / (N \bullet W)$$

Jeśli założymy, że $U \times N = -V$ oraz, że $N = U \times V$ (czyli że wektor normalny wielokąta i dwa wektory orientujące układ współrzędnych tekstuury są prostopadłe do siebie nawzajem), to otrzymamy:

$$x = (-V \times W \bullet O) / (U \times V \bullet W)$$

I wreszcie na koniec:

$$x = (V \times O \bullet W) / (U \times V \bullet W)$$

$$y = (O \times U \bullet W) / (U \times V \bullet W)$$

Te równania są już znacznie prostsze, ale to jeszcze nie koniec uproszczeń, które możemy wprowadzić. Zauważmy, że kiedy wykonujemy mapowanie tekstuury dla konkretnego wielokąta, jedynym elementem, który się wtedy zmienia, jest wektor W . Wektory O , U i V pozostają niezmienne. Dzięki temu możemy już na początku wyliczyć dla wielokąta iloczyny wektorowe tych trzech wektorów i potem używać tylko tych wyliczonych wartości:

$$A = V \times O$$

$$B = O \times U$$

$$C = U \times V$$

Tak więc podczas wykonywania mapowania tekstuury dla konkretnego wielokąta trzeba już tylko rozwiązywać takie proste równania:

$$x = A \bullet W / C \bullet W$$

$$y = B \bullet W / C \bullet W$$

Hej! To jest to — właśnie wyprowadziliśmy wygodne równania mapowania tekstuur. Teraz wystarczy je tylko wykorzystać przy tworzeniu mechanizmu mapowania tekstuur.

Prosty mechanizm mapowania tekstur

Po pierwsze konieczne jest opisanie orientacji tekstury. W tym celu przygotujemy klasę Rectangle3D, przedstawioną na listingu 8.1, która jest po prostu czworokątem dowolnie zorientowanym w przestrzeni trójwymiarowej. Innymi słowy, nie musi być zorientowana względem osi układu współrzędnych, tak jak klasa Rectangle2D w java.awt.geom.

Listing 8.1. Rectangle3D.java

```
package com.brackeen.javagamebook.math3D;

/**
 * Klasa Rectangle3D jest prostokątem w przestrzeni 3D, zdefiniowanym za pomocą
 * początku lokalnego układu i wektorów wskazujących w kierunku jego podstawy
 * (szerokość) i boku (wysokość).
 */
public class Rectangle3D implements Transformable {

    private Vector3D origin;
    private Vector3D directionU;
    private Vector3D directionV;
    private Vector3D normal;
    private float width;
    private float height;

    /**
     * Tworzy prostokąt w określonym punkcie początkowym i o szerokości
     * oraz wysokości równych zero.
     */
    public Rectangle3D() {
        origin = new Vector3D();
        directionU = new Vector3D(1.0, 0, 0);
        directionV = new Vector3D(0, 1.0, 0);
        width = 0;
        height = 0;
    }

    /**
     * Tworzy nowy prostokąt Rectangle3D o podanym początku
     * i kierunkach podstawy(directionU) oraz boku
     * (directionV).
     */
    public Rectangle3D(Vector3D origin, Vector3D directionU,
                      Vector3D directionV, float width, float height)
    {
        this.origin = new Vector3D(origin);
        this.directionU = new Vector3D(directionU);
        this.directionU.normalize();
        this.directionV = new Vector3D(directionV);
        this.directionV.normalize();
        this.width = width;
        this.height = height;
    }
}
```

```
/**  
 * Przypisuje wartościom tego prostokąta Rectangle3D  
 * inny podany prostokąt Rectangle3D.  
 */  
public void setTo(Rectangle3D rect) {  
    origin.setTo(rect.origin);  
    directionU.setTo(rect.directionU);  
    directionV.setTo(rect.directionV);  
    width = rect.width;  
    height = rect.height;  
}  
  
/**  
 * Pobiera początek tego prostokąta Rectangle3D.  
 */  
public Vector3D getOrigin() {  
    return origin;  
}  
  
/**  
 * Pobiera kierunek podstawy tego prostokąta Rectangle3D.  
 */  
public Vector3D getDirectionU() {  
    return directionU;  
}  
  
/**  
 * Pobiera kierunek boku tego prostokąta Rectangle3D.  
 */  
public Vector3D getDirectionV() {  
    return directionV;  
}  
  
/**  
 * Pobiera szerokość tego prostokąta Rectangle3D.  
 */  
public float getWidth() {  
    return width;  
}  
  
/**  
 * Definiuje szerokość tego prostokąta Rectangle3D.  
 */  
public void setWidth(float width) {  
    this.width = width;  
}  
  
/**  
 * Pobiera wysokość tego prostokąta Rectangle3D.  
 */  
public float getHeight() {  
    return height;  
}
```

```
/***
 * Definiuje wysokość tego prostokąta Rectangle3D.
 */
public void setHeight(float height) {
    this.height = height;
}

/***
 * Wylicza wektor normalny tego prostokąta Rectangle3D.
 */
protected Vector3D calcNormal() {
    if (normal == null) {
        normal = new Vector3D();
    }
    normal.setToCrossProduct(directionU, directionV);
    normal.normalize();
    return normal;
}

/***
 * Pobiera wektor normalny tego prostokąta Rectangle3D.
 */
public Vector3D getNormal() {
    if (normal == null) {
        calcNormal();
    }
    return normal;
}

/***
 * Ustawia wektor normalny tego prostokąta Rectangle3D.
 */
public void setNormal(Vector3D n) {
    if (normal == null) {
        normal = new Vector3D(n);
    }
    else {
        normal.setTo(n);
    }
}

public void add(Vector3D u) {
    origin.add(u);
    // Nie dokonuj translacji wektorów kierunku ani rozmiaru.
}

public void subtract(Vector3D u) {
    origin.subtract(u);
    // Nie dokonuj translacji wektorów kierunku ani rozmiaru.
}
```

```
public void add(Transform3D xform) {
    addRotation(xform);
    add(xform.getLocation());
}

public void subtract(Transform3D xform) {
    subtract(xform.getLocation());
    subtractRotation(xform);
}

public void addRotation(Transform3D xform) {
    origin.addRotation(xform);
    directionU.addRotation(xform);
    directionV.addRotation(xform);
}

public void subtractRotation(Transform3D xform) {
    origin.subtractRotation(xform);
    directionU.subtractRotation(xform);
    directionV.subtractRotation(xform);
}

}
```

Klasa prostokąta Rectangle3D po prostu przechowuje informacje o trzech wektorach definiujących układ współrzędnych tekstury (wektorze początku układu, wektorze *U* i wektorze *V*) oraz o wektorze normalnym tego prostokąta.

Teraz przygotujemy prostą klasę SimpleTexturedPolygonRenderer, przedstawioną na listingu 8.2. Jest to dość prosta podklasa klasy PolygonRenderer, którą tworzyliśmy w rozdziale 7. — dodaje ona po prostu równania umożliwiające mapowanie tekstur.

Listing 8.2. SimpleTexturedPolygonRenderer.java

```
package com.brackeen.javagamebook.graphics3D;

import java.awt.*;
import java.awt.image.*;
import java.io.File;
import java.io.IOException;
import javax.imageio.ImageIO;
import com.brackeen.javagamebook.math3D.*;

/**
 * Ta klasa (SimpleTexturedPolygonRenderer) demonstruje
 * podstawy mapowania tekstur z zachowaniem perspektywy.
 * Działa jednak bardzo powoli i mapuje na każdym wielokącie tę samą teksturę.
 */
public class SimpleTexturedPolygonRenderer extends PolygonRenderer
{
    protected Vector3D a = new Vector3D();
    protected Vector3D b = new Vector3D();
    protected Vector3D c = new Vector3D();
    protected Vector3D viewPos = new Vector3D();
    protected Rectangle3D textureBounds = new Rectangle3D();
    protected BufferedImage texture;
```

```
public SimpleTexturedPolygonRenderer(Transform3D camera,
    ViewWindow viewWindow, String textureFile)
{
    super(camera, viewWindow);
    texture = loadTexture(textureFile);
}

/**
 * Ładuje obraz tekstury z pliku. Obraz ten będzie mapowany
 * na wszystkich wielokątach.
 */
public BufferedImage loadTexture(String filename) {
    try {
        return ImageIO.read(new File(filename));
    }
    catch (IOException ex) {
        ex.printStackTrace();
        return null;
    }
}

protected void drawCurrentPolygon(Graphics2D g) {

    // Wylicz granice tekstury.
    // Granice te powinny być już wyliczone i przechowywane
    // wraz z wielokątem. Współrzędne wyliczamy jednak w celach
    // demonstracyjnych.
    Vector3D textureOrigin = textureBounds.getOrigin();
    Vector3D textureDirectionU = textureBounds.getDirectionU();
    Vector3D textureDirectionV = textureBounds.getDirectionV();

    textureOrigin.setTo(sourcePolygon.getVertex(0));

    textureDirectionU.setTo(sourcePolygon.getVertex(3));
    textureDirectionU.subtract(textureOrigin);
    textureDirectionU.normalize();

    textureDirectionV.setTo(sourcePolygon.getVertex(1));
    textureDirectionV.subtract(textureOrigin);
    textureDirectionV.normalize();

    // Transformuj granice tekstury.
    textureBounds.subtract(camera);

    // Rozpocznij wyliczenia związane z mapowaniem tekstury.
    a.setToCrossProduct(textureBounds.getDirectionV(),
        textureBounds.getOrigin());
    b.setToCrossProduct(textureBounds.getOrigin(),
        textureBounds.getDirectionU());
    c.setToCrossProduct(textureBounds.getDirectionU(),
        textureBounds.getDirectionV());

    int y = scanConverter.getTopBoundary();
    viewPos.z = -viewWindow.getDistance();
```

```
while (y <= scanConverter.getBottomBoundary()) {
    ScanConverter.Scan scan = scanConverter.getScan(y);

    if (scan.isValid()) {
        viewPos.y =
            viewWindow.convertFromScreenYToViewY(y);
        for (int x = scan.left; x <= scan.right; x++) {
            viewPos.x =
                viewWindow.convertFromScreenXToViewX(x);

            // Wylicz lokalizację tekstury.
            int tx = (int)(a.getDotProduct(viewPos) /
                c.getDotProduct(viewPos));
            int ty = (int)(b.getDotProduct(viewPos) /
                c.getDotProduct(viewPos));

            // Pobierz kolor rysowania.
            try {
                int color = texture.getRGB(tx, ty);

                g.setColor(new Color(color));
            }
            catch (ArrayIndexOutOfBoundsException ex) {
                g.setColor(Color.red);
            }

            // Narysuj piksel
            g.drawLine(x, y, x, y);
        }
    }
    y++;
}
```

Ta klasa wykorzystuje do przechowywania tekstury klasę buforowanego obrazu `BufferedImage`. Podczas procesu renderowania tworzy dla każdego piksela nowy obiekt `Color`. Ponieważ klasa `Graphics2D` nie zawiera metody umożliwiającej rysowanie piksela `drawPixel()`, wykorzystujemy tutaj jej metodę `drawLine()`, by narysować linię o długości 1 piksela. Należy przy tym pamiętać, że nasz mechanizm renderujący nie układa tekstury na wielokącie. Aby nie było pustych powierzchni, tekstura musi być rozmiarów wielokąta. Układaniem tekstur na wielokątach zajmiemy się w kolejnym podrozdziale, na razie jednak taki mechanizm renderowania powinien nam wystarczyć. Teraz pora przygotować prosty test, który umożliwi nam wypróbowanie tego mechanizmu.

Klasa `TextureMapTest1`, przedstawiona na listingu 8.3, tworzy prosty wielokąt i rysuje go na ekranie. Użytkownik może swobodnie wędrować naokoło tego wielokąta.

Listing 8.3. *TextureMapTest1.java*

```
import com.brackeen.javagamebook.math3D.*;
import com.brackeen.javagamebook.graphics3D.*;
import com.brackeen.javagamebook.test.GameCore3D;
```

```

public class TextureMapTest1 extends GameCore3D {

    public static void main(String[] args) {
        new TextureMapTest1().run();
    }

    public void createPolygons() {
        Polygon3D poly;

        // Na razie tylko jedna ściana.
        poly = new Polygon3D(
            new Vector3D(-128, 256, -1000),
            new Vector3D(-128, 0, -1000),
            new Vector3D(128, 0, -1000),
            new Vector3D(128, 256, -1000));
        polygons.add(poly);
    }

    public void createPolygonRenderer() {
        viewWindow = new ViewWindow(0, 0,
            screen.getWidth(), screen.getHeight(),
            (float) Math.toRadians(75));

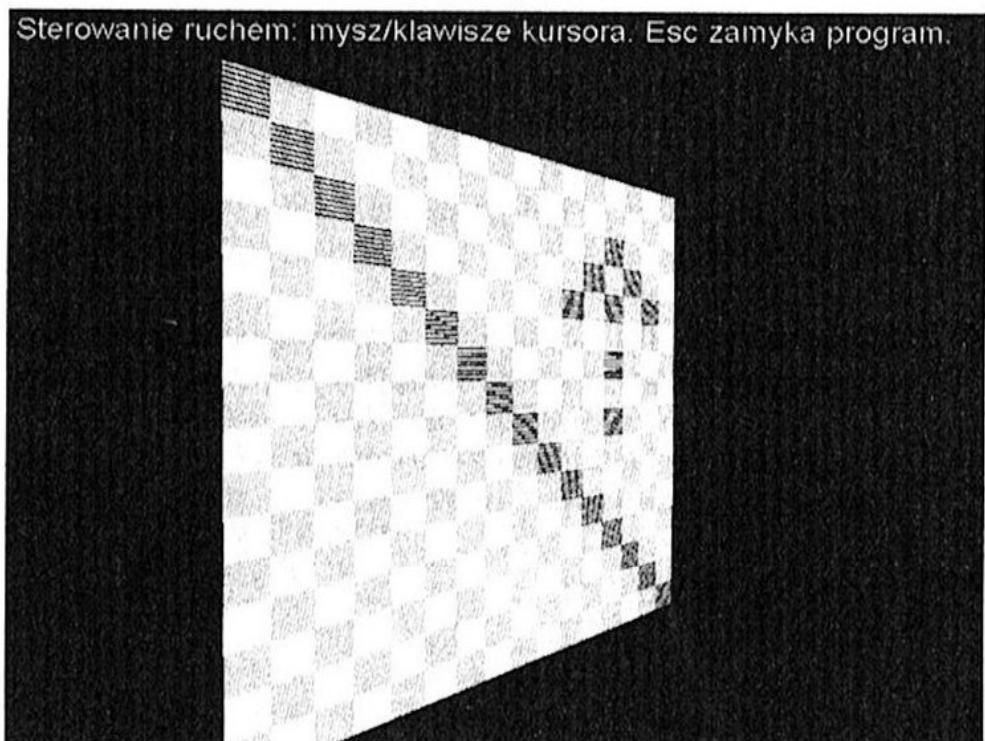
        Transform3D camera = new Transform3D(0, 100, 0);
        polygonRenderer = new SimpleTexturedPolygonRenderer(
            camera, viewWindow, "../images/test_pattern.png");
    }
}

```

Klasa TextureMapTest1 rozszerza możliwości klasy GameCore3D, która jest bardzo podobna do klasy Simple3DTest2, przedstawionej w poprzednim rozdziale. Przykład TextureMapTest1 po prostu tworzy wielokąt i obiekt mechanizmu renderującego oraz pozwala użytkownikowi poruszać się po stworzonym świecie. Rysunek 8.5 pokazuje, jak wygląda efekt działania tego programu.

Rysunek 8.5.

Przykład
TextureMapTest1
rysuje na powierzchni
wielokąta testową
teksturę



Tekstura wykorzystywana w przykładzie TextureMapTest1 jest prostym asymetrycznym wzorem, który pozwala sprawdzić, czy tekstura została prawidłowo mapowana na wielokąt. Dzięki asymetryczności wzoru łatwo jest ustalić, która strona tekstury jest dołem, a która góra — można więc bez trudu sprawdzić, czy poprawnie zastosowaliśmy równania. Jeśli strzałka widoczna na tekstuze byłaby skierowana w złym kierunku lub obraz byłby zamieniony stronami, wiedzielibyśmy, że w równaniach jest jakiś błąd lub że gdzieś pomyliliśmy znak.

Wady naszego prostego mechanizmu renderującego

Po uruchomieniu przykładu łatwo zauważyc, że działa on przeraźliwie wolno. Narysowanie pojedynczej klatki może zajmować nawet ponad sekundę, w zależności od szybkości komputera, na którym przykład jest uruchamiany. Dzieje się tak z kilku powodów:

- ◆ Tworzenie dla każdego piksela nowego obiektu `Color` jest zbytnim obciążeniem. Ponadto w ten sposób również mechanizm oczyszczania pamięci Javy ma więcej pracy.
- ◆ Wywoływanie metod `setColor()` i `drawLine()` osobno dla każdego piksela również bardzo spowalnia pracę programu.
- ◆ Metoda `getRGB()` klasy `BufferedImage` musi konwertować piksel na kolor 24-bitowy, jeśli kolor piksela nie został wcześniej zdefiniowany właśnie w tym formacie.
- ◆ Podczas rysowania pojedynczego piksela wykonujemy zdecydowanie zbyt wiele obliczeń.
- ◆ Przechwytywanie wyrzuconego wyjątku `ArrayIndexOutOfBoundsException` jest wolniejsze niż ręczne sprawdzanie granic tablicy. Niemniej w tym przypadku nie jest to aż tak wielki problem, ponieważ został przygotowany test, który pozwala uniknąć tych wyjątków.

Warto zauważyć, że żaden z wymienionych tu problemów nie jest sam z siebie powodem opóźnienia i w normalnych sytuacjach nie będzie spowalniał programu — nie musimy się obawiać, że tworzenie obiektów `Color` lub przechwytywanie wyjątków będzie zawsze znaczco spowalniać działanie grafiki. W większości przypadków rozwiązania te sprawdzają się całkiem dobrze. Tutaj wszystko działa wolno, ponieważ wykonujemy te operacje dla każdego piksela ekranu i pragniemy wyświetlać wiele klatek obrazu na sekundę. Aby umożliwić Ci wyobrażenie sobie rozmiarów pracy, którą musi wykonać komputer, powiem tylko, że dla ekranu o rozdzielczości 640 na 480 i przy prędkości 60 klatek na sekundę w każdej sekundzie będzie musiało zostać narysowane 18 432 000 pikseli. Dlatego właśnie tak ważny jest wybór najszybszej możliwej metody rysowania pojedynczego piksela i dlatego właśnie należy zrezygnować z technik programowania, które w innej sytuacji byłyby jak najbardziej na miejscu.

Optymalizowanie mapowania tekstur

Niestety interfejs Graphics2D nie zawiera wystarczająco szybko działającego mechanizmu rysowania różnokolorowych pikseli linia po linii. Co więcej, język Java nie udostępnia w tym momencie interfejsu pozwalającego na bezpośredni dostęp do pamięci wideo. Teoretycznie można by to było osiągnąć za pomocą bufora ByteBufer, wskazującego bezpośrednio do pamięci wideo, nadal jednak skazani będziemy na powolną technikę rysowania piksela po pikselu, stosowaną przez interfejs JNI.

Aby się z tym uporać, należy najpierw narysować piksele w buforze BufferedImage, a dopiero potem przenieść jego zawartość na ekran. Przypomina to ideę podwójnego buforowania, opisaną w rozdziale 2. (poświęconym grafice i animacji dwuwymiarowej). Korzystając z buforowanych obrazów BufferedImage, będziemy mogli pobierać dane z obrazu zapisanego jako tablica i kopiować dane opisujące piksel wprost do tego piksela. Kopiowanie elementu tablicy jest jedną z operacji, które komputerowi zajmują najmniej czasu.

Wadą tego rozwiązania jest jednak dodatkowy czas, potrzebny na skopiowanie obrazu do pamięci wideo za pomocą odwołania do metody Graphics.drawImage(). Wywołanie to zużywa około 5 % do 10 % czasu procesora na komputerach, które testowałem. Niemniej, mimo to, korzyści z zastosowania tej metody przeważają nad kosztami.

Trzeba jeszcze podjąć decyzję, jaką głębię koloru zastosować w obrazie BufferedImage. Oczywiście w obrazie tym powinna być wykorzystana taka sama głębia koloru i taki sam układ pikseli, jak w obrazie widocznym na ekranie. Dzięki temu będzie możliwa po prostu skopiować bity buforowanego obrazu na ekran bez konieczności wykonywania po drodze żadnej konwersji. Dlatego też należy wybrać właściwą, 8-, 16- lub 24-bitową głębię kolorów. Jeśli zastosujemy kolory 8-bitowe, to nieuchronnie ograniczeni będziemy do bardzo wąskiej palety kolorów, która znacznie wpłynie na możliwości tworzenia takich efektów, jak na przykład cieniowanie. Kolory 24-bitowe (lub nawet 32-bitowe) z kolei dadzą nam lepszą jakość obrazu, niemniej grafika będzie wtedy wolniejsza niż w przypadku kolorów 16-bitowych, ponieważ program będzie musiał obsługiwać znacznie więcej danych. Kolory 16-bitowe mają gorszą jakość niż kolory 24-bitowe, niemniej są w zupełności wystarczające dla gry z trójwymiarową grafiką i mapowaniem tekstur oraz efektami oświetlenia. Przy założeniu, że korzystamy z 16-bitowego obrazu BufferedImage, tablicę zawierającą obraz można pobrać w następujący sposób:

```
BufferedImage doubleBuffer;
short[] doubleBufferData;
...
// Pobierz dane z bufora.
DataBuffer dest = doubleBuffer.getRaster().getDataBuffer();
doubleBufferData = ((DataBufferUshort)dest).getData();
```

Teraz omówimy formaty przechowywania tekstur.

Przechowywanie tekstur

Dobrze by było, aby — podobnie jak podwójny bufor — również nasze tekstury zachowywały tę samą głębokość kolorów i format pikseli co obraz wyświetlany na ekranie. Moglibyśmy przechowywać tekstury w obiektach `BufferedImage`, tak jak postępowaliśmy w przypadku tekstur wykorzystywanych w pierwszym przykładzie. Tak naprawdę jednak nie potrzebujemy wszystkich możliwości, które oferuje klasa `BufferedImage`. Przygotujemy więc zamiast tego abstrakcyjną klasę `Texture`, zaprezentowaną na listingu 8.4, która po prostu umożliwia kodowi wywołującemu ją pobranie koloru znajdującego się w miejscu określonym wewnętrznymi współrzędnymi (x, y) tekstury.

Listing 8.4. *Texture.java*

```
/**  
 * Klasa Texture jest klasą abstrakcyjną, reprezentującą  
 * teksturę o 16-bitowych kolorach.  
 */  
public abstract class Texture {  
  
    protected int width;  
    protected int height;  
  
    /**  
     * Tworzy nową teksturę Texture o podanej szerokości i wysokości.  
     */  
    public Texture(int width, int height) {  
        this.width = width;  
        this.height = height;  
    }  
  
    /**  
     * Pobiera szerokość tej tekstury Texture.  
     */  
    public int getWidth() {  
        return width;  
    }  
  
    /**  
     * Pobiera wysokość tej tekstury Texture.  
     */  
    public int getHeight() {  
        return height;  
    }  
  
    /**  
     * Pobiera 16-bitowy kolor tej tekstury Texture w podanym  
     * punkcie  $(x,y)$ .  
     */  
    public abstract short getColor(int x, int y);  
}
```

Klasa Texture jest klasą abstrakcyjną, więc aby możliwe było korzystanie z niej, należy ją rozwinąć (przygotowując odpowiedni kod). Metoda `getColor()` musi zostać zaimplementowana w ten sposób, aby zwracała 16-bitową wartość koloru, którą można znaleźć w tekurze pod podaną lokacją (x, y) .

Należy pamiętać, że w razie potrzeby powinniśmy mieć możliwość wielokrotnego rozłożenia tekstury na wielokącie. Można to zrobić w dość prosty sposób, jeśli zadbane, aby szerokość i wysokość każdej z tekstur była jakąś kolejną potęgą liczby 2, taką jak na przykład: 16, 32, 128, itp. Powiedzmy, że mamy na przykład teksturę o szerokości 32 pikseli. Liczbę 32 zapisuje się binarnie (w 8-bitach) w następujący sposób:

00100000

Ta liczba pomniejszona o 1 (czyli 31) tworzyć będzie tak zwaną maskę. W przypadku kolejnych potęg liczby 2 odjęcie od danej liczby 1 jest równoważne operacji polegającej na zamianie bitowej 1 na 0, a następnych zer na jedynki. Tak więc nasza maska będzie miała następującą postać binarną:

00011111

Maska ta reprezentuje wszystkie możliwe bity, które mogą pojawić się w tym zakresie liczb — w naszym przypadku wszystkie możliwe bity, które mogą się pojawić w dowolnej współrzędnej x tekstury. Jeśli wykonamy na współrzędnej i masce bitową operację AND (za pomocą operatora `&`), to odetniemy niepotrzebne nam pierwsze bity i otrzymamy wartość współrzędnej mieszczącej się w przedziale od 0 do 31. Należy przy tym pamiętać, że ten trik działa tylko dla kolejnych potęg liczby 2. Pomyśl ten został wykorzystany w klasie tekstury `PowerOf2Texture` przedstawionej na listingu 8.5.

Listing 8.5. PowerOf2Texture.java

```

    {
        super(1 << widthBits, 1 << heightBits);
        this.buffer = buffer;
        this.widthBits = widthBits;
        this.heightBits = heightBits;
        this.widthMask = getWidth() - 1;
        this.heightMask = getHeight() - 1;
    }

    /**
     * Pobiera 16-bitowy kolor piksela w punkcie (x,y)
     * mapy bitowej.
     */
    public short getColor(int x, int y) {
        return buffer[
            (x & widthMask) +
            ((y & heightMask) << widthBits)];
    }
}

```

Klasa PowerOf2Texture przechowuje tablicę short[], zawierającą teksturę oraz maski szerokości i wysokości tekstury. Metoda getColor() wykorzystuje te maski do pobierania właściwych wartości x i y, dzięki czemu można łatwo układać tekstury na wielokącie.

Na koniec potrzebować będziemy łatwego sposobu na pobieranie danych z buforowanych obrazów BufferedImage, by móc tworzyć tekstury. W tym celu dodamy do klasy Texture statyczną metodę createTexture(). W tej chwili tworzy ona tylko tekstury PowerOf2Texture, w przyszłości rozwiniemy ją jednak, by mogła tworzyć również i inne typy tekstur.

```

    /**
     * Tworzy teksturę Texture z podanego obrazu.
     */
    public static Texture createTexture(BufferedImage image, boolean shaded){
        int type = image.getType();
        int width = image.getWidth();
        int height = image.getHeight();

        if (!isPowerOfTwo(width) || !isPowerOfTwo(height)) {
            throw new IllegalArgumentException(
                "Rozmiar tekstury musi być potęgą liczby 2.");
        }

        // Konwertuj obraz na obraz o 16-bitowych kolorach.
        if (type != BufferedImage.TYPE USHORT_565_RGB) {
            BufferedImage newImage = new BufferedImage(
                image.getWidth(), image.getHeight(),
                BufferedImage.TYPE USHORT_565_RGB);
            Graphics2D g = newImage.createGraphics();
            g.drawImage(image, 0, 0, null);
            g.dispose();
            image = newImage;
        }
    }

```

```

        DataBuffer dest = image.getRaster().getDataBuffer();
        return new PowerOf2Texture(
            ((DataBufferUShort)dest).getData(),
            countbits(width-1), countbits(height-1));
    }
}

```

W tej funkcji zakładamy, że wyświetlany 16-bitowy obraz będzie miał ten sam format pikseli, co typ `TYPE USHORT_565_RGB` klasy `BufferedImage`. Jak dotąd nie spotkałem 16-bitowego systemu wyświetlania, który nie miałby takiego formatu pikseli, niemniej teoretycznie taka sytuacja jest możliwa. Alternatywnie można też użyć metody `createCompatibleImage()` klasy `GraphicsConfiguration`, by przygotować za jej pomocą obraz kompatybilny z systemem wyświetlania.

Następnie przygotujemy nową klasę `PolygonRenderer`, którą nazwiemy `FastTexture-PolygonRenderer`. Ta nowa klasa będzie zawierała takie same procedury przygotowujące do renderowania (wyliczające z wyprzedzeniem wektory A , B i C), jakie miała klasa `SimpleTexturedPolygonRenderer`. Dodatkowo przygotowywać będzie tablicę po- dwojnego bufora `doubleBufferData`, do której będziemy kopiować piksele. Pod koniec każdej klatki bity z tego bufora będą kopowane na ekran komputera.

Klasa ta zawiera również wewnętrzną klasę, noszącą nazwę `ScanRenderer`. Klasa `Scan-Renderer` jest klasą abstrakcyjną zawierającą metody pozwalające na rysowanie poziomego skanu wielokąta.

```

/**
 * Abstrakcyjna klasa ScanRenderer jest wewnętrzną klasą
 * FastTexturedPolygonRenderer, dostarczającą interfejsu umożliwiającego
 * renderowanie poziomej linii skanu.
 */
public abstract class ScanRenderer {

    protected Texture currentTexture;

    public void setTexture(Texture texture) {
        this.currentTexture = texture;
    }

    public abstract void render(int offset,
        int left, int right);
}

```

W zależności od techniki rysowania będziemy tworzyć różne obiekty `ScanRenderer`.

Prosta optymalizacja

Najprostsza wersja renderera skanów `ScanRenderer`, którą będziemy tworzyć, wykonywać będzie tylko procedurę wypełniającą wielokąt. Tak się składa, że jest również wersją najszybszą (niestety właściwie nic nie robi).

```

public void render(int offset, int left, int right) {
    for (int x=left; x<=right; x++) {
        doubleBufferData[offset++] = (short)0x0007;
    }
}

```

Teraz, gdy już mamy ogólne pojęcie o renderowaniu skanów wielokąta, przygotujemy renderer ScanRenderer, który wykonywać będzie takie same obliczenia, jak SimpleTexturedPolygonRenderer:

```
public void render(int offset, int left, int right) {
    for (int x=left; x<=right; x++) {
        int tx = (int)(a.getDotProduct(viewPos) /
                      c.getDotProduct(viewPos));
        int ty = (int)(b.getDotProduct(viewPos) /
                      c.getDotProduct(viewPos));
        doubleBufferData[offset++] =
            currentTexture.getColor(tx, ty);
        viewPos.x++;
    }
}
```

Pstryk i mamy znacznie szybszy mechanizm mapowania tekstu! Teraz kopiowanie danych piksela z tekstury do podwójnego bufora odbywa się znacznie szybciej. Testowałem ten renderer skanów na dwóch różnych komputerach, podjeżdżając kamerą możliwie blisko wielokąta, aby zajmował on cały ekran o rozdzielczości 640×480. Na komputerze z procesorem Pentium 4 i zegarem 2,4 GHz renderowanie wykonywane było 46 razy szybciej, a na procesorze G4 z zegarem 867 MHz aż 144 razy szybciej. Całkiem niezły wynik. Już udało się nam znacznie zwiększyć prędkość renderowania, a może być jeszcze lepiej. Na pozór wszystkie obliczenia wyglądają na proste: tylko kilka operacji mnożenia i parę operacji dzielenia — nic trudnego. Niemniej należy pamiętać, że takie obliczenia trzeba wykonać dla każdego piksela na ekranie.

W tym miejscu można zastosować jedną z prostszych metod optymalizacji, która polega na przemieszczeniu najbardziej pracochłonnego kodu poza pętlę. W tym przypadku nie musimy tak naprawdę wyliczać iloczynów skalarnych dla każdego pojedynczego piksela, ponieważ wektory a , b i c nie będą się zmieniać. Ponadto wiemy, że wartość viewPos będzie wzrastać o 1 dla każdego kolejnego piksela, tak więc jesteśmy w stanie przewidywać, w jaki sposób iloczyny skalarne będą się zmieniać. Oto kolejna wersja renderera ScanRenderer, która wykorzystuje obie wspomniane tu techniki optymalizacji:

```
public void render(int offset, int left, int right) {
    float u = a.getDotProduct(viewPos);
    float v = b.getDotProduct(viewPos);
    float z = c.getDotProduct(viewPos);
    float du = a.x;
    float dv = b.x;
    float dz = c.x;
    for (int x=left; x<=right; x++) {
        doubleBufferData[offset++] =
            currentTexture.getColor(
                (int)(u/z), (int)(v/z));
        u+=du;
        v+=dv;
        z+=dz;
    }
}
```

W porównaniu z poprzednią wersją renderera ta przyspiesza działanie kodu 1,6 raza na komputerze z procesorem Pentium i 1,4 raza na komputerze z procesorem G4.

W kolejnym kroku można skorzystać ze sztuczki wykorzystywanej w konwerterze skanującym w poprzednim rozdziale: użyć liczb całkowitych (tzw. stałoprzecinkowych) w miejsce liczb zmienoprzecinkowych. Konwertowanie każdego piksela z liczby zmienoprzecinkowej na całkowitą byłoby jednak kosztowne, więc zamiast tego będziemy cały czas używać liczb całkowitych:

```
public static final int SCALE_BITS = 12;
public static final int SCALE = 1 << SCALE_BITS;

...
public void render(int offset, int left, int right) {
    int u = (int)(SCALE * a.getDotProduct(viewPos));
    int v = (int)(SCALE * b.getDotProduct(viewPos));
    int z = (int)(SCALE * c.getDotProduct(viewPos));
    int du = (int)(SCALE * a.x);
    int dv = (int)(SCALE * b.x);
    int dz = (int)(SCALE * c.x);
    for (int x=left; x<=right; x++) {
        doubleBufferData[offset++] =
            currentTexture.getColor(u/z, v/z);
        u+=du;
        v+=dv;
        z+=dz;
    }
}
```

Procesor Pentium 4 dość dobrze radzi sobie z operacjami na liczbach zmienoprzecinkowych, więc zastosowanie liczb stałoprzecinkowych niczego specjalnie tu nie przyspieszy. Natomiast w przypadku procesora G4 prędkość renderowania zwiększy się jeszcze 1,2 raza.

W tym miejscu docieramy prawie do granic optymalizacji, których nie da się przekroczyć bez korzystania z technik sprzętowych. Jedyne, co można tutaj jeszcze zrobić, to dodać parę odwołań do tablic oraz kilka operacji dodawania i dzielenia na liczbach stałoprzecinkowych.

Jeśli przyjrzeć się uważnie kodowi, łatwo dostrzec główne wąskie gardło: dwie operacje dzielenia. Dodawanie wykonuje się stosunkowo szybko, niemniej dzielenie zajmuje już więcej cykli zegara.

Jednak te dzielenia nie są wcale konieczne, prawda? Można się ich bez trudu pozbyć, bez widocznego pogorszenia jakości mapowania tekstur.

W tym właśnie tkwi problem: pogorszenie jakości mapowania tekstury. Na szczęście można pozbyć się dzielenia tak, aby pogorszenie jakości było praktycznie niezauważalne dla oka, a jednocześnie by udało się nam poprawić wydajność.

Pomysł polega na tym, aby wyliczać właściwe współrzędne tekstury tylko dla co któregoś z ciągu pikseli, a dla wszystkich pozostałych interpolować wartości na dwóch bazie sąsiednich pikseli, dla których kolor został wyliczony prawidłowo. W matematyce *interpolacja* oznacza zgadywanie wartości znajdujących się między dwiema wartościami, które są nam dokładnie znane. W kolejnym wariantie renderera ScanRenderer będziemy wyliczać prawidłowe współrzędne tekstury tylko dla co szesnastego piksela (lub czasem gęściej, jeśli w skanie jest mniej niż 16 pikseli). W ten sposób między dwoma prawidłowo wyliczonymi pikselami pozostałe można będzie wyliczać, wykonując tylko parę operacji dodawania i przesuwania bitów.

```
public class PowerOf2TextureRenderer extends ScanRenderer {

    public void render(int offset, int left, int right) {
        PowerOf2Texture texture =
            (PowerOf2Texture)currentTexture;
        float u = SCALE * a.getDotProduct(viewPos);
        float v = SCALE * b.getDotProduct(viewPos);
        float z = c.getDotProduct(viewPos);
        float du = INTERP_SIZE * SCALE * a.x;
        float dv = INTERP_SIZE * SCALE * b.x;
        float dz = INTERP_SIZE * c.x;
        int nextTx = (int)(u/z);
        int nextTy = (int)(v/z);
        int x = left;
        while (x <= right) {
            int tx = nextTx;
            int ty = nextTy;
            int maxLength = right-x+1;
            if (maxLength > INTERP_SIZE) {
                u+=du;
                v+=dv;
                z+=dz;
                nextTx = (int)(u/z);
                nextTy = (int)(v/z);
                int dtx = (nextTx-tx) >> INTERP_SIZE_BITS;
                int dty = (nextTy-ty) >> INTERP_SIZE_BITS;
                int endOffset = offset + INTERP_SIZE;
                while (offset < endOffset) {
                    doubleBufferData[offset++] =
                        texture.getColor(
                            tx >> SCALE_BITS, ty >> SCALE_BITS);
                    tx+=dtx;
                    ty+=dty;
                }
                x+=INTERP_SIZE;
            }
        else {
            // Zmienny rozmiar obszaru interpolowanego.
            int interpSize = maxLength;
            u += interpSize * SCALE * a.x;
            v += interpSize * SCALE * b.x;
            z += interpSize * c.x;
            nextTx = (int)(u/z);
            nextTy = (int)(v/z);
            int dtx = (nextTx-tx) / interpSize;
            int dty = (nextTy-ty) / interpSize;
```

```

        int endOffset = offset + interpSize;
        while (offset < endOffset) {
            doubleBufferData[offset++] =
                texture.getColor(
                    tx >> SCALE_BITS, ty >> SCALE_BITS);
            tx+=dtx;
            ty+=dty;
        }
        x+=interpSize;
    }

}

```

Na moich dwóch testowych komputerach zastosowanie tej optymalizacji dało efekt 1,6-krotnego zwiększenia prędkości w przypadku procesora Pentium 4 i 1,4-krotnego przyspieszenia w przypadku procesora G4.

Jak wspomniałem wcześniej, ta wersja renderera ScanRenderer pogarsza trochę jakość obrazu, niemniej w większości przypadków będzie to dla użytkownika niezauważalne. Efekt ten widoczny jest tylko wtedy, gdy głębokość wielokąta znacznie zmienia się wzdłuż poziomej linii skanu. Jeśli więc spoglądamy na wielokąt od przodu, głębokość wielokąta (położenie względem osi z) nie będzie się zmieniać wraz z poruszaniem się wzdłuż linii skanu i końcowy rezultat będzie wyglądał całkiem dobrze. Podobnie podłogi nie będą wyglądać źle, bo tu głębokość zmienia się z dołu do góry, a nie z lewej do prawej. Niemniej, gdy patrzyć będziemy na wielokąt pod ostрыm kątem (jak w przypadku bocznych ścian), głębokość będzie się znacznie zmieniać w linii poziomej i tekstury mogą wyglądać na odrobinę zniekształcone. Jest to szczególnie ważne w przypadku ekranów o małej rozdzielczości, gdzie odległość 16 pikseli oznacza dość duży dystans.

Sposobem na zmniejszenie tego zniekształcenia jest korzystanie ze zmiennej odległości między granicznymi pikselami będącymi bazą interpolacji, w zależności od tego, jak szybko zmienia się głębokość wielokąta w linii poziomej. Jeśli głębokość ta się nie zmienia lub zmienia nieznacznie, to odległość między precyzyjnie wyliczonymi pikselami może być większa, natomiast w przypadku znaczniejszych zmian głębokości powinna być mniejsza. Tę optymalizację pozostawiam Ci jako zadanie domowe.

Rozwijanie metod w miejscu wywołania

Istnieje jeszcze jedna optymalizacja warta wspomnienia: rozwijanie metod w miejscu wywołania. Wywoływanie metody pociąga za sobą pewne koszty, a my przecież wywołujemy dla każdego piksela metodę `texture.getColor()`. Szczęśliwie maszyna wirtualna *HotSpot* (dołączana domyślnie do wersji 1.4 Javy) jest na tyle inteligentna, że potrafi w oczywistych sytuacjach rozwijać kod metod w miejscu wywołania. Rozwijanie kodu metody w miejscu wywołania polega po prostu na przeniesieniu kodu z metody i umieszczeniu go w miejscu jej wywołania, co eliminuje koszty związane z wywoływaniem metody. *HotSpot* rozwija przede wszystkim te krótkie metody, które rozpozna jako metody niewirtualne. (Jak pamiętasz, metoda niewirtualna to taka, w przypadku której nie ma podklas, w których metoda może zostać przesłonięta). W przypadku

tekstury Texture tylko klasa PowerOf2Texture implementuje metodę getColor() i metoda ta jest na tyle krótka, by maszyna wirtualna HotSpot mogła rozwijać jej kod w miejscu wywołania.

W dalszej części tego rozdziału będziemy jednak korzystać z innych podklas klasy Texture, takich jak na przykład ShadedTexture. Jeśli będziemy mieli inną klasę, która również implementuje metodę getColor(), HotSpot nie będzie rozwijać metody w miejscu wywołania i będziemy zmuszeni wywoływać ją na nowo dla każdego piksela.

Próbowałem skłonić maszynę wirtualną HotSpot podstępem do rozwijania metod w miejscu wywołania, deklarując finalną lokalną zmienną w nadziei, że HotSpot rozpozna klasę finalnej zmiennej lokalnej i będzie mogła dzięki temu rozwijać metodę w miejscu wywołania:

```
final Texture texture = currentTexture;
```

Ponieważ ta lokalna zmienna jest zmienną finalną, jej klasa nigdy się nie zmienia — sądziłem więc, że maszyna wirtualna HotSpot będzie w stanie teraz rozwijać kod metody getColor(). Gdyby ten trik zadziałał, byłoby nieźle. Niestety okazało się, że nie działa. W tym momencie nie wiem jeszcze, dlaczego tak się dzieje, co oznacza, że musimy szukać innego rozwiązania.

Rozwiązanie, które będzie w tym przypadku działać, polega na przygotowaniu różnych rendererów ScanRendererer dla każdego z typów tekstur. Kod każdego z renderera będzie dokładnie taki sam, za wyjątkiem lokalnej zmiennej Texture, której zadaniem jest bezpośrednie odwoływanie się do określonego typu tekstuury.

```
public class PowerOf2TextureRenderer extends ScanRendererer {  
  
    public void render(int offset, int left, int right) {  
        PowerOf2Texture texture = (PowerOf2Texture)currentTexture;  
        ...  
        // tutaj kod mapujący teksturę  
    }  
}
```

W przypadku renderera PowerOf2TextureRenderer maszyna wirtualna HotSpot będzie „wiedziała”, że tekstura będzie typu PowerOf2Texture — będzie możliwe dzięki temu odpowiednie rozwinięcie kodu metody getColor(). Nie jest to najlegantniejsze rozwiązanie z punktu widzenia solidnego programowania obiektowego — jeśli zmienimy jeden z rendererów ScanRendererer, będzie konieczna zmiana kodu wszystkich — ma jednak tę zaletę, że działa. Kolejnym minusem jest konieczność deklarowania klasy PowerOf2Texture (oraz każdej innej klasy z podklas klasy Texture, która będzie miała własny renderer ScanRendererer) jako klasy finalnej, aby żadna inna klasa nie mogła uczyć się z niej swojej podklasy, uniemożliwiając w ten sposób maszynie wirtualnej HotSpot rozwijanie kodu metody.

Czasami niestety trzeba pracować z tym, czym się dysponuje — to jest właśnie jedna z takich sytuacji. Programiści piszący w asemblerze oraz językach takich, jak C czy C++ bywają czasami zmuszeni do robienia szalonych rzeczy, aby zoptymalizować kod dla potrzeb określonego urządzenia lub systemu operacyjnego. Podobnie programiści Javy czasem zmuszeni są do popełniania takich szaleństw dla potrzeb maszyn wirtualnych.

Na dwóch komputerach, na których testowałem kod, optymalizacja polegająca na rozwijaniu kodu metod w miejscu wywołania przyspieszyła działanie kodu 2,3 raza dla komputera z procesorem Pentium 4 i 1,3 raza na komputerze z procesorem G4. W sumie, w porównaniu z oryginalnym powolnym mechanizmem mapowania tekstur, nasz końcowy mechanizm renderowania działa mniej więcej 284 razy szybciej dla procesora Pentium 4 i 436 razy szybciej dla procesora G4. Niezły wynik!

Przykładowy program korzystający z szybkiego mapowania tekstur

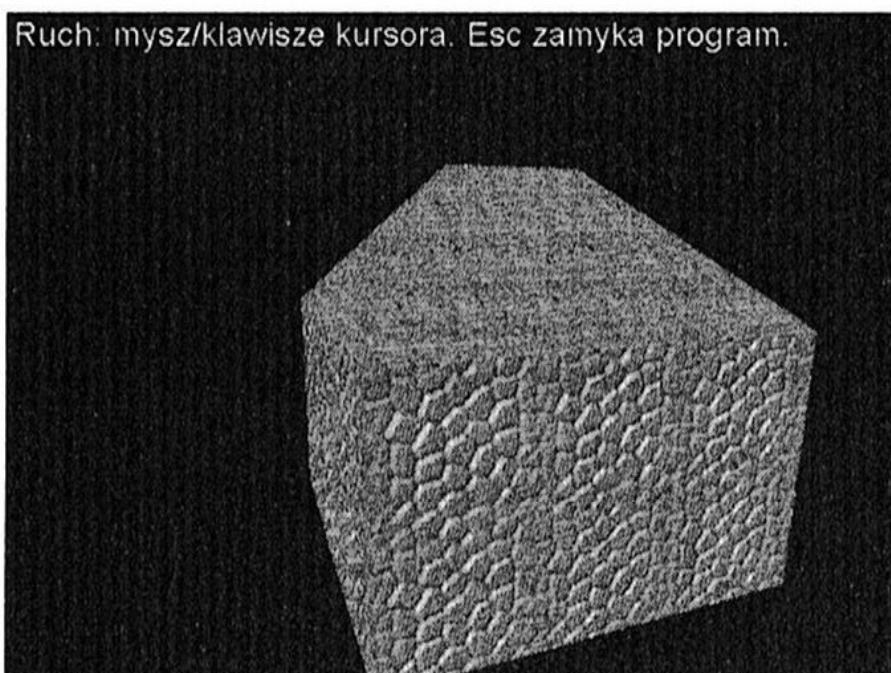
W porządku, skoro mamy już klasę tekstur Texture i renderer FastTexturePolygon-Renderer, możemy przygotować przykładowy program. Program TextureMapTest2, którego ekran został przedstawiony na rysunku 8.6, tworzy wypukłą bryłę, pokrytą ze wszystkich stron teksturą.

Rysunek 8.6.

Przykład

TextureMapTest2 rysuje bryłę, na której bokach mapujemy tekstury, znacznie szybciej niż wykonywał to program TextureMapTest1

Ruch: mysz/klawisze kurSORA. Esc zamyka program.



Przykład TextureMapTest2 wykorzystuje klasę TexturedPolygon3D, która jest prawie taka sama, jak zwykła klasa Polygon3D — z tą tylko różnicą, że ma specjalne pola do przechowywania tekstury wielokąta i jej granic.

Kod źródłowy przykładu TextureMapTest2 po prostu tworzy zestaw wielokątów. Jedyną wartą wspomnienia metodą w programie TextureMapTest2 jest metoda setTexture():

```
public void setTexture(TexturedPolygon3D poly,
    Texture texture)
{
    Vector3D origin = poly.getVertex(0);

    Vector3D dv = new Vector3D(poly.getVertex(1));
    dv.subtract(origin);

    Vector3D du = new Vector3D();
    du.setToCrossProduct(poly.getNormal(), dv);
```

```

        Rectangle3D textureBounds = new Rectangle3D(origin, du, dv,
            texture.getWidth(), texture.getHeight());

        poly.setTexture(texture, textureBounds);
    }
}

```

Metoda ta tworzy granice tekstury dla wielokąta. Skorzystanie z niej zwalnia nas od konieczności ręcznego tworzenia granic tekstury dla każdego wielokąta.

Dobrą wiadomością jest to, że program TextureMapTest2 jest znacznie, ale to znacznie szybszy od poprzedniego przykładu! Jak zwykle, wciśnięcie R pozwala zobaczyć, z jaką prędkością klatki obrazu renderowane są na naszym komputerze. Teraz pora wzbogacić nasz przykładowy program, dodając do niego odrobinę cieniowania.

Prosty mechanizm generowania oświetlenia

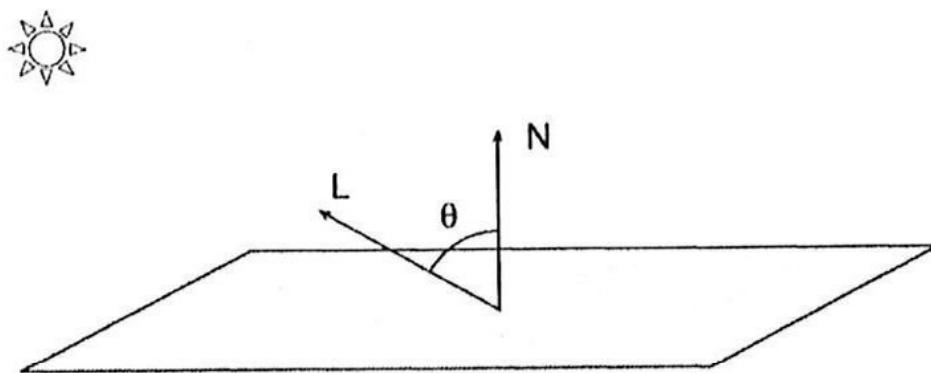
Podrozdział ten będzie poświęcony dodawaniu do wielokątów realistycznie wyglądającego cieniowania. Ostatni przykład w poprzednim rozdziale prezentował trójwymiarowy domek, stworzony z wielokątów pomalowanych jednolitymi kolorami. Boczne ściany domu były pomalowane ciemniejszymi kolorami niż jego front — to był nasz prosty sposób na symulowanie cieniowania wielokątów. Teraz zajmiemy się wprowadzeniem prawdziwego cieniowania, wykorzystującego obecne w scenerii źródła światła zamiast ręcznie kolorowanych wielokątów.

Odbicie rozproszone

Będziemy tutaj korzystać z rodzaju oświetlenia zwanego *odbiciem rozproszonym* (ang. *diffuse reflection*), korzystającym z prawa Lambert'a. Prawo to mówi, że intensywność odbitego światła zależy od kąta między wektorem normalnym powierzchni a kierunkiem źródła światła, co pokazuje rysunek 8.7.

Rysunek 8.7.

Odbicie rozproszone ustala stopień oświetlenia wielokąta w zależności od kąta między wektorem normalnym wielokąta a kierunkiem źródła światła



Im mniejszy będzie ten, kąt, tym więcej światła odbijać będzie wielokąt i tym jaśniejszy powinien się on wydawać obserwatorowi. I odwrotnie, przy większym kącie wielokąt będzie odbijał mniej światła i wydawać się będzie ciemniejszy. Zakładając, że wektor

normalny wielokąta i wektor wskazujący kierunek źródła światła są znormalizowane, prawo to można wyrazić matematycznie za pomocą następującego równania (litera i oznacza tu intensywność odbijanego światła):

$$i = N \bullet L$$

W prawdziwym świecie przykładem odbicia rozproszonego może być zamknięte pomieszczenie z jednym źródłem światła, na przykład ciemny pokój, w którym ściany i sufit mają ten sam kolor, a źródłem światła jest pojedyncza żarówka zawieszona pod sufitem. Jeśli przyjrzymy się powierzchniom w rogu między sufitem a ścianami, to zauważymy, że powierzchnie w pobliżu rogu będą ciemniejsze niż reszta ściany. Dzieje się tak dlatego, że ściany oświetlone są światłem padającym na wprost, podczas gdy sufit oświetlony jest światłem padającym pod znacznie ostrzejszym kątem. Dlatego sufit, choć jest w tym samym kolorze co ściany, będzie się nam wydawał ciemniejszy.

Światło otoczenia

Ponadto w trakcie dnia możemy w pokoju nie potrzebować żadnego światła, ponieważ przez okno będzie wpadać światło słoneczne, odbijające się od wszystkich powierzchni i oświetlające cały pokój. Modelowanie niezależnego światła, które oświetla pokój, odbijając się od wszystkich powierzchni, byłoby zbyt trudne obliczeniowo, dlatego też będziemy symulować ten efekt za pomocą tzw. światła otaczającego (ang. *ambient lighting*). Światło otaczające wyznacza po prostu pewien minimalny stopień oświetlenia każdego obiektu. Dla przykładu w oświetlonym słońcem pokoju można ustalić minimalną intensywność światła na 0,7. Można to dość łatwo przedstawić w równaniu, dodając po prostu czynnik a , odpowiadający światłu otaczającemu:

$$i = a + N \bullet L$$

Intensywność światła otaczającego może przyjmować wartości z przedziału od 0 do 1.

Uwzględnianie intensywności światła pochodzącego ze źródła światła

Na podobnej zasadzie, na jakiej definiujemy intensywność światła otaczającego, przypiswaną potem każdemu wielokątowi, możemy również ustalić intensywność każdego ze źródeł światła wykorzystywanych w scenie. Efekt zmiany intensywności źródła światła będzie taki, jak zamiana żarówki z 60-watowej na 100-watową. Jeśli oznaczymy intensywność źródła światła jako l , to nasze równanie intensywności oświetlenia przyjmie następującą postać:

$$i = a + l(N \bullet L)$$

Spadek intensywności światła wraz z odległością

Możemy nasze modelowanie światła uczynić jeszcze bardziej realistycznym i sprawić, aby intensywność światła malała wraz z oddaleniem od źródła światła. Dzięki temu, im dalej obiekt będzie się znajdował od źródła światła, tym słabszym światłem będzie

oświetlony. W kolejnych równaniach literą f będziemy oznaczać odległość od źródła światła, w której do obiektu nie będzie już docierać żadne światło wysyłane przez źródło (odległość wygasania). Litera I oznacza, jak poprzednio, intensywność źródła światła, a litera d — odległość od obiektu do źródła światła. Teraz nasze równania oświetlenia będą wyglądać tak:

$$I' = I (f - d) / (f + d)$$
$$i = a + I' (N \bullet L)$$

Implementowanie punktowego źródła światła

Punktowe źródło światła o ustalonej intensywności i opcjonalnie definiowanej odległości wygasania światła zdefiniowane zostało w klasie PointLight3D, przedstawionej na listingu 8.6.

Listing 8.6. PointLight3D.java

```
package com.brackeen.javagamebook.math3D;

/**
 * Klasa PointLight3D to punktowe źródło światła o ustalonej
 * intensywności(miedzy 0 a 1) i - opcjonalnie - odległości wygasania,
 * która pozwala na osłabianie światła wraz z odlegością.
 */
public class PointLight3D extends Vector3D {

    public static final float NO_DISTANCE_FALLOFF = -1;

    private float intensity;
    private float distanceFalloff;

    ...

    /**
     * Pobiera intensywność światła z tego źródła w
     * określonej odległości od niego.
     */
    public float getIntensity(float distance) {
        if (distanceFalloff == NO_DISTANCE_FALLOFF) {
            return intensity;
        }
        else if (distance >= distanceFalloff) {
            return 0;
        }
        else {
            return intensity * (distanceFalloff - distance)
                / (distanceFalloff + distance);
        }
    }
}
```

Klasa PointLight3D jest wektorem Vector3D, zawierającym dane o intensywności źródła światła i odległości definiującej zanikanie światła wraz z odległością od źródła. Metoda getIntensity() tej klasy pobiera intensywność światła pochodzącego z określonego źródła światła w określonej odległości od tego źródła.

Implementowanie oświetlania tekstur

Problem, który spróbujemy tutaj rozwiązać, to implementowanie oświetlenia dla tekstur. Jak pamiętamy, intensywność oświetlenia wielokąta może wawać się w przedziale od 0 do 1. Zastanówmy się teraz, jak zastosować tę intensywność na tekstuze. Wartość 1 powinna pozostawać teksturom taką, jaka jest, a wartości mniejsze niż 1 powinny modyfikować teksturom, tak by wraz ze zmniejszaniem się intensywności oświetlenia stopniowo stawała się coraz ciemniejsza, aż do wartości 0, która powinna czynić teksturom prawie czarną.

Nie można przy tym po prostu pomnożyć 16-bitowej wartości koloru przez intensywność oświetlenia, ponieważ 16-bitowa wartość koloru jest specjalnie upakowaną strukturą danych. Prawdę powiedziawszy, trzeba z niej wybrać osobno wartości kolorów czerwonego, zielonego i niebieskiego, następnie pomnożyć każdą z tych wartości z osobna przez intensywność oświetlenia, po czym z powrotem konwertować zmodyfikowane wartości kolorów składowych na 16-bitową wartość koloru. Jak łatwo zauważyc, to trochę zbyt wiele pracy, jeśli wykonujemy ją dla każdego piksela.

Alternatywnym rozwiązaniem może być ograniczenie możliwych wartości intensywności oświetlenia do, powiedzmy, 64 różnych wartości i po prostu ładowanie 64 różnych wersji każdej teksturom, od najciemniejszej do najjaśniejszej, po czym każdorazowe wybieranie odpowiedniej z 64 teksturom dla wielokąta, w zależności od intensywności oświetlenia.

Rozwiązanie to jest bardzo szybkie, ale niebyvale obciąża pamięć. 16-bitowa teksturom o rozmiarach 128×128 zajmuje 32 kilobajty pamięci, więc 64 wersje tej teksturom będą zajmować aż 2 MB pamięci! Oczywiście w ten sposób znacznie ograniczymy liczbę różnych teksturom, które będzie można przechowywać w pamięci.

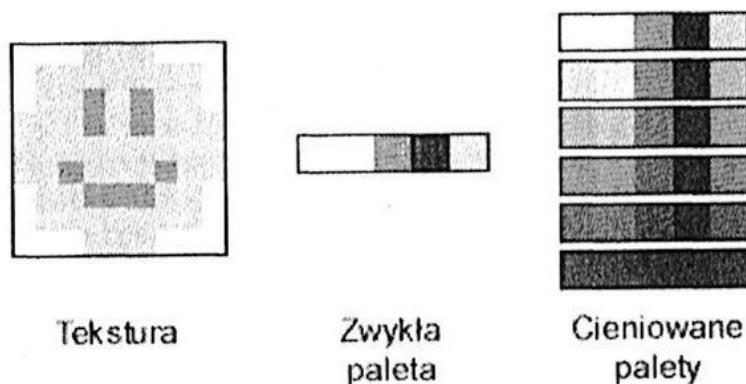
Kolejne rozwiązanie, pozwalające uniknąć tworzenia 64 wersji każdej teksturom, polega na przygotowaniu tylko jednej wersji każdej z teksturom, ale 64 wersji jej palety kolorów. Najpierw ustalamy, że wszystkie teksturom będą korzystać z ośmiobitowych kolorów. Dzięki temu każda teksturom będzie miała swoją własną małą, 256-kolorową paletę. Następnie — zamiast tworzenia 64 wersji teksturom — po prostu przygotujemy 64 wersje palety kolorów.

Jak to wygląda w uproszczeniu, można zobaczyć na rysunku 8.8.

Gdy palety wykorzystują kolory 16-bitowe, to 64 palety 256-kolorowe zajmują tylko 32 kilobajty pamięci. W przypadku teksturom 128×128 , opartych na 8-bitowych kolorach, oznaczać to będzie konieczność wykorzystania tylko 80 kilobajtów pamięci, co jest znacznie oszczędniejsze niż poprzednie rozwiązanie, które wymagało 2 MB pamięci dla każdej mapy bitowej.

Rysunek 8.8.

Aby podświetlić tekstury, przygotujemy normalną, w pełni podświetloną teksturę i kilka wersji jej palety kolorów, począwszy od palety dla tekstury w pełni podświetlanej po paletę dla tekstury zupełnie tonącej w mroku



Aby zaimplementować ten pomysł, przygotujemy klasę cieniowanej tekstury ShadedTexture, przedstawioną na listingu 8.7.

Listing 8.7. ShadedTexture.java

```
package com.brackeen.javagamebook.graphics3D.texture;

import java.awt.Color;
import java.awt.image.IndexColorModel;

/**
 * Klasa ShadedTexture jest podklassą klasy Texture z cieniowanymi
 * wariantami tekstury. Źródłowa tekstura przechowywana jest jako 8-bitowy
 * obraz, który dla każdego odcienia posiada osobną paletę.
 */
public final class ShadedTexture extends Texture {

    public static final int NUM_SHADE_LEVELS = 64;
    public static final int MAX_LEVEL = NUM_SHADE_LEVELS-1;

    private static final int PALETTE_SIZE_BITS = 8;
    private static final int PALETTE_SIZE = 1 << PALETTE_SIZE_BITS;

    private byte[] buffer;
    private IndexColorModel palette;
    private short[] shadeTable;
    private int defaultShadeLevel;
    private int widthBits;
    private int widthMask;
    private int heightBits;
    private int heightMask;

    /**
     * Tworzy nową teksturę ShadedTexture z podanej palety
     * i 8-bitowego bufora obrazu. Szerokość mapy bitowej to
     * 2 do potęgi widthBits, czyli (1 << widthBits). Podobnie,
     * wysokość mapy bitowej to 2 do potęgi heightBits, czyli
     * (1 << heightBits). Tekstura cieniowana jest od jej
     * oryginalnego koloru do czerni.
     */
    public ShadedTexture(byte[] buffer,
                         int widthBits, int heightBits,
                         IndexColorModel palette)
    {
        this(buffer, widthBits, heightBits, palette, Color.BLACK);
    }
}
```

```

    /**
     * Tworzy nową teksturę ShadedTexture z podanej palety oraz
     * 8-bitowego bufora obrazu i docelowego odcienia. Szerokość
     * mapy bitowej to 2 do potęgi widthBits, czyli  $(1 << \text{widthBits})$ .
     * Podobnie, wysokość mapy bitowej to 2 do potęgi heightBits,
     * czyli  $(1 << \text{heightBits})$ . Tekstura cieniowana jest od jej
     * oryginalnego koloru do docelowego odcienia.
    */
    public ShadedTexture(byte[] buffer,
        int widthBits, int heightBits,
        IndexColorModel palette, Color targetShade)
    {
        super(1 << widthBits, 1 << heightBits);
        this.buffer = buffer;
        this.widthBits = widthBits;
        this.heightBits = heightBits;
        this.widthMask = getWidth() - 1;
        this.heightMask = getHeight() - 1;
        this.buffer = buffer;
        this.palette = palette;
        defaultShadeLevel = MAX_LEVEL;

        makeShadeTable(targetShade);
    }

    /**
     * Tworzy tablicę odcieni dla tej tekstury ShadedTexture. Każda
     * pozycja w palecie będzie stopniowo cieniowana od oryginalnego
     * koloru do podanego odcienia.
    */
    public void makeShadeTable(Color targetShade) {
        shadeTable = new short[NUM_SHADE_LEVELS*PALETTE_SIZE];
        for (int level=0; level<NUM_SHADE_LEVELS; level++) {
            for (int i=0; i<palette.getMapSize(); i++) {
                int red = calcColor(palette.getRed(i),
                    targetShade.getRed(), level);
                int green = calcColor(palette.getGreen(i),
                    targetShade.getGreen(), level);
                int blue = calcColor(palette.getBlue(i),
                    targetShade.getBlue(), level);

                int index = level * PALETTE_SIZE + i;
                // RGB 5:6:5
                shadeTable[index] = (short)(
                    ((red >> 3) << 11) |
                    ((green >> 2) << 5) |
                    (blue >> 3));
            }
        }
    }

    private int calcColor(int palColor, int target, int level) {
        return (palColor - target) * (level+1) /
            NUM_SHADE_LEVELS + target;
    }
}

```

```

    /**
     * Ustawia domyślny poziom cieniowania wykorzystywany, gdy
     * wywoływana jest metoda getColor().
    */
    public void setDefaultShadeLevel(int level) {
        defaultShadeLevel = level;
    }

    /**
     * Pobiera domyślny poziom cieniowania wykorzystywany, gdy
     * wywoływana jest metoda getColor().
    */
    public int getDefaultShadeLevel() {
        return defaultShadeLevel;
    }

    /**
     * Pobiera 16-bitowy kolor tej tekstury Texture w podanej
     * lokacji (x,y), wykorzystując domyślny poziom cieniowania.
    */
    public short getColor(int x, int y) {
        return getColor(x, y, defaultShadeLevel);
    }

    /**
     * Pobiera 16-bitowy kolor tej tekstury Texture w podanej
     * lokacji (x,y), wykorzystując podany poziom cieniowania.
    */
    public short getColor(int x, int y, int shadeLevel) {
        return shadeTable[(shadeLevel << PALETTE_SIZE_BITS) |
            (0xff & buffer[
                (x & widthMask) |
                ((y & heightMask) << widthBits)])];
    }
}

```

W klasie ShadedTexture sama tekstura przechowywana jest w buforze tablicy byte[]. Tablica byte[] sama w sobie jest bezużyteczna, ponieważ nie zawiera żadnych danych na temat kolorów, lecz jedynie indeksy do 256-kolorowej palety. W tablicy odcienni shadeTable przechowywane są 64 wersje tej 256-kolorowej palety. Tablica odcienni tworzona jest w metodzie makeShadeTable(). Z kolei w metodzie getColor() tablica odcienni shadeTable wykorzystywana jest do konwertowania indeksu z 8-bitowego bufora na odpowiedni 16-bitowy kolor w palecie.

Warto również zwrócić uwagę, że tablica odcienni tworzona jest domyślnie w ten sposób, że krańcową wartością cieniowania jest kolor czarny, dzięki czemu wszystkie tekstury w normalnej palecie będą stopniowo cieniowane od współczynnika 1 do całkowitej czerni dla współczynnika 0. Korzystając z innych kolorów, można uzyskać odrobinę inne efekty. Kolejny pomysł polega na przygotowaniu efektu oświetlenia powierzchni snopem światła poprzez odwrócenie kierunku cieniowania palety od 0 dla ciemności, aż do pełnego oświetlenia dla wartości 1.

Teraz pora wypróbować nowe, cieniowane tekstury. Przygotujemy klasę renderera tekstur ShadedTexturedPolygonRenderer (patrz listing 8.8), która będzie rysować tekstury ShadedTexture, na bieżąco wyliczając intensywność oświetlenia dla każdego wielokąta.

Listing 8.8. *ShadedTexturedPolygonRenderer.java*

```
package com.brackeen.javagamebook.graphics3D;

import java.awt.*;
import java.awt.image.*;
import com.brackeen.javagamebook.math3D.*;
import com.brackeen.javagamebook.graphics3D.texture.*;

/**
 * Klasa ShadedTexturedPolygonRenderer jest podklassą PolygonRenderer,
 * renderującą dynamicznie teksturę ShadedTexture, oświetloną jednym
 * światłem. Domyślnie intensywność światła otoczenia wynosi 0.5 i nie
 * ma żadnego światła punktowego.
 */
public class ShadedTexturedPolygonRenderer
    extends FastTexturedPolygonRenderer
{

    private PointLight3D lightSource;
    private float ambientLightIntensity = 0.5f;
    private Vector3D directionToLight = new Vector3D();

    public ShadedTexturedPolygonRenderer(Transform3D camera,
                                         ViewWindow viewWindow)
    {
        this(camera, viewWindow, true);
    }

    public ShadedTexturedPolygonRenderer(Transform3D camera,
                                         ViewWindow viewWindow, boolean clearViewEveryFrame)
    {
        super(camera, viewWindow, clearViewEveryFrame);
    }

    /**
     * Pobiera punktowe źródło światła dla tego renderera.
     */
    public PointLight3D getLightSource()
    {
        return lightSource;
    }

    /**
     * Ustawia źródło światła dla tego renderera.
     */
    public void setLightSource(PointLight3D lightSource)
    {
        this.lightSource = lightSource;
    }
}
```

```
/***
     Pobiera intensywność światła otoczenia.
 */
public float getAmbientLightIntensity() {
    return ambientLightIntensity;
}

/***
     Ustawia intensywność światła otoczenia, zasadniczo między 0 a
     1.
 */
public void setAmbientLightIntensity(float i) {
    ambientLightIntensity = i;
}

protected void drawCurrentPolygon(Graphics2D g) {
    // Ustaw poziom cieniowania wielokąta, zanim zaczniesz rysować.
    if (sourcePolygon instanceof TexturedPolygon3D) {
        TexturedPolygon3D poly =
            ((TexturedPolygon3D)sourcePolygon);
        Texture texture = poly.getTexture();
        if (texture instanceof ShadedTexture) {
            calcShadeLevel();
        }
    }
    super.drawCurrentPolygon(g);
}

/***
     Wylicza poziom cieniowania dla bieżącego wielokąta.
 */
private void calcShadeLevel() {
    TexturedPolygon3D poly = (TexturedPolygon3D)sourcePolygon;
    float intensity = 0;
    if (lightSource != null) {

        // Punkt będący średnią z wierzchołków wielokąta.
        directionToLight.setTo(0.0.0);
        for (int i=0; i<poly.getNumVertices(); i++) {
            directionToLight.add(poly.getVertex(i));
        }
        directionToLight.divide(poly.getNumVertices());

        // Utwórz wektor wychodzący z tak uśrednionego wierzchołka
        // i skierowany w kierunku światła.
        directionToLight.subtract(lightSource);
        directionToLight.multiply(-1);

        // Pobierz odległość od źródła światła dla potrzeb wygaszania.
        float distance = directionToLight.length();
```

```

    // Wylicz odbicie dyfuzyjne.
    directionToLight.normalize();
    Vector3D normal = poly.getNormal();
    intensity = lightSource.getIntensity(distance)
        * directionToLight.getDotProduct(normal);
    intensity = Math.min(intensity, 1);
    intensity = Math.max(intensity, 0);
}

intensity+=ambientLightIntensity;
intensity = Math.min(intensity, 1);
intensity = Math.max(intensity, 0);
int level =
    Math.round(intensity*ShadedTexture.MAX_LEVEL);
((ShadedTexture)poly.getTexture()).
    setDefaultShadeLevel(level);
}
}

```

Klasa ShadedTexturedPolygonRenderer wylicza dla każdego wielokąta poziom cieniowania w metodzie calcShadeLevel(). Klasa ta służy wyłącznie celom demonstracyjnym. Tak naprawdę poziom cieniowania trzeba przeliczać na nowo tylko wtedy, gdy zmieni się w jakiś sposób źródło światła albo gdy wielokąt się poruszy. Ponadto, jak pamiętamy, renderer ShadedTexturedPolygonRenderer jest dość uproszczony — używa tylko jednego, punktowego źródła światła, a intensywność światła otoczenia jest taka sama dla każdego wielokąta.

Pomimo swej prostoty klasa ta pozwala nam jednak zmieniać na bieżąco intensywność źródła światła, co jest bardzo przydatną opcją. Aby skorzystać z nowego mechanizmu renderującego, przygotujemy przykład ShadingTest1. Efekt działania tego przykładu został przedstawiony na rysunku 8.9.

Rysunek 8.9.

Przykład ShadingTest1 przypisuje każdemu wielokątowi inne cieniowanie w zależności od położenia i intensywności źródła światła, które to parametry można zmieniać dynamicznie



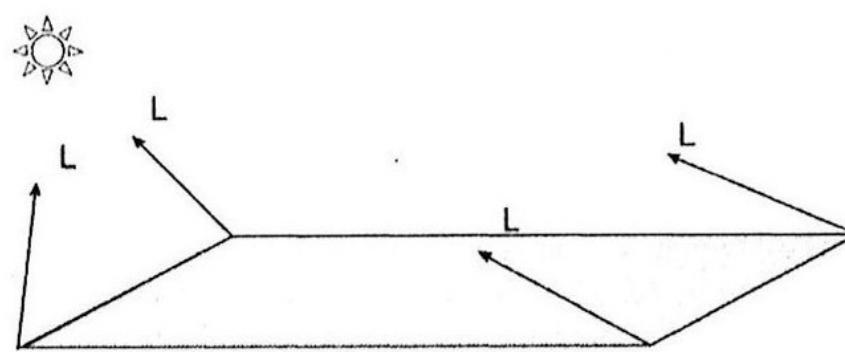
Przykład ShadingTest1 przypomina program TextureMapTest2, z tą tylko różnicą, że korzysta z klasy ShadedTexturedPolygonRenderer i że umożliwia zmianianie intensywności światła za pomocą klawiszy +/-.

Tworzenie zaawansowanych trików oświetleniowych za pomocą map cieniowania

Poprzedni przykład był całkiem niezły, ale cieniowanie całego wielokąta w ten sam sposób nie ma wiele wspólnego z rzeczywistością. Różne części tego samego wielokąta mogą być przecież ustawione pod różnym kątem oraz znajdująć się w różnej odległości od źródła światła, tak jak to zostało pokazane na rysunku 8.10 — szczególnie, jeśli wielokąt jest duży.

Rysunek 8.10.

Jednolite oświetlenie wielokąta nie jest zbyt zgodne z rzeczywistością, ponieważ różne części wielokąta mogą być ustawione pod różnym kątem oraz znajdująć się w różnej odległości do źródła światła



Aby przygotować bardziej realistyczne oświetlenie wielokątów, trzeba podzielić wielokąty na mniejsze wielokąty o rozmiarach 16×16 . Zamiast dzielić wielokąty na mniejsze, można też wyliczać osobno intensywność oświetlenia dla każdego tekscela wielokąta. Jak łatwo się zorientować, wyliczanie intensywności oświetlenia dla każdego tekscela będzie zbyt pracochłonne, aby wykonywać to na bieżąco, szczególnie jeśli w rysowanym świecie znajduje się kilka źródeł światła. Zamiast tego zazwyczaj wylicza się intensywności oświetlenia przed rozpoczęciem gry i przechowuje te wartości w mapach cieniowania przygotowanych dla każdego wielokąta.

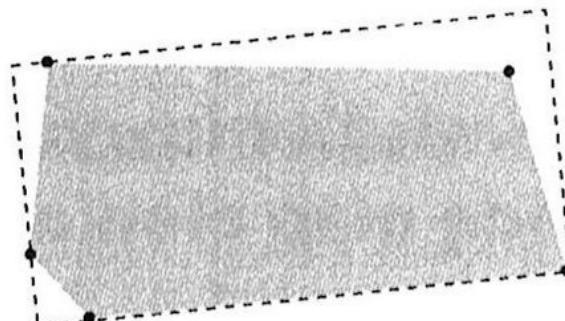
Aby ograniczyć pamięć zużywaną przez mapy cieniowania, można wyliczać dokładną intensywność oświetlenia tylko dla co któregoś piksela — na przykład dla co szesnastego piksela w pionie i w poziomie. W ten sposób mapa cieniowania będzie relatywnie mała w porównaniu z rozmiarami wielokąta.

Odnajdywanie prostokąta ograniczającego

Mapa cieniowania powinna być prostokątną mapą bitową, która pokrywać będzie całą powierzchnię wielokąta. Istnieje kilka sposobów znajdowania odpowiedniego prostokąta, a jednym z nich jest technika polegająca na odnajdywaniu najmniejszego prostokąta zawierającego wielokąt, przedstawiona na rysunku 8.11. Najmniejszy prostokąt zawierający wielokąt to taki prostokąt, który zupełnie pokrywa wielokąt, a jeden z jego boków pokrywa się częściowo z bokiem wielokąta.

Rysunek 8.11.

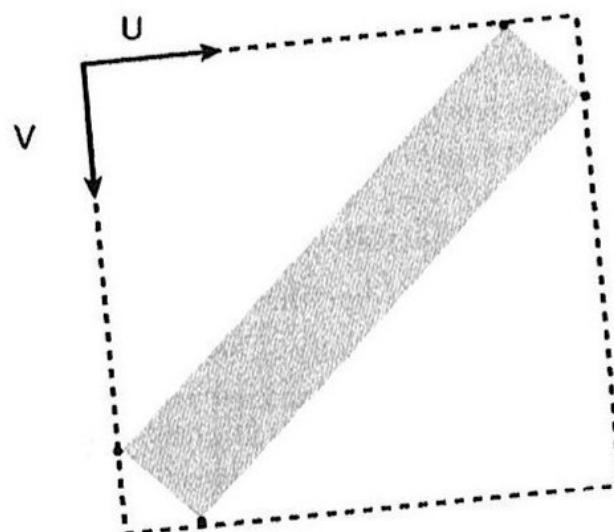
Najmniejszy prostokąt zawierający wielokąt zawsze będzie miał przynajmniej jeden bok pokrywający się częściowo z jednym z boków wielokąta



Niemniej tak naprawdę zależy nam na prostokącie, który będzie zorientowany w ten sam sposób co granice tekstuury, aby linie cieniowania pokrywały się z liniami tekstuury. Właśnie z takiego wariantu mapy cieniowania będziemy korzystać, należy jednak pamiętać, że rozwiązanie to ma jedną wadę: zużywa trochę więcej pamięci niż trzeba. Na przykład, jeśli zorientowalibyśmy wielokąt przedstawiony na rysunku 8.12 zgodnie z orientacją granic tekstuury, to jego powierzchnia byłaby znacznie większa niż powierzchnia wielokąta.

Rysunek 8.12.

Użycie prostokąta zawierającego wielokąt, jeśli będzie on zorientowany według orientacji tekstuury, może prowadzić do niepotrzebnego marnotrawienia pamięci



Do klasy TexturedPolygon3D dodamy metodę calcBoundingBox(), która wyliczać będzie granice prostokąta zawierającego wielokąt, zorientowanego tak, jak tekstura. Metoda ta odnajduje długości boków prostokąta, korzystając z iloczynu skalarnego, dzięki czemu każdy z wierzchołków wielokąta znajdzie się wewnątrz prostokąta.

```
/**
 * Wylicza prostokąt zawierający ten wielokąt i zorientowany
 * tak, jak tekstura.
 */
public Rectangle3D calcBoundingBox() {

    Vector3D u = new Vector3D(textureBounds.getDirectionU());
    Vector3D v = new Vector3D(textureBounds.getDirectionV());
    Vector3D d = new Vector3D();
    u.normalize();
    v.normalize();

    float uMin = 0;
    float uMax = 0;
    float vMin = 0;
    float vMax = 0;
```

```
for (int i=0; i<getNumVertices(); i++) {
    d.setTo(getVertex(i));
    d.subtract(getVertex(0));
    float uLength = d.getDotProduct(u);
    float vLength = d.getDotProduct(v);
    uMin = Math.min(uLength, uMax);
    uMax = Math.max(uLength, uMax);
    vMin = Math.min(vLength, vMax);
    vMax = Math.max(vLength, vMax);
}

Rectangle3D boundingRect = new Rectangle3D();
Vector3D origin = boundingRect.getOrigin();
origin.setTo(getVertex(0));
d.setTo(u);
d.multiply(uMin);
origin.add(d);
d.setTo(v);
d.multiply(vMin);
origin.add(d);
boundingRect.getDirectionU().setTo(u);
boundingRect.getDirectionV().setTo(v);
boundingRect.setWidth(uMax - uMin);
boundingRect.setHeight(vMax - vMin);

// Ręcznie ustaw wektor normalny, ponieważ kierunki tekstury
// mogłyby dać wektor normalny skierowany odwrotnie niż
// wektor normalny wielokąta.
boundingRect.setNormal(getNormal());

return boundingRect;
}
```

Stosowanie mapy cieniowania

Kolejnym krokiem jest ustalenie, kiedy zastosować mapę na wielokącie. Jeden z pomysłów polega na połączeniu mapy cieniowania z tekstrurą wielokąta, by zawsze przygotować cieniowaną powierzchnię, tak jak to zostało przedstawione na rysunku 8.13.

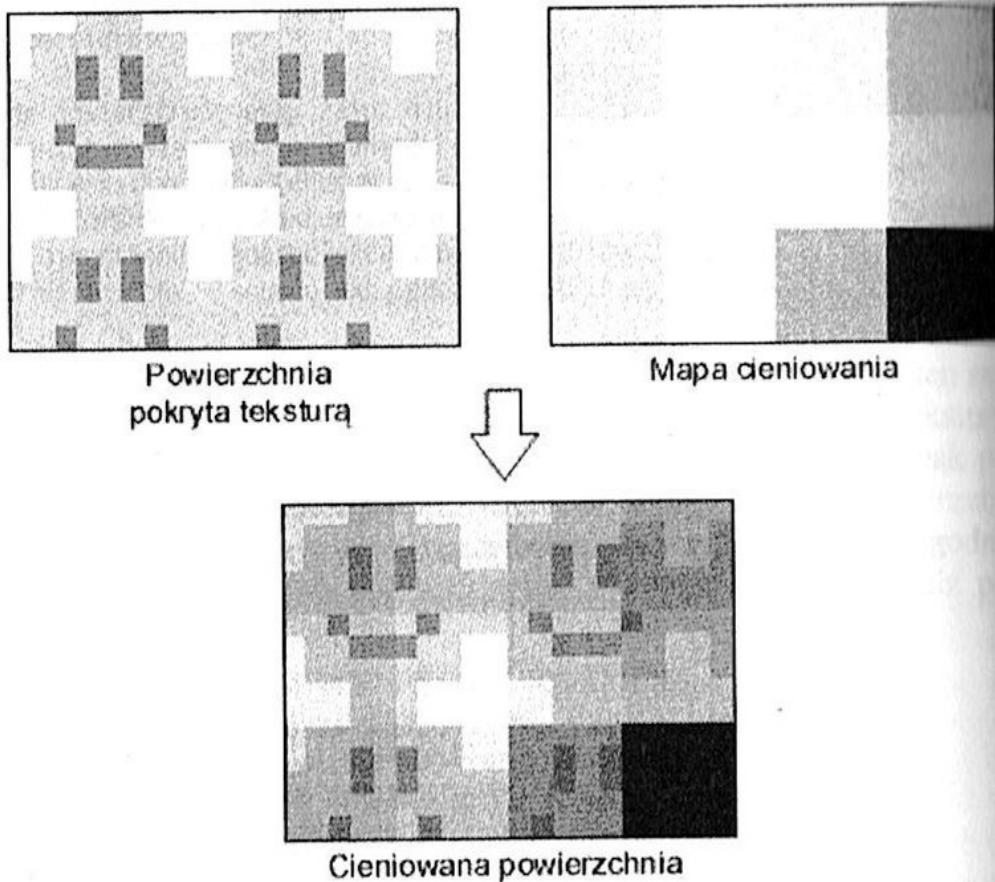
Ta prosta technika tworzy cieniowaną powierzchnię, której cieniowanie jest jednak odrobinę kanciaste z racji małej dokładności mapy cieniowania. Aby to naprawić, będziemy interpolować wartości między wybranymi punktami mapy cieniowania, by uzyskać obraz taki, jak na rysunku 8.14.

Zamiast zaczynać od tworzenia powierzchni, można też spróbować wykonywać cieniowanie w tym samym momencie co mapowanie tekstuur. Technika ta wydaje się rozsądna, jednak ma tę wadę, że nie da się jej zbytnio zoptymalizować za pomocą interpolacji mapy cieniowania, ponieważ wyliczanie interpolowanej wartości cieniowania dla każdego piksela ekranu będzie trwało dość długo.

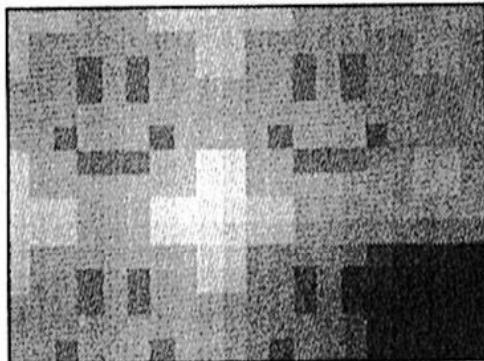
Z kolei zastosowanie wcześniejszej przygotowanych, cieniowanych powierzchni będzie działać tak szybko, jak mapowanie tekstuur już po zbudowaniu powierzchni. Technika korzystająca z gotowych powierzchni wymaga jednak wykorzystania dużych zasobów

Rysunek 8.13.

Mapa cieniowania pokrywa całą powierzchnię wielokąta i jest łączona z teksturą, co pozwala otrzymać cieniowaną powierzchnię

**Rysunek 8.14.**

Zastosowanie na mapie cieniowania techniki interpolacji sprawia, że ostatecznie otrzymana powierzchnia wygląda znacznie lepiej

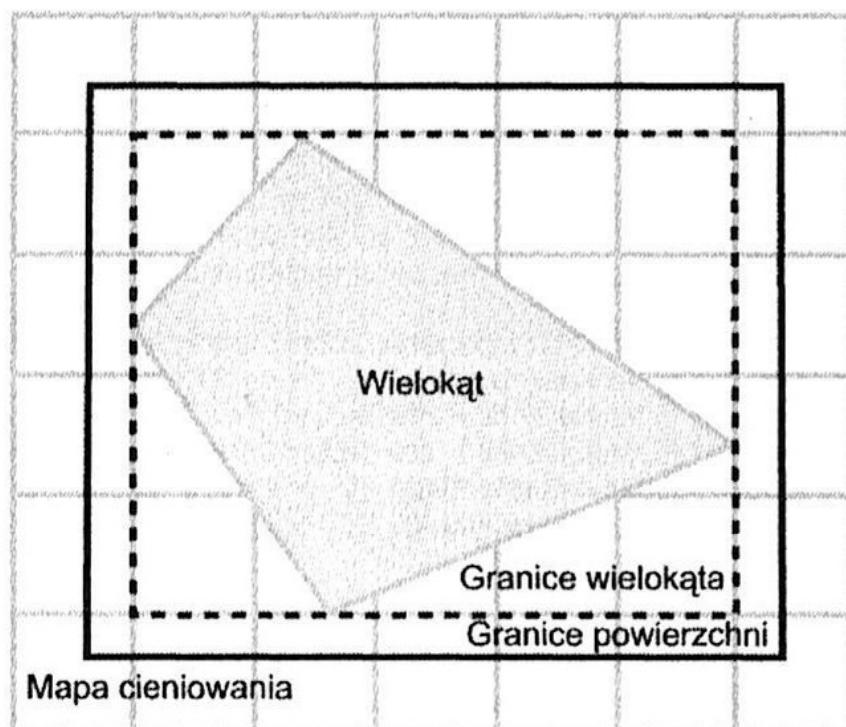


pamięci (o sprawach związanych z pamięcią opowiem w dalszej części książki). Tymczasem po prostu spróbujmy napisać prosty kod, który będzie tworzył mapy cieniowania i cieniowane powierzchnie.

Po pierwsze, aby doprecyzować nasze definicje granic wielokąta, mapy cieniowania i powierzchni, przyjrzyjmy się rysunkowi 8.15. Na tym rysunku granice powierzchni są trochę na zewnątrz względem granic wielokąta, aby skorygować ewentualne błędy zaokrąglenia powstające podczas obliczeń na liczbach zmiennoprzecinkowych, wykonywanych przy mapowaniu tekstur. Błąd zaokrąglenia może przesunąć ostateczne współrzędne (x, y) powierzchni o teksel lub dwa. Nie zależy nam przecież na wyliczaniu podczas mapowania tekstur wartości znajdujących się poza granicami powierzchni. Powiększenie powierzchni względem granic wielokąta pozwala nam poradzić sobie z tym problemem. Ponadto warto zwrócić uwagę, że mapa cieniowania jest orientowana według granic wielokąta, a nie według granic powierzchni.

Rysunek 8.15.

Mapa cieniowania zostanie odrobine poszerzona
na brzegach, aby zabezpieczyć
je przed efektami błędów
rozkąglenia podczas
obliczeń na liczbach
jemnoprzecinkowych.
Mapę cieniowania orientujemy
według granic wielokąta,
a nie granic powierzchni



Budowanie mapy cieniowania

Teraz możemy przystąpić do budowania mapy cieniowania. Należy pamiętać, że jeśli wielokąt się nie porusza względem źródła światła, a źródło światła się nie zmienia, to dla każdego wielokąta wystarczy zbudować mapę cieniowania tylko raz. W przedstawionym tutaj mechanizmie renderującym mapą cieniowania będzie statyczna, więc nie będziemy jej musieli już więcej przeliczać na nowo. Kod budujący mapę cieniowania przedstawiony został na listingu 8.9, prezentującym klasę ShadedSurface. Klasa ShadedSurface jest podklassą klasy Texture i zawiera źródłową teksturę, dane na temat powierzchni oraz mapę cieniowania.

Listing 8.9. Budowanie mapy cieniowania w ShadedSurface.java

```

public static final int SURFACE_BORDER_SIZE = 1;

public static final int SHADE_RES_BITS = 4;
public static final int SHADE_RES = 1 << SHADE_RES_BITS;
public static final int SHADE_RES_MASK = SHADE_RES - 1;
public static final int SHADE_RES_SQ = SHADE_RES * SHADE_RES;
public static final int SHADE_RES_SQ_BITS = SHADE_RES_BITS * 2;

...
/** Tworzy mapę cieniowania dla tej powierzchni, uwzględniając
 * listę punktowych źródeł światła oraz intensywność światła otoczenia.
 */
public void buildShadeMap(List pointLights,
                           float ambientLightIntensity)
{
    Vector3D surfaceNormal = surfaceBounds.getNormal();
}

```

```

int polyWidth = (int)surfaceBounds.getWidth() -
    SURFACE_BORDER_SIZE*2;
int polyHeight = (int)surfaceBounds.getHeight() -
    SURFACE_BORDER_SIZE*2;
// Zakładamy, że SURFACE_BORDER_SIZE <= SHADE_RES
shadeMapWidth = polyWidth / SHADE_RES + 4;
shadeMapHeight = polyHeight / SHADE_RES + 4;
shadeMap = new byte[shadeMapWidth * shadeMapHeight];

// Wylicz początek układu współrzędnych dla mapy cieniowania
Vector3D origin = new Vector3D(surfaceBounds.getOrigin());
Vector3D du = new Vector3D(surfaceBounds.getDirectionU());
Vector3D dv = new Vector3D(surfaceBounds.getDirectionV());
du.multiply(SHADE_RES - SURFACE_BORDER_SIZE);
dv.multiply(SHADE_RES - SURFACE_BORDER_SIZE);
origin.subtract(du);
origin.subtract(dv);

// Wylicz cieniowanie dla każdego przykładowego punktu.
Vector3D point = new Vector3D();
du.setTo(surfaceBounds.getDirectionU());
dv.setTo(surfaceBounds.getDirectionV());
du.multiply(SHADE_RES);
dv.multiply(SHADE_RES);
for (int v=0; v<shadeMapHeight; v++) {
    point.setTo(origin);
    for (int u=0; u<shadeMapWidth; u++) {
        shadeMap[u + v * shadeMapWidth] =
            calcShade(surfaceNormal, point,
                      pointLights, ambientLightIntensity);
        point.add(du);
    }
    origin.add(dv);
}
}

/**
Ustala cieniowanie wybranego punktu na powierzchni wielokąta.
Wylicza oświetlenie według wzoru Lambert'a dla każdego punktu
na płaszczyźnie. Każdy punktowe źródło światła ma określoną
intensywność i odległość całkowitego wygaszenia, niemniej nie są
tutaj wyliczane cienie lub odbicia innych wielokątów. Zwraca wartość
z przedziału od 0 do ShadedTexture.MAX_LEVEL.
*/
protected byte calcShade(Vector3D normal, Vector3D point,
                          List pointLights, float ambientLightIntensity)
{
    float intensity = 0;
    Vector3D directionToLight = new Vector3D();

    for (int i=0; i<pointLights.size(); i++) {
        PointLight3D light = (PointLight3D)pointLights.get(i);
        directionToLight.setTo(light);
        directionToLight.subtract(point);
    }
}

```

```
        float distance = directionToLight.length();
        directionToLight.normalize();
        float lightIntensity = light.getIntensity(distance)
            * directionToLight.getDotProduct(normal);
        lightIntensity = Math.min(lightIntensity, 1);
        lightIntensity = Math.max(lightIntensity, 0);
        intensity += lightIntensity;
    }

    intensity = Math.min(intensity, 1);
    intensity = Math.max(intensity, 0);

    intensity+=ambientLightIntensity;

    intensity = Math.min(intensity, 1);
    intensity = Math.max(intensity, 0);
    int level = Math.round(intensity*ShadedTexture.MAX_LEVEL);
    return (byte)level;
}
}
```

Kod ten jest dość prosty i oczywisty. Metoda `buildShadeMap()` tworzy mapę cienowania dla określonej powierzchni, natomiast metoda `calcShade()` wylicza intensywność oświetlenia dla każdego punktu na mapie cienowania.

W tym mechanizmie renderującym tworzymy mapy cienowania tuż przed rozpoczęciem gry, niemniej można również przechowywać wszystkie mapy cienowania w pliku i ładować je z tego pliku przed początkiem gry. W ten sposób nie będzie konieczne ponowne przeliczanie map w momencie uruchamiania gry.

Budowanie powierzchni

W kolejnym etapie zajmiemy się przygotowywaniem powierzchni, co zostało pokazane na listingu 8.10. Kod ten wymaga zastosowania dwóch osobnych metod: jednej, która umożliwia wyliczanie intensywności oświetlenia na bazie interpolacji między dokładnie wyliczonymi wartościami na mapie cienowania oraz drugiej, która narysuje na powierzchni teksturę już z właściwym cienowaniem.

Listing 8.10. Budowanie powierzchni w `ShadedSurface.java`

```
/**
 * Buduje powierzchnię. Najpierw metoda ta wywołuje
 * retrieveSurface(), by sprawdzić, czy powierzchnia powinna
 * być przebudowywana. Jeśli nie, powierzchnia jest budowana
 * poprzez ułożenie tekstury i nałożenie już gotowej mapy cienowania.
 */
public void buildSurface() {

    if (retrieveSurface()) {
        return;
    }
```

```

int width = (int)surfaceBounds.getWidth();
int height = (int)surfaceBounds.getHeight();

// Utwórz nową powierzchnię (bufor).
newSurface(width, height);

// Buduje powierzchnię przy założeniu,
// że granice powierzchni i granice tekstury są
// zorientowane tak samo (choć mogą się zaczynać w różnych punktach).
Vector3D origin = sourceTextureBounds.getOrigin();
Vector3D directionU = sourceTextureBounds.getDirectionU();
Vector3D directionV = sourceTextureBounds.getDirectionV();

Vector3D d = new Vector3D(surfaceBounds.getOrigin());
d.subtract(origin);
int startU = (int)((d.getDotProduct(directionU) -
    SURFACE_BORDER_SIZE));
int startV = (int)((d.getDotProduct(directionV) -
    SURFACE_BORDER_SIZE));
int offset = 0;
int shadeMapOffsetU = SHADE_RES - SURFACE_BORDER_SIZE -
    startU;
int shadeMapOffsetV = SHADE_RES - SURFACE_BORDER_SIZE -
    startV;

for (int v=startV; v<startV + height; v++) {
    sourceTexture.setCurrRow(v);
    int u = startU;
    int amount = SURFACE_BORDER_SIZE;
    while (u < startU + width) {
        getInterpolatedShade(u + shadeMapOffsetU,
            v + shadeMapOffsetV);

        // Rysuj, dopóki nie zajdzie konieczność ponownego wyliczenia
        // interpolowanego cienia (każdego piksela SHADE_RES).
        int endU = Math.min(startU + width, u + amount);
        while (u < endU) {
            buffer[offset++] =
                sourceTexture.getColorCurrRow(u,
                    shadeValue >> SHADE_RES_SQ_BITS);
            shadeValue+=shadeValueInc;
            u++;
        }
        amount = SHADE_RES;
    }
}
/** Pobiera cieniowanie (z mapy cieniowania) dla wybranej
 *  lokacji (u,v). Bity wartości u i v powinny zostać przesunięte
 *  w lewo SHADE_RES_BITS, a dodatkowe bity są wykorzystywane
 *  do wyliczania interpolowanych wartości. Przykład interpolacji:
 *  lokacja w połowie drogi między wartościami 1 i 3 cieniowania
 *  będzie miała wartość 2.
 */
public int getInterpolatedShade(int u, int v) {

```

```
int fracU = u & SHADE_RES_MASK;
int fracV = v & SHADE_RES_MASK;

int offset = (u >> SHADE_RES_BITS) +
    ((v >> SHADE_RES_BITS) * shadeMapWidth);

int shade00 = (SHADE_RES - fracV) * shadeMap[offset];
int shade01 = fracV * shadeMap[offset + shadeMapWidth];
int shade10 = (SHADE_RES - fracV) * shadeMap[offset + 1];
int shade11 = fracV * shadeMap[offset + shadeMapWidth + 1];

shadeValue = SHADE_RES_SQ/2 +
    (SHADE_RES - fracU) * shade00 +
    (SHADE_RES - fracU) * shade01 +
    fracU * shade10 +
    fracU * shade11;

// Wartość zwiększana wraz ze zwiększaniem u.
shadeValueInc = -shade00 - shade01 + shade10 + shade11;

return shadeValue >> SHADE_RES_SQ_BITS;
}

/**
 * Pobiera cieniowanie (ze zbudowanej mapy cieniowania) dla
 * podanej lokacji (u,v).
 */
public int getShade(int u, int v) {
    return shadeMap[u + v * shadeMapWidth];
}
```

Oczywiście dobrze byłoby, aby tworzenie powierzchni na bieżąco przebiegało w trakcie gry najszybciej, jak to tylko możliwe. Dlatego brak optymalizacji w metodzie buildSurface() jest jej słabością.

Dla przykładu dodaliśmy do klasy ShadedTexture dwie metody setCurrRow() i getColorCurrRow(). Ponieważ budujemy powierzchnię wiersz po wierszu, możemy użyć tych metod zamiast metody getColor(), dzięki czemu nie trzeba będzie przeliczać na nowo przesunięcia wiersza dla każdego tekscela.

Metoda getInterpolatedShade() pobiera wartość cieniowania dla określonej lokacji. Nie ma potrzeby wywoływania tej metody dla każdego tekscela, ponieważ wszystkie wiersze powierzchni budujemy w poziomie. Zamiast tego można użyć wartości shadeValueInc, aby powiększać po prostu odpowiednio wartość shadeValue dla każdego tekscela. W ten sposób będziemy musieli pobierać właśnie wyliczoną wartość, stanowiącą podstawę interpolacji tylko dla co szesnastego tekscela — w podobny sposób, w jaki optymalizowaliśmy mapowanie tekstur.

Na koniec listing 8.11 przedstawia kod, który zajmuje się tworzeniem cieniowanej powierzchni ShadedSurface.

Listing 8.11. Tworzenie instancji klasy ShadedSurface w ShadedSurface.java

```


    /**
     * Tworzy powierzchnię ShadedSurface dla danego wielokąta.
     * Mapa cieniowana tworzona jest dla określonej listy punktowych
     * źródeł światła o ustalonej intensywności.
    */
    public static void createShadedSurface(
        TexturedPolygon3D poly, ShadedTexture texture,
        Rectangle3D textureBounds,
        List lights, float ambientLightIntensity)
    {

        // Tworzy granice powierzchni.
        poly.setTexture(texture, textureBounds);
        Rectangle3D surfaceBounds = poly.calcBoundingRectangle();

        // Dodaje granicom surfaceBounds dodatkową zewnętrzną granicę,
        // by skorygować drobne błędy powstałe w trakcie mapowania tekstury.
        Vector3D du = new Vector3D(surfaceBounds.getDirectionU());
        Vector3D dv = new Vector3D(surfaceBounds.getDirectionV());
        du.multiply(SURFACE_BORDER_SIZE);
        dv.multiply(SURFACE_BORDER_SIZE);
        surfaceBounds.getOrigin().subtract(du);
        surfaceBounds.getOrigin().subtract(dv);
        int width = (int) Math.ceil(surfaceBounds.getWidth() +
            SURFACE_BORDER_SIZE*2);
        int height = (int) Math.ceil(surfaceBounds.getHeight() +
            SURFACE_BORDER_SIZE*2);
        surfaceBounds.setWidth(width);
        surfaceBounds.setHeight(height);

        // Tworzy teksturę cieniowanej powierzchni.
        ShadedSurface surface = new ShadedSurface(width, height);
        surface.setTexture(texture, textureBounds);
        surface.setSurfaceBounds(surfaceBounds);

        // Tworzy mapę cieniowania dla powierzchni.
        surface.buildShadeMap(lights, ambientLightIntensity);

        // Definiuje powierzchnię wielokąta.
        poly.setTexture(surface, surfaceBounds);
    }


```

Przechowywanie powierzchni w pamięci podręcznej

Należy pamiętać, że powierzchnie zajmują bardzo wiele miejsca w pamięci, więc przygotowywanie zawsze powierzchni dla każdego wielokąta w grze może nie być możliwe. Zamiast tego może być konieczne tworzenie powierzchni na bieżąco i tylko dla tych wielokątów, które są aktualnie widoczne.

Ponieważ budowanie powierzchni jest bardzo kosztowne, nie jest zbyt dobrym pomysłem tworzenie każdej powierzchni dla każdej klatki. Zamiast tego mechanizm renderujący przechowuje w pamięci listę „widocznych” powierzchni. Tylko te powierzchnie,

które występują w bieżącej klatce, są przechowywane na liście, natomiast pozostałe są usuwane z pamięci, aby zrobić miejsce dla kolejnych przestrzeni.

Zawartość listy zależy będzie od wielu warunków, nam jednak zależy będzie przede wszystkim na przechowywaniu tych wielokątów, które pojawiać się będą dość często, choć niekoniecznie w każdej klatce. Na przykład jeśli gracz skręca szybko to w prawo, to w lewo, rozglądając się po scenerii, dana powierzchnia może się co chwilę pojawiać i znikać, co będzie powodować konieczność odtwarzania jej za każdym razem, gdy się znowu pojawi.

W takich sytuacjach mechanizm sprzątania pamięci nie zdąży nawet usunąć powierzchni z pamięci, zanim ta nie pojawi się ponownie. Czy nie byłoby wspaniale, by — zamiast odbudowywać tę powierzchnię od zera — „poprosić” po prostu mechanizm oczyszczania pamięci Javy, by zwrócił ją nam, o ile nie została jeszcze usunięta? Okazuje się, że można to bez trudu zrobić. Służą do tego tzw. *miękkie odwołania* (ang. *soft references*).

Obiekt klasy *SoftReference* umożliwia nam tworzenie miękkich odwołań w odróżnieniu od zwykłych, twardych *odwołań* (ang. *hard references*). Jeśli istnieje tylko miękkie odwołanie do obiektu (tzn. obiekt jest *osiągalny* za pomocą miękkich odwołań, ang. *softly reachable*), to mechanizm oczyszczania pamięci nie musi go od razu usuwać. Zazwyczaj więc, zanim przystąpi do usuwania takich obiektów, najpierw usunie te, do których nie istnieją żadne odwołania. Ponadto obiekty, do których odwołują się tylko miękkie odwołania, są ostatecznie usuwane zanim maszyna wirtualna zgłosi wyjątek *OutOfMemoryException*.

W klasie *ShadedSurface* nowe miękkie odwołanie *SoftReference* do bufora powierzchni tworzy się w następujący sposób:

```
/**
 * Tworzy nową powierzchnię i odnoszącą się
 * do niej miękkie odwołanie SoftReference.
 */
protected void newSurface(int width, int height) {
    buffer = new short[width*height];
    bufferReference = new SoftReference(buffer);
}
```

Później, jeśli mechanizm renderowania wielokąta ustali, że powierzchnia nie powinna być już dłużej widoczna, usuwane jest twarde odwołanie:

```
/**
 * Oczyszcza tę powierzchnię, pozwalając mechanizmowi usuwania
 * śmieci usunąć ją z pamięci w razie konieczności.
 */
public void clearSurface() {
    buffer = null;
}
```

W tym momencie tablica bufora jest dostępna tylko za pośrednictwem miękkiego odwołania, więc mechanizm oczyszczania pamięci nie musi się spieszyć z usuwaniem jej z pamięci.

Jeśli później będziemy potrzebować powierzchni z powrotem, będziemy mogli sprawdzić, czy można ją odzyskać z mechanizmu oczyszczania pamięci. Jeśli tak jest, metoda `get()` klasy `SoftReference` zwróci obiekt, który będzie można z powrotem przypisać do trwałego odwołania.

```
/*
 * Jeśli wcześniej zbudowany bufor został oczyszczony, ale
 * został jeszcze usunięty z pamięci przez mechanizm oczyszczania
 * pamięci, to metoda ta spróbuje go odzyskać. Jeśli się to uda,
 * metoda zwraca wartość true.
 */
public boolean retrieveSurface() {
    if (buffer == null) {
        buffer = (short[])bufferReference.get();
    }
    return !(buffer == null);
}
```

Jeśli obiekt został już uprzątnięty przez mechanizm oczyszczania pamięci, to metoda `get()` zwróci wartość `null` i konieczne będzie ponowne zbudowanie powierzchni.

Czyli, mówiąc w uproszczeniu, zaimplementowaliśmy mechanizm, który przypomina wygrzebywanie starych powierzchni z kosza na śmieci.

Sun zaleca, aby mechanizmy oczyszczania pamięci nie wyrzucały ostatnio używanych lub niedawno utworzonych obiektów, dostępnych za pomocą miękkich odwołań, niemniej należy pamiętać, że nie każdy z nich postępuje w ten sposób. Można też spróbować samodzielnie zaimplementować kolejny poziom przechowywania powierzchni w pamięci, choć skorzystanie z miękkich odwołań jest najprostszym rozwiązaniem.

To w zasadzie wszystko, co jest potrzebne do rysowania zacienionych powierzchni. Dla porządku prezentuję jeszcze resztę klasy `ShadedSurface` na listingu 8.12.

Listing 8.12. Pozostałe metody z `ShadedSurface.java`

```
package com.brackeen.javagamebook.graphics3D.texture;

import java.lang.ref.SoftReference;
import java.util.List;
import com.brackeen.javagamebook.math3D.*;

/*
 * Klasa ShadedSurface jest zawczasu cieniowaną teksturą Texture, która
 * jest następnie mapowana na wielokąt.
 */
public final class ShadedSurface extends Texture {

    public static final int SURFACE_BORDER_SIZE = 1;

    public static final int SHADE_RES_BITS = 4;
    public static final int SHADE_RES = 1 << SHADE_RES_BITS;
    public static final int SHADE_RES_MASK = SHADE_RES - 1;
    public static final int SHADE_RES_SQ = SHADE_RES*SHADE_RES;
    public static final int SHADE_RES_SQ_BITS = SHADE_RES_BITS*2;
```

```
private short[] buffer;
private SoftReference bufferReference;
private boolean dirty;
private ShadedTexture sourceTexture;
private Rectangle3D sourceTextureBounds;
private Rectangle3D surfaceBounds;
private byte[] shadeMap;
private int shadeMapWidth;
private int shadeMapHeight;

// Dla przyrostowego wyliczania wartości cieniowania.
private int shadeValue;
private int shadeValueInc;

/** Tworzy powierzchnię ShadedSurface o określonej szerokości i wysokości.
 */
public ShadedSurface(int width, int height) {
    this(null, width, height);
}

/** Tworzy powierzchnię ShadedSurface z podanym buforem oraz o określonej szerokości i wysokości.
 */
public ShadedSurface(short[] buffer, int width, int height) {
    super(width, height);
    this.buffer = buffer;
    bufferReference = new SoftReference(buffer);
    sourceTextureBounds = new Rectangle3D();
    dirty = true;
}

/** Pobiera 16-bitowy kolor piksela w lokacji (x,y) na mapie bitowej. Przyjmuje się, że wartości x i y znajdują się w obrębie granic przestrzeni; w przeciwnym razie pojawi się wyjątek ArrayIndexOutOfBoundsException.
 */
public short getColor(int x, int y) {
    return buffer[x + y * width];
}

/** Pobiera 16-bitowy kolor piksela w lokacji (x,y) na mapie bitowej. Sprawdzamy, czy wartości x i y są w obrębie granic powierzchni; jeśli nie ma ich tam, zwracany jest piksel krawędzi tekstury.
 */
public short getColorChecked(int x, int y) {
    if (x < 0) {
        x = 0;
    }
}
```

```
        else if (x >= width) {
            x = width-1;
        }
        if (y < 0) {
            y = 0;
        }
        else if (y >= height) {
            y = height-1;
        }
        return getColor(x,y);
    }

    /**
     * Zaznacza, czy ta powierzchnia jest „brudna” (dirty). Może być
     * konieczne oczyszczanie takich powierzchni poza naszym mechanizmem.
     */
    public void setDirty(boolean dirty) {
        this.dirty = dirty;
    }

    /**
     * Sprawdza, czy powierzchnia jest „brudna” (dirty). Może być
     * konieczne oczyszczanie takich powierzchni poza naszym mechanizmem.
     */
    public boolean isDirty() {
        return dirty;
    }

    /**
     * Ustawia źródłową teksturę i jej granice dla tej
     * powierzchni ShadedSurface.
     */
    public void setTexture(ShadedTexture texture,
                          Rectangle3D bounds)
    {
        setTexture(texture);
        sourceTextureBounds.setTo(bounds);
    }

    /**
     * Ustawia granice powierzchni ShadedSurface.
     */
    public void setSurfaceBounds(Rectangle3D surfaceBounds) {
        this.surfaceBounds = surfaceBounds;
    }

    /**
     * Pobiera granice dla tej powierzchni ShadedSurface.
     */
    public Rectangle3D getSurfaceBounds() {
        return surfaceBounds;
    }
}
```

Omówilem już prawie wszystko w kodzie klasy ShadedSurface poza znacznikiem „brudnej” powierzchni (dirty). Znacznik ten nie jest istotny w odniesieniu do samej powierzchni ShadedSurface — klasa ShadedSurface nigdzie nie sprawdza, czy znacznik dirty jest ustawiony czy nie. Znacznik można wykorzystywać w zewnętrznym kodzie do sprawdzania, czy powierzchnia powinna zostać oczyszczona. Przykład takiego sprawdzania zaprezentowałem w klasie renderera ShadedSurfacePolygonRenderer na listingu 8.13. Nie różni się ona w zasadzie niczym od pozostałych mechanizmów renderujących, z tą tylko różnicą, że przechowuje listę wszystkich powierzchni, które zostały wcześniej zbudowane. Jak wspominałem wcześniej, budujemy tylko te powierzchnie, które są widoczne. Jeśli dana powierzchnia nie jest widoczna (czyli została oznaczona jako dirty), to wszelkie dane powierzchni są czyszczone, a sama „brudna” powierzchnia jest usuwana z listy.

Listing 8.13. *ShadedSurfacePolygonRenderer.java*

```
package com.brackeen.javagamebook.graphics3D;

import java.awt.*;
import java.awt.image.*;
import java.util.List;
import java.util.LinkedList;
import java.util.Iterator;

import com.brackeen.javagamebook.math3D.*;
import com.brackeen.javagamebook.graphics3D.texture.*;

/**
 * ShadedSurfacePolygonRenderer jest podklassą klasy PolygonRenderer,
 * która renderuje wielokąt z powierzchniami ShadedSurface. Śledzi ona
 * zbudowane powierzchnie i oczyszcza wszystkie powierzchnie, które nie były
 * używane w ostatniej renderowanej klatce, by oszczędzać pamięć.
 */
public class ShadedSurfacePolygonRenderer
    extends FastTexturedPolygonRenderer
{

    private List builtSurfaces = new LinkedList();

    public ShadedSurfacePolygonRenderer(Transform3D camera,
                                         ViewWindow viewWindow)
    {
        this(camera, viewWindow, true);
    }

    public ShadedSurfacePolygonRenderer(Transform3D camera,
                                         ViewWindow viewWindow, boolean eraseView)
    {
        super(camera, viewWindow, eraseView);
    }

    public void endFrame(Graphics2D g) {
        super.endFrame(g);
    }
}
```

```

// Oczyszczenie wszystkich powierzchni, które nie były używane w tej klatce.
Iterator i = builtSurfaces.iterator();
while (i.hasNext()) {
    ShadedSurface surface = (ShadedSurface)i.next();
    if (surface.isDirty()) {
        surface.clearSurface();
        i.remove();
    }
    else {
        surface.setDirty(true);
    }
}
}

protected void drawCurrentPolygon(Graphics2D g) {
    buildSurface();
    super.drawCurrentPolygon(g);
}

/**
 * Buduje wszystkie powierzchnie wielokąta, jeśli jego
 * powierzchnia ShadedSurface została oczyszczona.
 */
protected void buildSurface() {
    // Buduje powierzchnię, w razie potrzeby.
    if (sourcePolygon instanceof TexturedPolygon3D) {
        Texture texture =
            ((TexturedPolygon3D)sourcePolygon).getTexture();
        if (texture instanceof ShadedSurface) {
            ShadedSurface surface =
                (ShadedSurface)texture;
            if (surface.isCleared()) {
                surface.buildSurface();
                builtSurfaces.add(surface);
            }
            surface.setDirty(false);
        }
    }
}
}

```

Klasa ShadedSurfacePolygonRenderer wywołuje metodę buildSurface() dla każdego wielokąta. Jeśli powierzchnia jest czysta, to jest budowana na nowo (lub odzyskiwana z mechanizmu oczyszczania pamięci), a następnie dodawana do listy budowanych powierzchni.

Przykład z cieniowaniem powierzchni

Teraz możemy już przygotować przykładowy program korzystający z cieniowanych powierzchni. Klasa ShadingTest2 nie różni się w zasadzie niczym od klasy poprzedniego przykładu TextureMapTest2, tyle tylko, że tworzy punktowe źródła światła PointLight3D i korzysta z renderera ShadedSurfaceRenderer. Ekran działającego programu ShadingTest2 został przedstawiony na rysunku 8.16.

Rysunek 8.16.
Program *ShadingTest2*
demonstruje rysowanie
cieniowanych powierzchni

Ruch: mysz/klawisze kurSORA. Esc zamyka program.



Tak, jak w poprzednich przykładach, użytkownik może obejść dom dookoła i przyrzeć mu się ze wszystkich stron. Domowi przypisano niewielką intensywność światła otoczenia, a dodatkowo oświetla go kilka silnych, punktowych świateł, dających przedsmak efektów oświetleniowych, które można przygotować w grze.

Dodatkowe pomysły

W tym podrozdziale omówię kilka sposobów pozwalających na poprawienie technik mapowania tekstur i efektów oświetlenia, które tworzyliśmy w tym rozdziale.

Prawdę powiedziawszy, ledwo dotknęliśmy powierzchni (dosłownie i w przenośni) problemu. W przypadku cieniowanych powierzchni na przykład cieniowaliśmy tylko tekstury, podczas gdy można przygotować również wiele innych efektów, w tym efekty wyliczane na bieżąco w trakcie działania programu. Dla przykładu moglibyśmy zaprogramować efekt „spalania”, wywoływany, gdy gracz skieruje miotacz ognia na ścianę, albo też dodać pojawiające się od czasu do czasu „rysy” na powierzchni, żeby tekstura ścian nie była tak monotonna. Można by także pokazać na podłodze ślady eksplozji w miejscu, gdzie gracz zniszczył wrogiego robota.

Sugerowanie głębi

Kolejnym trikiem związanym z oświetleniem jest wprowadzenie efektu, który sprawi, że obiekty znajdujące się dalej będą się wydawały ciemniejsze niż te blisko gracza, tak jakby do coraz dalszych obiektów docierało coraz mniej światła. Technika ta nazywana jest *sugerowaniem głębi* (ang. *depth cueing*) i została spopularyzowana przez gry takie, jak *Doom* (jest też często stosowanym efektem w grach Nintendo 64). W grach takich,

jak *Doom* tekstury wraz z oddalaniem się od gracza stają się coraz ciemniejsze. Niemniej nie ma żadnego powodu, dla którego najdalsze obiekty miałyby się stawać com bardziej czarne. Kilka gier przeznaczonych dla konsoli Nintendo 64 tworzy w ten sposób efekt „mgły”, po prostu zastępując kolor czarny odcieniem szarości.

Fałszywe cienie

W kolejnym rozdziale, poświęconym obiektom trójwymiarowym, będziemy tworzyć indywidualne obiekty, które będą mogły poruszać się po naszym trójwymiarowym świecie. Czasami jednak obserwatorowi trudno jest bez żadnych dodatkowych wskazówek ustalić, jakie jest położenie obiektu względem podłogi i dlatego obiekt przemieszczający się w pionie może sprawiać wrażenie, jakby pływał po powierzchni. Typową techniką wykorzystywaną do korygowania tego fałszywego złudzenia jest rysowanie pod obiektem okrągłego „fałszywego” cienia. Mówimy, że jest to fałszywy cień, ponieważ nie reprezentuje on kształtu obiektu (nie będzie na przykład oddawał wystającego ramienia robota), niemniej ma tę zaletę, że daje nam sposób na szybkie przygotowanie wizualnej wskazówki dla gracza, gdzie właściwie znajduje się obiekt.

Mapowanie MIP

W przypadku powierzchni wykorzystywanych w grafice trójwymiarowej wielkim marznotrawstwem czasu i pamięci jest tworzenie wielkich powierzchni dla wielokątów, które znajdują się bardzo daleko. Prawdę powiedziawszy, sensownym rozwiązaniem byłoby, aby powierzchnie, które są bardzo daleko od kamery, miały mniejsze rozmiary.

Technika ta zwana jest *Mapowaniem MIP* (ang. *MIP mapping*). Dla ciekawych powiem, że *MIP* to skrót od łacińskich słów *Multum in Parvum*, które na polski tłumaczy się „wiele w jednym”. Dla odległych wielokątów będą przygotowywane mniejsze powierzchnie, a w miarę ich zbliżania się do kamery będą one w razie potrzeby odpowiednio powiększane. Technika ta pozwala zaoszczędzić wiele pamięci i sprawia, że powierzchnie bardziej oddalone od kamery wyglądają na gładsze i bardziej jednolite.

Interpolacja dwuliniowa

Jeżeli uruchomimy przykłady zaprezentowane w tym rozdziale, zauważymy, że gdy zbliżymy się zbytnio do tekstury, poszczególne teksele powiększą się do rozmiarów małych kwadratów. Podobnie, gdy jesteśmy zbyt daleko od tekstury, pojawią się nieistniejące kształty (artefakty), ponieważ nie każdy teksel powierzchni będzie rysowany na ekranie. Jest to szczególnie zauważalne, jeśli w teksturach pojawiają się ostre, wyraźne linie.

Interpolacja dwuliniowa (lub bilinear) pozwala zaradzić częściowo obu tym problemom. Na podobnej zasadzie, co wykonywana przez nas wcześniej interpolacja między dwiema sąsiadującymi wartościami w mapie cieniowania, można wykonywać interpolację między dwoma sąsiadującymi tekselami tekstury. Wygładzimy w ten sposób tekstury bliskich obiektów, aby ich teksele nie były takie wielkie, oraz odległe tekstury, aby nie pojawiało się tam aż tyle szczegółów.

Tekstury prezentowane w tej książce przygotowałem w ten sposób, żeby nie zawierały zbyt wielu ostrzych linii, tak więc efekt przypadkowych kształtów pojawiających się, gdy jesteśmy w dużej odległości od tekstuury, został zminimalizowany. Jednak mimo to po zastosowaniu interpolacji dwuliniowej wyglądałyby one lepiej. Jeśli skorzystamy z czegoś w rodzaju testowej tekstuury w formie „barwnego wzoru”, zauważymy, że gdy oddalimy ją od kamery, pojawiać się wszelkiego rodzaju dziwnie wyglądające kształty. Interpolacja dwuliniowa pozwoliłaby na eliminację wielu z takich niepożądanych artefaktów.

Interpolacja trójliniowa

Interpolacja trójliniowa (lub trylinearna) jest zasadniczo połączeniem interpolacji dwuliniowej i mapowania MIP. Zamiast interpolacji między sąsiadującymi tekselami tekstuury w technice tej wykonywana jest interpolacja między sąsiadującymi mapami MIP (jedna mapa wykorzystywana w interpolacji MIP znajduje się dalej od kamery, a druga bliżej). W ten sposób eliminujemy zauważalny efekt przeskoku podczas przechodzenia od jednej mapy MIP do drugiej, ponieważ wykorzystujemy obie mapy MIP do stworzenia łagodnego przejścia między nimi za pomocą interpolacji.

Mapy wektorów normalnych i mapy głębokości

Niektóre z tekstuur wykorzystywanych w tym rozdziale mają odrobinę własnego, „fałszywego” oświetlenia, wbudowanego w samą teksturę w celu uzyskania wrażenia głębi. Na przykład wystające kamienie są odrobinę jaśniejsze, a przy krawędziach skał domałowane są cienie.

Oczywiście to fałszywe oświetlenie nie jest zgodne z rzeczywistością, ponieważ nie uwzględnia rzeczywistych źródeł światła, znajdujących się w otoczeniu tekstuury. W takim „fałszywym” oświetleniu na przykład z góry zakładane jest, że główne źródło światła oświetla ją od strony lewego górnego rogu, podczas gdy w rzeczywistości źródło światła może znajdować się w zupełnie innym miejscu.

Niektóre z kart akceleratorów grafiki 3D potrafią wykorzystywać mapy wektorów normalnych, by za ich pomocą tworzyć tekstuury, które uwzględniają położenie źródeł światła w otoczeniu tekstuury.

Zamiast mapy tekstuury mamy w tym przypadku trzy osobne mapy: mapę kolorów (bez żadnego oświetlenia), mapę głębokości (definiującą „głębokość” każdego piksela z mapy kolorów) i mapę wektorów normalnych (definiującą wektor normalny każdego piksela w mapie kolorów). Korzystając z koloru, wysokości i wektora normalnego, można wyliczyć właściwe oświetlenie dla każdego piksela, w efekcie czego otrzymamy teksturę, która będzie uwzględniać rzeczywiste światło z otoczenia tekstuury.

Nawet jeśli jest to dość kosztowny proces, niektóre karty akceleratorów grafiki 3D potrafią łączyć te mapy na tyle szybko, by można z nich było korzystać w czasie rzeczywistym.

Inne typy oświetlenia

Ostatnią kwestią, o której warto wspomnieć, są źródła światła. W tym rozdziale używaliśmy tylko punktowych źródeł światła, które wysyłają światło we wszystkich kierunkach, tak jak słońce. Można jednak rozbudować model oświetlenia, dodając kolorowe światła, światła oświetlające scenę stożkowym snopem światła (zamiast świecić jednakowo we wszystkich kierunkach) lub światła mające pewną powierzchnię, które mogą na przykład mieć prostokątny kształt i oświetlać scenę strumieniem światła w kształcie ściany piramidy.

Ponadto, choć nie było tego widać w naszych przykładach, nasze promienie światła mogły przechodzić przez jeden wielokąt i oświetlać inny wielokąt po jego drugiej stronie. Dobrze by było usunąć ten niepożądany efekt, upewniając się, że promień światła, który oświetla powierzchnię, nie przeszedł wcześniej przez żadną inną powierzchnię.

Podsumowanie

W tym rozdziale poruszyłem bardzo wiele spraw, włączając w to szybkie mapowanie tekstur, oświetlenie rozproszone i odpowiednie oświetlanie wielokątów za pomocą map cieniowania. Po drodze omówiłem też wiele technik optymalizacyjnych, problem efektywnego przechowywania cieniowanych tekstur i przechowywanie powierzchni w pamięci podręcznej przy wykorzystaniu miękkich odwołań. Wspomniałem również o paru dodatkowych pomysłach na poprawienie mechanizmów renderujących poprzez dodanie takich operacji, jak mapowanie MIP, interpolacja dwuliniowa i pewne zaawansowane techniki oświetlenia.

W kolejnym rozdziale skoncentrujemy się na animowaniu i poruszaniu trójwymiarowych obiektów występujących w grze. Dowiesz się także, w jaki sposób mechanizm renderujący za pomocą z-bufora jest sobie w stanie poradzić również z obiekty innymi niż tylko niezasłaniające się, wypukłe bryły.

Rozdział 9.

Obiekty trójwymiarowe

W tym rozdziale:

- ◆ Usuwanie ukrytych powierzchni.
- ◆ Animacja trójwymiarowa.
- ◆ Grupy wielokątów.
- ◆ Wczytywanie grup wielokątów z pliku *OBJ*.
- ◆ Obiekty w grze.
- ◆ Zarządzanie obiektami w grze.
- ◆ Łączenie elementów.
- ◆ Możliwe rozszerzenia w przyszłości.
- ◆ Podsumowanie.

W typowej grze gracz ma do czynienia z obiektami, od których może się odbić, które może podnieść, które może przeskoczyć, na które może wskoczyć, z którymi może walczyć, które może zjeść, złapać, odepchnąć, trącić, rzucić, zastrzelić lub z którymi może zrobić cokolwiek innego.

W tym rozdziale stworzymy system animacji obiektów trójwymiarowych, w którym będzie można przemieszczać i obracać obiekty trójwymiarowe. Do rysowania tego typu obiektów będziemy wykorzystywać z-buforowanie — w celu wyświetlania wielokątów we właściwej z-kolejności (ang. *z-order*). Stworzymy także parser wczytujący obiekty z plików *OBJ*, dzięki czemu nie będziemy musieli definiować mnóstwa wielokątów w kodzie Javy. Na końcu połączymy zebrane doświadczenia i stworzymy kilka prostych obiektów, które mogą się przydać podczas pisania gier, a także menedżera takich obiektów. Poznanie technik omówionych w tym rozdziale zbliży nas do tworzenia prawdziwej gry trójwymiarowej, w której gracz będzie miał możliwość oddziaływania na pewne obiekty.

Usuwanie ukrytych powierzchni

Problem związany z rysowaniem wielokątów polega na tym, że powinniśmy je wyświetlać w taki sposób, by nie były widoczne wielokąty całkowicie ukryte lub częściowo zasłonięte przez inne wielokąty. Problem ten jest zwykle rozwiązywany za pomocą odpowiedniego algorytmu usuwania ukrytych powierzchni lub za pomocą połączenia kilku takich algorytmów.

W poprzednim rozdziale nasz algorytm usuwania ukrytych powierzchni usuwał i przykrywał jedynie tył renderowanego obiektu. Wykorzystanie takiego algorytmu sprawiało, że mogliśmy poprawnie rysować wyłącznie wielościany wypukłe. W tym podrozdziale omówimy kilka innych technik usuwania ukrytych powierzchni, które umożliwiają rysowanie wielokątów tworzących dowolny kształt.

Pamiętaj, że różne algorytmy usuwania ukrytych powierzchni sprawdzają się w różnych sytuacjach. To, co daje doskonały efekt w przypadku obiektów statycznych, może się nie sprawdzać np. w przypadku poruszających się obiektów trójwymiarowych. W tym rozdziale skupimy się na algorytmie, który spełnia swoje zadanie w przypadku przemieszczanych i obracanych obiektów trójwymiarowych.

Algorytm malarza

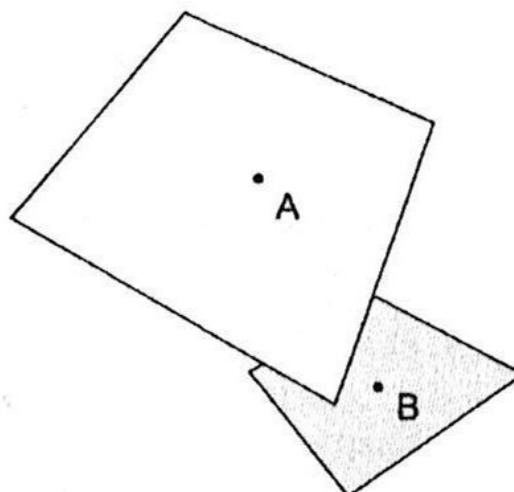
Podstawowym algorymem usuwania ukrytych powierzchni jest tzw. *algorytm malarza*. Malując obraz na płótnie, malarz w pierwszej kolejności pracuje nad tłem, później maluje obiekty znajdujące się bliżej pierwszego planu i dopiero na końcu tworzy obiekty pierwszoplanowe. Ta technika oparta jest na założeniu, że każdy rysowany obiekt znajduje się bliżej pierwszego planu niż wszystkie elementy namalowane na płótnie wcześniej. W przypadku grafiki trójwymiarowej omawiana idea przekłada się na rysowanie wielokątów w kolejności od tyłu do pierwszego planu.

Implementacja algorytmu malarza wymaga w pierwszej kolejności posortowania wszystkich wielokątów od ostatniego do pierwszego planu, zgodnie z pozycją kamery. Pewną trudnością jest w tym przypadku znalezienie właściwego algorytmu sortującego. Możemy po prostu posortować wielokąty na podstawie położenia ich punktów środkowych, jednak takie rozwiązanie nie sprawdzi się we wszystkich przypadkach. Przykładowo przeanalizuj problem wyświetlania dwóch wielokątów przedstawionych na rysunku 9.1.

W przypadku dwóch wielokątów z rysunku 9.1 możemy bez trudu stwierdzić, że wielokąt *A* znajduje się przed wielokątem *B*, zatem właśnie wielokąt *B* powinien być narysowany jako pierwszy. Problemem jest jednak zmierzenie odległości dzielącej wielokąty od kamery. Wykorzystanie do tego celu punktów środkowych wielokątów spowoduje, że będziemy mogli dojść do wniosku, że punkt środkowy wielokąta *A* znajduje się za punktem środkowym wielokąta *B*, co oznaczałoby, że to wielokąt *A* powinien być narysowany jako pierwszy. Byłoby to oczywiście nieprawidłowe rozwiązanie.

Rysunek 9.1.

W przypadku tych dwóch wielokątów algorytm malarza określi, że wielokąt B powinien być rysowany jako pierwszy, ponieważ znajduje się za wielokątem A



Kolejnym problemem związanym z algorymem malarza jest fakt, że rysowanie wielokątów od ostatniego do pierwszego planu może się wiązać z nadmiarem pracy (rysowaniem poszczególnych pikseli więcej niż raz), nie jest to więc najlepsze rozwiązanie w przypadku rysowania całej sceny.

Odwrotny algorytm malarza

Problemu nadmiaru pracy związanego z algorymem malarza można uniknąć, stosując tzw. *odwrotny algorytm malarza*. W przeciwieństwie do omówionego algorytmu malarza algorytm odwrotny rysuje wielokąty począwszy od tych, które znajdują się na pierwszym planie i skończywszy na wielokątach znajdujących się dalej od kamery. Odwrotny algorytm malarza nie wymaga rysowania żadnego piksela, który został już narysowany, można go zatem używać bez jakichkolwiek zbędnych nakładów pracy. Wystarczy narysować kolejno wielokąty aż do wypełnienia całego okna sceny. Pamiętaj jednak, że omawiany algorytm nadal wymaga zastosowania stu procentowo skutecznej techniki sortowania wielokątów. Odwrotny algorytm malarza sprawdza się tylko w sytuacjach, w których wielokąty całkowicie wypełniają obraz obejmowany przez kamerę (jak w przypadku scen w pomieszczeniach zamkniętych).

Z-bufor

Z-buforowanie, zwane także *buffersowaniem głębi*, pozwala uniknąć problemów charakterystycznych dla algorytmu malarza i jest stosunkowo łatwe w implementacji. W technice z-buforowania głębokość każdego piksela należącego do wielokąta jest zapisywana w buforze o takim samym rozmiarze, jak wyświetlane okno. Piksel jest rysowany tylko w przypadku, gdy jego głębokość jest mniejsza od aktualnie przechowywanej w odpowiednim miejscu z-bufora. W efekcie otrzymujemy doskonałą pod względem wyświetlanych pikseli scenę trójwymiarową, niezależnie od kolejności rysowania wielokątów. Ideę z-buforowania przedstawia rysunek 9.2.

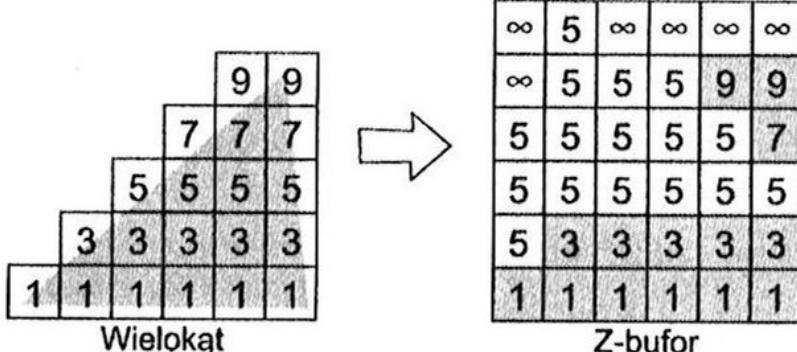
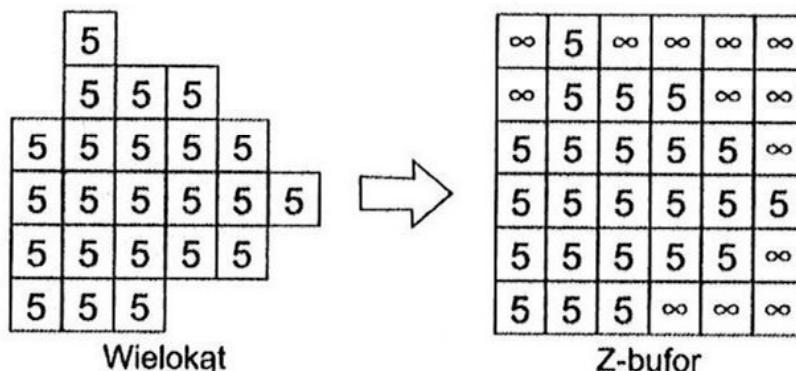
W sytuacji przedstawionej na rysunku 9.2 z-bufor był początkowo pusty — głębokości wszystkich pikseli były reprezentowane przez nieskończoności. Jest więc oczywiste, że każdy piksel pierwszego wielokąta znajduje się bliżej pierwszego planu niż odpowiednie wartości w z-buforze, zatem każdy piksel tego wielokąta jest rysowany, natomiast

Rysunek 9.2.

Z-bufor można wykorzystać do prawidłowego rysowania wielokątów (liczby reprezentują głębokość poszczególnych pikseli należących do wielokątów)

∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞

Z-bufor



odpowiednie pozycje z-bufora przyjmują wartości reprezentujące głębokości pikseli wielokąta. Drugi wielokąt przecina się z pierwszym, zatem jego część znajdująca się za pierwszym wielokątem nie jest rysowana.

Technika z-buforowania nie wymaga nadmiernych nakładów pamięci do przechowywania bufora. Obraz o 16-bitowej głębi kolorów i rozdzielcości 1024×768 będzie więc wymagał 1,5 MB pamięci. Nie jest to dużo, szczególnie jeśli porównamy tę wartość z zasobami pamięciowymi większości współczesnych komputerów.

Główną wadą tego rozwiązania jest dodatkowy czas potrzebny do rysowania wielokątów. Sprawdzanie wartości w z-buforze przed narysowaniem każdego piksela wymaga pewnych dodatkowych nakładów czasowych. Związanego z tym opóźnienie jest szczególnie widoczne, jeśli technika z-buforowania jest wykorzystywana dla każdego piksela pojawiającego się na ekranie. Z doświadczeń przeprowadzonych na przykładach prezentowanych w tej książce wynika, że z-buforowanie każdego rysowanego na ekranie piksela jest dwa do trzech razy wolniejsze niż zwykłe mapowanie tekstur.

Ponieważ jednak obiekty trójwymiarowe zajmują zwykle niewielką część ekranu, opóźnienia w wyświetlaniu wielokątów nie będą tak duże, jak w przypadku z-buforowania scen wypełniających cały ekran.

Z-bufor z wartościami 1/z

Jednym z problemów związanych z z-buforowaniem jest brak zależności precyzji zapisów z-bufora od odległości wielokąta od kamery. Przykładowo, jeśli precyza z-bufora jest na poziomie jednej jednostki (lub jednego tekscela), zachodzące na siebie wielokąty znajdujące się blisko kamery nie będą wyświetlane właściwie — część wielokąta może przecież mieć głębokość 10,7, a nie 11. Do wyświetlania wielokątów znajdujących się bliżej kamery potrzebujemy czegoś bardziej dokładnego. Oczekiwany efekt można uzyskać na kilka sposobów:

- ◆ Zastosować 32-bitowe liczby zmienoprzecinkowe (lub 32-bitowe liczby całkowite) do reprezentowania głębokości w z-buforze. Uzyskujemy w ten sposób większą dokładność kosztem dwukrotnie większego zużycia pamięci i spowolnienia procesu wyświetlania wielokątów.
 - ◆ Zastosować odwrotność głębokości, czyli wartości $1/z$.

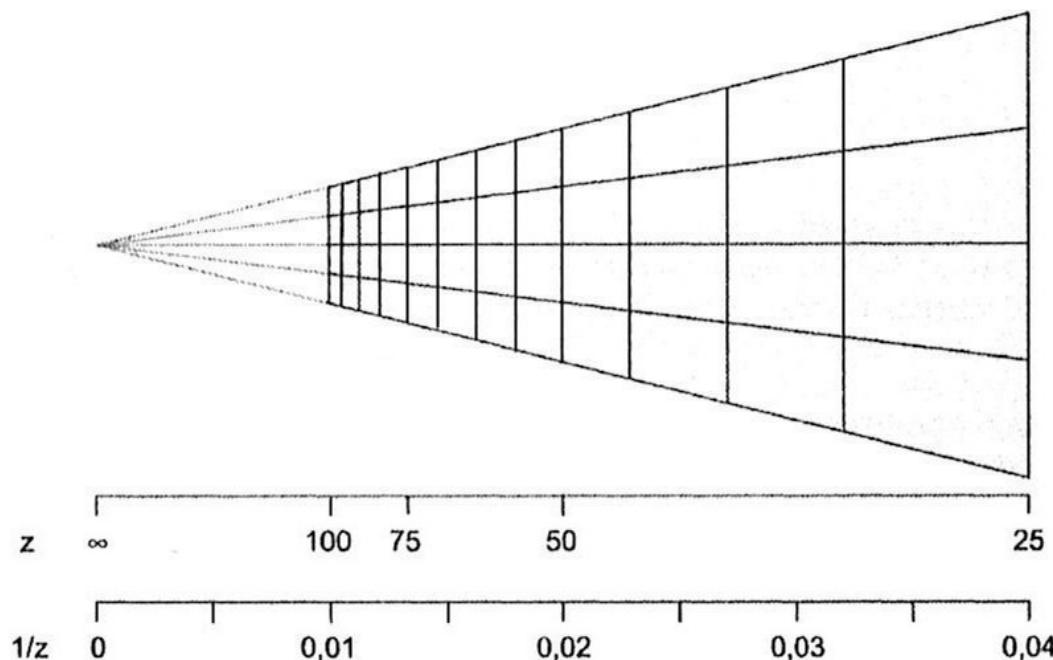
Technika z-buforowania nie wymaga przechowywania dokładnych głębokości dla każdego piksela należącego do wielokąta. W rzeczywistości konieczne jest jedynie właściwe porównywanie głębokości niektórych pikseli, co możemy robić także na podstawie wartości $1/z$. Wartości $1/z$ są liniowe w przestrzeni obrazu, co widać na przykładzie wielokąta z rysunku 9.3. Technika z-buforowania oparta na takim rozwiążaniu ma kilka zalet:

- ◆ Zastosowanie wartości $1/z$ daje nam większą precyzję reprezentacji głębokości obiektów znajdujących się bliżej kamery, gdzie ta precyzja ma największe znaczenie.
 - ◆ Ponieważ wartość $1/z$ zmienia się liniowo w przestrzeni obrazu, jako odwrotność głębokości z , nietrudno ustalić głębokość dla poziomej ścieżki — jest to łatwiejsze niż w przypadku mapowania tekstur (gdzie nie stosuje się wartości zmieniających się liniowo w przestrzeni obrazu). Nie jest konieczne interpolowanie każdych 16 pikseli, jak w przypadku techniki mapowania tekstur, zatem cały proces jest szybki i dokładny.

Rysunek 9.3.

Wartości 1/z

*zmieniają się liniowo
w przestrzeni obrazu*



Właśnie technikę z-buforowania z wartościami $1/z$ wykorzystujemy w tej książce dla obiektów trójwymiarowych. Ponieważ do reprezentowania głębokości wykorzystujemy 16-bitowe liczby całkowite, będziemy się starali utrzymać wartości w buforze w przedziale od 0 do Short.MAX_VALUE. Zakładamy, że $z \geq 1$ i używamy wyrażenia:

```
Short.MAX_VALUE / z
```

Powyższy wzór oznacza, że im większa jest wartość z , tym bliżej pierwszego planu znajduje się dany piksel; wartość 0 reprezentuje nieskończoną głębokość.

Spróbujemy teraz zaimplementować z-bufor. Klasa ZBuffer (patrz listing 9.1) tworzy prosty, 16-bitowy z-bufor, zaprojektowany z myślą o przechowywaniu wartości $1/z$.

Listing 9.1. ZBuffer.java

```
package com.brackeen.javagamebook.graphics3D;

/**
 * Klasa ZBuffer implementuje z-bufor, czyli bufor głębi, w którym są
 * przechowywane głębokości wszystkich pikseli obiektów trójwymiarowych
 * wyświetlanych na ekranie. Wartość zapisana dla każdego piksela jest
 * odwrotnością jego głębokości ( $1/z$ ). zatem precyzja obiektów znajdujących
 * się bliżej pierwszego planu jest większa niż w przypadku
 * obiektów znajdujących się dalej od kamery (gdzie duża precyzja
 * głębokości jest mniej istotna z punktu widzenia jakości obrazu).
 */
public class ZBuffer {

    private short[] depthBuffer;
    private int width;
    private int height;

    /**
     * Tworzy nowy z-bufor o określonej szerokości i wysokości.
     */
    public ZBuffer(int width, int height) {
        depthBuffer = new short[width*height];
        this.width = width;
        this.height = height;
        clear();
    }

    /**
     * Zwraca szerokość tego z-bufora.
     */
    public int getWidth() {
        return width;
    }

    /**
     * Zwraca wysokość tego z-bufora.
     */
    public int getHeight() {
        return height;
    }
}
```

```
/**  
 * Zwraca tablicę używaną do przechowywania tego bufora głębi.  
 */  
public short[] getArray() {  
    return depthBuffer;  
}  
  
/**  
 * Czyści z-bufor. Wszystkie głębokości będą miały wartość 0.  
 */  
public void clear() {  
    for (int i=0; i<depthBuffer.length; i++) {  
        depthBuffer[i] = 0;  
    }  
}  
  
/**  
 * Ustawia głębokość dla piksela w danej pozycji,  
 * zastępując dotychczasową wartość w buforze.  
 */  
public void setDepth(int offset, short depth) {  
    depthBuffer[offset] = depth;  
}  
  
/**  
 * Sprawdza głębokość piksela w danej pozycji, jeśli przekazana  
 * głębokość jest mniejsza (przekazana wartość jest większa lub  
 * równa wartości przechowywanej w z-buforze na danej pozycji);  
 * wówczas w buforze zapisywana jest nowa głębokość, a metoda  
 * zwraca wartość true. W przeciwnym przypadku w buforze nie są  
 * wprowadzane żadne zmiany, a metoda zwraca wartość false.  
 */  
public boolean checkDepth(int offset, short depth) {  
    if (depth >= depthBuffer[offset]) {  
        depthBuffer[offset] = depth;  
        return true;  
    }  
    else {  
        return false;  
    }  
}
```

Zauważ, że klasa ZBuffer zawiera dwie metody modyfikujące zawartość określonej pozycji z-bufora: `setDepth()` oraz `checkDepth()`. Metoda `setDepth()` jedynie ustawia głębokość piksela reprezentowanego na danej pozycji w z-buforze, usuwając wartość przechowywaną tam wcześniej. Metoda `checkDepth()` ustawia nową głębokość tylko w przypadku, gdy nowa wartość jest większa od głębokości przechowywanej na danej głębokości do tej pory; metoda zwraca wówczas wartość `true`, natomiast w przeciwnym przypadku zwraca wartość `false`.

Metoda `clear()` ustawia wszystkie wartości przechowywane w z-buforze na 0, czyli równaźność nieskończonej głębokości. Oznacza to, że należy wywoływać metodę `clear()` dla każdej wyświetlnej animacji.

Obliczanie z-głębokości

Kolejnym krokiem jest znalezienie równań umożliwiających wyznaczanie głębokości poszczególnych punktów wielokąta.

W rzeczywistości znamy już takie równanie. W rozdziale 8. — „Mapowanie tekstur i oświetlenie” — potrzebowaliśmy wartości P_z , czyli głębokości wielokąta w danym punkcie wyświetlanego okna obrazu W , dzięki czemu mogliśmy wyznaczyć sposób mapowania tekstur:

$$P_z = d(U \times V \bullet O) / (U \times V \bullet W)$$

W powyższym równaniu zmienna d reprezentuje odległość dzielącą kamerę od okna obrazu, U i V to współrzędne tekstury w kierunkach x i y , natomiast O jest początkiem układu współrzędnych tekstury (lub dowolnym punktem na płaszczyźnie wielokąta). Mamy więc wszystko, czego potrzebujemy (okazuje się, że warto było poświęcić nieco czasu na wyprowadzenie tych równań w poprzednim rozdziale). Po odwróceniu obu stron równania otrzymujemy:

$$1/z = (U \times V \bullet W) / (d(U \times V \bullet O))$$

Większość elementów równania możemy wyliczać dla całego wielokąta, nie dla pojedynczych pikseli:

```
float w = SCALE * MIN_DISTANCE * Short.MAX_VALUE / (viewWindow.getDistance()
    * c.getDotProduct(textureBounds.getOrigin()));
```

Pamiętaj, że wektor C przechowuje wartości $U \times V$, zatem nie musimy ponownie wyznaczać $U \times V$.

Wystarczy pomnożyć otrzymaną w przedstawionym równaniu wartość przez `SCALE`, aby otrzymać wartość całkowitoliczbową.

Wynik mnożymy także przez wartość `MIN_DISTANCE`, czyli minimalną odległość, dla której ma być wykorzystywana technika z-buforowania. Wykorzystujemy wartość 12. Dzięki temu otrzymujemy nieco większą dokładność dla obiektów rysowanych dalej od pierwszego planu, jednak obiekty znajdujące się w mniejszej odległości od kamery niż 12 mają nieprawidłowe głębokości. Takie rozwiązanie jest logiczne, ponieważ i tak nigdy nie będziemy widzieli obiektów znajdujących się bliżej niż w odległości 12, kiedy zaimplementujemy wykrywanie kolizji w rozdziale 11. — „Wykrywanie kolizji”.

W przypadku stosowania techniki mapowania tekstur konieczne jest — przed mapowaniem tekstur — obliczenie głębokości pierwszego piksela (`depth`) i zmiana głębokości (`dDepth`):

```
float z = c.getDotProduct(viewPos);
float dz = c.x;
...
int depth = (int)(w*z);
int dDepth = (int)(w*dz);
```

Następnie sprawdzamy głębokość dla każdego piksela, ustawiamy piksel (jeśli jego głębokość jest ustalona) i zwiększamy głębokość:

```
...
if (zBuffer.checkDepth(offset, (short)(depth >> SCALE_BITS))) {
    doubleBufferData[offset] = texture.getColor(tx >> SCALE_BITS, ty >> SCALE_BITS);
}
depth += dDepth;
...
```

To wystarczy, by narysować piksele z wykorzystaniem z-bufora. W klasie ZBufferedRenderer, która jest podklassą klasy ShadedSurfacePolygonRenderer, gromadzimy wyniki obliczeń głębokości dla wszystkich pikseli należących do wielokąta. Klasa ShadedSurfacePolygonRenderer działa jak klasa bazowa, z wyjątkiem procesu renderowania, kiedy klasa sprawdza głębokość wszystkich rysowanych pikseli. Zadaniem klasy jest także czyszczenie z-bufora przed narysowaniem każdej klatki (lub stworzenie nowej, jeśli rozmiar okna uległ zmianie):

```
public void startFrame(Graphics2D g) {
    super.startFrame(g);
    // Inicjalizuje bufor głębi.
    if (zBuffer == null ||
        zBuffer.getWidth() != viewWindow.getWidth() ||
        zBuffer.getHeight() != viewWindow.getHeight())
    {
        zBuffer = new ZBuffer(viewWindow.getWidth(), viewWindow.getHeight());
    }
    else if (clearViewEveryFrame) {
        zBuffer.clear();
    }
}
```

To wszystko, co musimy zrobić, aby obsłużyć proces z-buforowania. Zajmiemy się teraz wykorzystaniem tej techniki do tworzenia trójwymiarowych obiektów poruszających się na scenie.

Animacja trójwymiarowa

Na początku rozdziału 7. — „Grafika trójwymiarowa” — stworzyliśmy proste drzewo, które obracając się wokół osi pionowej, przybliżało się i oddalało od kamery. W tamtej wersji nie wprowadzaliśmy jeszcze ruchu kamery; zamiast tego przesuwaliśmy wielokąty tworzące drzewo.

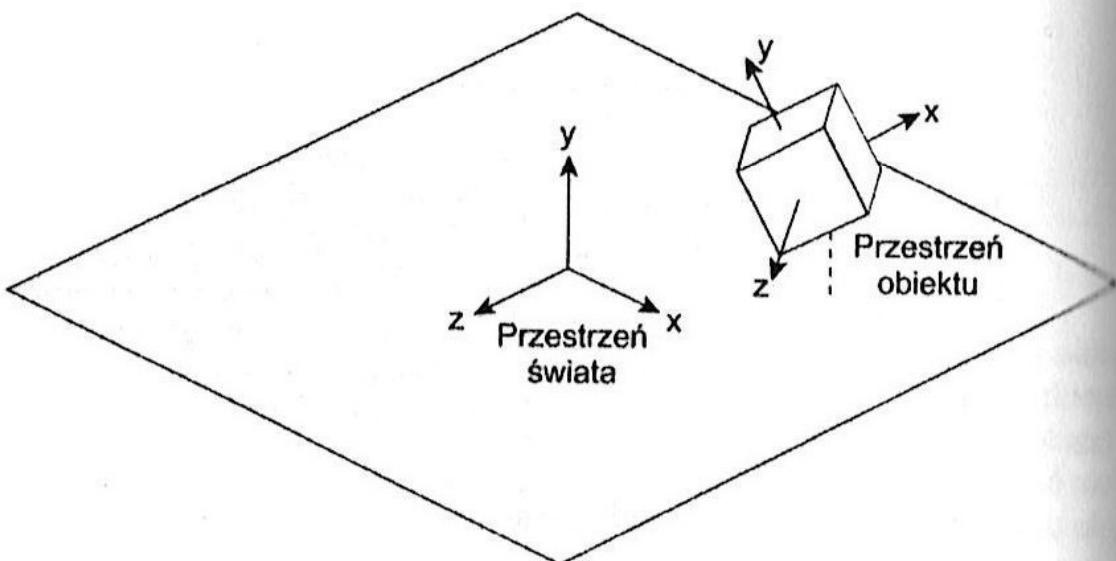
Tą samą technikę możemy wykorzystać do przesuwania i obracania dowolnych obiektów trójwymiarowych występujących w grze — każdy trójwymiarowy obiekt może się poruszać według charakterystycznego dla siebie schematu. Przyjmujemy, że obiektem trójwymiarowym jest zbiór powiązanych ze sobą wielokątów — często używa się

też określenia *siatki wielokątów* lub *modelu wielokątów*. Taka grupa wielokątów może się poruszać niezależnie od pozostałych elementów sceny trójwymiarowej. Przykładowo wszystkie wielokąty składające się na trójwymiarowy model robota należą do tego samego obiektu trójwymiarowego, zatem będą się poruszać według tego samego schematu.

Taka technika wymaga stosowania dla trójwymiarowego obiektu jego własnego układu współrzędnych, który różni się od układu współrzędnych pozostałych elementów sceny (patrz rysunek 9.4).

Rysunek 9.4.

Trójwymiarowe obiekty stosują inne układy współrzędnych niż pozostałe elementy sceny



Nie przesuwamy jawnie i nie obracamy wielokątów tworzących trójwymiarowe obiekty — zamiast tego wielokąty pozostają statyczne, a zmienia się jedynie położenie samego obiektu. Jak wspomnieliśmy w poprzednich rozdziałach, gdybyś zmieniał położenie wielokątów należących do trójwymiarowych obiektów i obracał je, po jakimś czasie błędy reprezentacji liczb zmiennoprzecinkowych skumulowałyby się, powodując znieskończalanie obiektu. Zamiast tego podczas procesu renderowania kopujemy wszystkie wielokąty do tymczasowych reprezentacji i przekształcamy je zgodnie ze schematem obowiązującym dla tworzonego przez nie obiektu (patrz przykład drzewa z rozdziału 7.).

Aby otrzymać efekt niezależnego przemieszczania się obiektów w naszym mechanizmie grafiki trójwymiarowej, wystarczy każdemu z tych obiektów przypisać własną klasę Transform3D z rozdziału 7.

Jeśli jednak szukasz prostego sposobu wprowadzenie efektu ruchu dla swojego obiektu, możesz po prostu rozbudować klasę Transform3D, aby dodać różne typy ruchu obiektu na scenie.

Pamiętaj, że istnieją dwa rodzaje ruchu: ruch postępowy (zmiana położenia obiektu) oraz ruch obrotowy (obracanie obiektu wokół osi). Istnieją także trzy rodzaje ruchu obrotowego, po jednym dla każdej z osi: x, y lub z.

Co więcej, możesz zaplanować, by różne obiekty poruszały się i obracały z różnymi szybkościami oraz by niektóre obiekty przemieszczały się lub obracały przez pewien okres czasu, po czym zatrzymywały się. Przykładowo możesz nakazać robotowi obrót do momentu, w którym będzie on ustawiony przodem do określonego kierunku, po czym

zatrzymanie się lub przejście w kierunku gracza i zatrzymanie się. Mając na uwadze tego typu możliwości w zakresie poruszania trójwymiarowych obiektów na scenie, spróbujmy zrealizować bardziej zaawansowane operacje przemieszczania obiektów:

- ◆ Przesuń obiekt do punktu (x, y, z) z szybkością s .
- ◆ Przesuwaj obiekt w kierunku punktu (x, y, z) z szybkością s przez kolejne t sekund lub bez przerwy.
- ◆ Obracaj obiekt wokół jednej z osi z szybkością s przez kolejne t sekund lub bez przerwy.
- ◆ Obracaj obiekt z szybkością s , aż znajdzie się przodem do kierunku $3\pi/4$ radianów.
- ◆ Zatrzymaj ruch obiektu.

Wszystkie powyższe zadania przemieszczania i obracania obiektów można sprowadzić do krótkiego polecenia: „przemieszczaj obiekt z szybkością x przez t czasu”. Po upływie określonej ilości czasu ruch obiektu jest przerywany (pamiętaj jednak, że niektóre zadania nie mają limitu czasu, zatem konieczne jest znalezienie sposobu obsługi stałego ruchu). Poza zdefiniowaniem szybkości i ilości czasu ruch postępowy wymaga także określenia kierunku.

Wszystkie opisane zadania możemy zrealizować w podklasie klasy `Transform3D`, zwanej `MovingTransform3D`. Najważniejszym elementem klasy `MovingTransform3D` jest wewnętrzna podkласa `Movement` (patrz listing 9.2).

Listing 9.2. *Movement, wewnętrzna podklasa klasy MovingTransform3D*

```
/***
 * Klasa Movement przechowuje szybkość oraz czas, przez jaki
 * obiekt ma się poruszać z tą szybkością.
 */
protected static class Movement {
    // Zmiana na milisekundę.
    float speed;
    long remainingTime;

    /**
     * Ustawia właściwości ruchu zgodnie z przekazaną szybkością
     * i ilością czasu (w milisekundach).
     */
    public void set(float speed, long time) {
        this.speed = speed;
        this.remainingTime = time;
    }

    public boolean isStopped() {
        return (speed == 0) || (remainingTime == 0);
    }

    /**
     * Zwraca odległość przebytą w określonym czasie (liczba milisekund).
     */
}
```

```

*/
public float getDistance(long elapsedTime) {
    if (remainingTime == 0) {
        return 0;
    }
    else if (remainingTime != FOREVER) {
        elapsedTime = Math.min(elapsedTime, remainingTime);
        remainingTime -= elapsedTime;
    }
    return speed * elapsedTime;
}

```

Klasa Movement jedynie przechowuje wartości reprezentujące szybkość i pozostały czas, w którym obiekt będzie w ruchu. Opcjonalnie pozostały czas może mieć wartość FOREVER — w takim przypadku ruch obiektu nigdy nie zostanie przerwany. Przykładowo ruch pocisku wystrzelonego w powietrzu nigdy się nie zakończy (chyba że wydamy odpowiednią dyspozycję). Metoda getDistance() zwraca odległość przebytą przez obiekt, obliczoną na podstawie liczby milisekund (pamiętaj, że $odległość = prędkość \times czas$). W tej metodzie jest także zmniejszana wartość zmiennej remainingTime o wartość przechowywaną w zmiennej elapsedTime, przynajmniej dopóki wartość tej pierwszej zmiennej nie osiągnie zera.

W klasie Movement przechowujemy jedynie pozostały czas ruchu obiektu, nie bezwzględny czas zakończenia. Gdybyśmy operowali na bezwzględnym czasie końca ruchu, elementy modyfikujące przepływ czasu w grze — np. wstrzymywanie gry lub zapisywanie i późniejsze wczytywanie jej stanu — z oczywistych względów powodowałyby nieprawidłową obsługę ruchu. Przykładowo jeśli o 7:58 wydałeś polecenie: „przemieszczaj się do przodu do godziny 8:00”, a o godzinie 7:59 użytkownik wstrzymał grę do godziny 8:01, ruch obiektu zostanie przerwany, co byłoby niezgodne z Twoimi zamierzeniami — ruch obiektu trwałby tylko minutę, chociaż chciałeś, by obiekt poruszał się przez dwie minuty.

Będziemy używali jednej kopii klasy Movement dla ruchu postępowego i trzech kopii dla ruchu obrotowego (po jednym dla każdej osi). W pierwszej kolejności przeanalizujemy i zaimplementujemy ruch postępowy.

Ruch postępowy

Podobnie jak w przypadku animacji w rozdziale 2., „Grafika 2D oraz animacja”, nadamy naszym obiektom określoną prędkość, aby otrzymać efekt ruchu postępowego. W pierwszej kolejności zdefiniujemy w klasie MovingTransform3D kilka pól reprezentujących szybkość ruchu:

```

// Szybkość (w jednostkach na milisekundę)
private Vector3D velocity;
private Movement velocityMovement;

```

Ponieważ wektor szybkości musi reprezentować zarówno wielkość, jak i kierunek, nie potrzebujemy w tym przypadku specjalnego pola dla szybkości w klasie Movement.

Szybkość w klasie Movement ma więc wartość 1, jeśli tylko wartość wektora prędkości jest różna od 0. W przeciwnym przypadku szybkość wynosi właśnie 0.

Spróbujmy teraz stworzyć kilka metod jawnie ustawiających szybkość ruchu:

```
/***
 * Ustawia wartość szybkości dla danego wektora.
 */
public void setVelocity(Vector3D v) {
    setVelocity(v, FOREVER);
}

/***
 * Ustawia szybkość. Wartość ta jest automatycznie zerowana
 * w momencie, gdy upłynie przekazana ilość czasu (liczba
 * milisekund). Jeśli przekazano wartość FOREVER, szybkość
 * nigdy nie zostanie zredukowana do zera.
 */
public void setVelocity(Vector3D v, long time) {
    if (velocity != v) {
        velocity.setTo(v);
    }
    if (v.x == 0 && v.y == 0 && v.z == 0) {
        velocityMovement.set(0, 0);
    }
    else {
        velocityMovement.set(1, time);
    }
}

/***
 * Dodaje przekazaną wartość do aktualnej szybkości ruchu.
 * Jeśli ten obiekt MovingTransform3D jest aktualnie w ruchu,
 * czas pozostały do zatrzymania go nie ulegnie zmianie.
 * W przeciwnym przypadku pozostały czas będzie miał wartość
 * FOREVER.
 */
public void addVelocity(Vector3D v) {
    if (isMoving()) {
        velocity.add(v);
    }
    else {
        setVelocity(v);
    }
}

/**
 * Zwraca true, jeśli obiekt jest w ruchu.
 */
public boolean isMoving() {
    return !velocityMovement.isStopped() && !velocity.equals(0,0,0);
}
```

Metody `setVelocity()` ustawiają szybkość ruchu na określoną ilość czasu lub określającą, że ruch będzie trwał stale (w przypadku wywołania wersji jednoargumentowej lub wywołania z wartością FOREVER drugiego argumentu) — w pierwszym przypadku celem

jest przerwanie ruchu obiektu po upływie określonego czasu. Za pomocą metody `addVelocity()` możesz dodać do bieżącego ruchu nowy (przykładowo do animacji poruszającego się obiektu można dodać efekt grawitacji). Ostatnia metoda, `isMoving()`, służy sprawdzeniu, czy obiekt rzeczywiście się porusza.

W kolejnym kroku napiszemy metodę jawnie definiującą cel ruchu:

```
/**
 * Ustawia wartości dla ruchu do określonego miejsca
 * z określoną szybkością.
 */
public void moveTo(Vector3D destination, float speed) {
    temp.setTo(destination);
    temp.subtract(location);

    // Oblicza czas potrzebny na przebycie wyznaczonej
    // odległości.
    float distance = temp.length();
    long time = (long)(distance / speed);

    // Normalizuje wektor kierunku.
    temp.divide(distance);
    temp.multiply(speed);

    setVelocity(temp, time);
}
```

Metoda `moveTo()` jest wygodnym rozwiązaniem służącym do wyznaczania — na podstawie przekazanej prędkości — wektora ruchu (zarówno wartości, jak i kierunku) oraz czasu potrzebnego do osiągnięcia pozycji docelowej. W metodzie wykorzystujemy wektor tymczasowy `temp`, dla którego wywołujemy metodę `setVelocity()`. Możemy w ten sposób wyznaczyć miejsce docelowe ruchu obiektu bez obliczania kierunku i potrzebnego czasu.

Ruch obrotowy

Ruch obrotowy, lub po prostu *obrót*, jest podobny do ruchu postępowego, jednak nie wymaga wektora ruchu — konieczne jest natomiast użycie trzech kopii klasy `Movement`:

```
// Prędkość ruchu obrotowego (w radianach na milisekundę):
private Movement velocityAngleX;
private Movement velocityAngleY;
private Movement velocityAngleZ;
```

Obsługa ruchu obrotowego jest niemal taka sama dla wszystkich osi, skupimy się więc na osi *y*. W pierwszej kolejności stworzymy proste metody ustawiające ruch obrotowy i sprawdzimy, czy dany obiekt aktualnie obraca się wokół osi *y*:

```
/**
 * Ustawia szybkość ruchu obrotowego wokół osi y.
 */
public void setAngleVelocityY(float speed) {
    setAngleVelocityY(speed, FOREVER);
}
```

```
/**  
 * Ustawia szybkość ruchu obrotowego wokół osi y  
 * na określony czas.  
 */  
public void setAngleVelocityY(float speed, long time) {  
    velocityAngleY.set(speed, time);  
}  
  
/**  
 * Zwraca true, jeśli obiekt aktualnie obraca się  
 * wokół osi y.  
 */  
public boolean isTurningY() {  
    return !velocityAngleY.isStopped();  
}
```

W metodach `setAngleVelocityY()` przyjęliśmy, że prędkość dodatnia jest równoważna z ruchem obrotowym w kierunku przeciwnym do ruchu wskazówek zegara; prędkość ujemna oznacza ruch zgodny z ruchem wskazówek zegara.

Znacznie ciekawsze jest jednak nakazanie obiekowi obrócenia się w taki sposób, by ustawił się przodem do wyznaczonego kierunku. W takim przypadku zawsze trzeba obliczyć, jaki obrót będzie szybszy: zgodny z ruchem wskazówek zegara czy w przeciwnym kierunku.

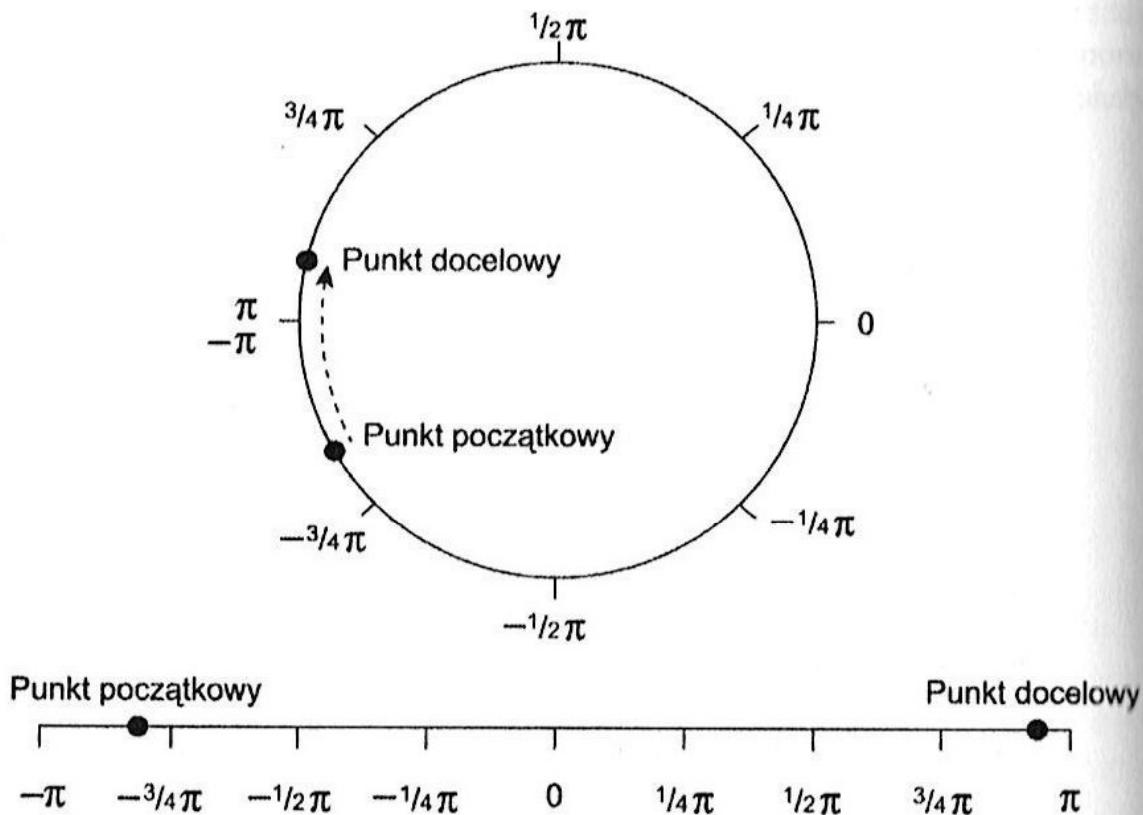
W większości przypadków rozwiązywanie tego problemu jest trywialne, jednak nie zawsze mamy do czynienia z takimi sytuacjami. Na rysunku 9.5 przedstawiliśmy przykład sytuacji, w której rozpoczynamy od jednego końca skali, blisko wartości $-\pi$, a kierunkiem docelowym jest inny koniec skali, blisko wartości π . Jeśli spojrzymy na te kąty, przedstawione graficznie w postaci okręgu, wskazanie właściwego kierunku obrotu będzie bardzo proste. Nie jest to jednak takie oczywiste, kiedy spojrzymy na liniową skalę od $-\pi$ do π (patrz dolna część rysunku 9.5). Nie chcemy przecież, by obiekt obracał się prawie o kąt pełny, jeśli nakażemy mu obrót od $-\pi$ do π .

Aby rozwiązać ten problem, obliczamy oba kąty dzielące aktualny kierunek od kierunku docelowego (dla obrotu zgodnego z ruchem wskazówek zegara i obrotu w kierunku przeciwnym). Następnie porównujemy obie wartości i obracamy obiekt w kierunku, dla którego kąt obrotu jest mniejszy. Operacja ta jest realizowana przez metodę `turnTo()`:

```
/**  
 * Obraca obiekt wokół osi y do danego kąta z określona  
 * prędkością.  
 */  
public void turnYTo(float angleDest, float speed) {  
    turnTo(velocityAngleY, getAngleY(), angleDest, speed);  
}  
  
/**  
 * Obraca z określona prędkością obiekt wokół osi y, aż  
 * przyjmie on pozycję przodem do kierunku danego wektora  
 * (x, z).  
 */
```

Rysunek 9.5.

Znajdowanie
mniejszego
kąta obrotu
pomiędzy
dwoma kątami



```
/*
public void turnYTo(float x, float z, float angleOffset, float speed)
{
    turnYTo((float)Math.atan2(-z,x) + angleOffset, speed);
}

/**
 * Zmienia z określona szybkością kąt ruchu od kąta startAngle
 * do kąta endAngle.
 */
protected void turnTo(Movement movement, float startAngle, float endAngle,
float speed) {
    startAngle = ensureAngleWithinBounds(startAngle);
    endAngle = ensureAngleWithinBounds(endAngle);
    if (startAngle == endAngle) {
        movement.set(0,0);
    }
    else {

        float distanceLeft;
        float distanceRight;
        float pi2 = (float)(2*Math.PI);

        if (startAngle < endAngle) {
            distanceLeft = startAngle - endAngle + pi2;
            distanceRight = endAngle - startAngle;
        }
        else {
            distanceLeft = startAngle - endAngle;
            distanceRight = endAngle - startAngle + pi2;
        }
    }
}
```

```

        if (distanceLeft < distanceRight) {
            speed = -Math.abs(speed);
            movement.set(speed, (long)(distanceLeft / -speed));
        }
        else {
            speed = Math.abs(speed);
            movement.set(speed, (long)(distanceRight / speed));
        }
    }

/** Upewnia się, że dany kąt należy do przedziału od -pi do pi.
 * Zwraca ten kąt lub wartość poprawioną, jeśli kąt nie mieścił się
 * w prawidłowym przedziale.
 */
protected float ensureAngleWithinBounds(float angle) {
    if (angle < -Math.PI || angle > Math.PI) {
        // Przekształca przedział do postaci od 0 do 1.
        double newAngle = (angle + Math.PI) / (2*Math.PI);
        // Poprawia przedział.
        newAngle = newAngle - Math.floor(newAngle);
        // Przekształca przedział ponownie do postaci od -pi do pi.
        newAngle = Math.PI * (newAngle * 2 - 1);
        return (float)newAngle;
    }
    return angle;
}

```

Ponieważ metoda turnTo() wymaga, by przekazywane kąty należały do przedziału od $-\pi$ do π , metoda ta wywołuje metodę ensureAngleWithinBounds(), która — w razie potrzeby — przekształca kąt w taki sposób, by mieścił się w tym przedziale.

Przeanalizowaliśmy już wszystkie najważniejsze metody klasy MovingTransform3D, resztę kodu tej klasy przedstawiamy na listingu 9.3.

Listing 9.3. Reszta klasy MovingTransform3D.java

```

package com.brackeen.javagamebook.math3D;

/**
 * Klasa MovingTransform3D jest klasą pochodną klasy Transform3D
 * i obsługuje ruch postępowy oraz ruch obrotowy wokół osi x, y i z.
 */
public class MovingTransform3D extends Transform3D {

    public static final int FOREVER = -1;

    // Vector3D, używany do obliczeń:
    private static Vector3D temp = new Vector3D();

    // Prędkość (w jednostkach na milisekundę):
    private Vector3D velocity;
    private Movement velocityMovement;
}

```

```
// Prędkość ruchu obrotowego (w radianach na milisekundę):
private Movement velocityAngleX;
private Movement velocityAngleY;
private Movement velocityAngleZ;

/**
 * Tworzy nowy obiekt MovingTransform3D
 */
public MovingTransform3D() {
    init();
}

/**
 * Tworzy nowy obiekt MovingTransform3D na podstawie tych samych
 * wartości, które określono dla Transform3D.
 */
public MovingTransform3D(Transform3D v) {
    super(v);
    init();
}

protected void init() {
    velocity = new Vector3D(0,0,0);
    velocityMovement = new Movement();
    velocityAngleX = new Movement();
    velocityAngleY = new Movement();
    velocityAngleZ = new Movement();
}

public Object clone() {
    return new MovingTransform3D(this);
}

/**
 * Aktualizuje obiekt Transform3D na podstawie przekazanego czasu ruchu,
 * który upłynął. Aktualizowane jest położenie i kąty obiektu.
 */
public void update(long elapsedTime) {
    float delta = velocityMovement.getDistance(elapsedTime);
    if (delta != 0) {
        temp.setTo(velocity);
        temp.multiply(delta);
        location.add(temp);
    }

    rotateAngle(velocityAngleX.getDistance(elapsedTime),
    velocityAngleY.getDistance(elapsedTime), velocityAngleZ.getDistance(elapsedTime));
}

/**
 * Zatrzymuje obiekt Transform3D. Wartości wszystkich wektorów ruchu
 * są zerowane.
 */
```

```
/*
public void stop() {
    velocity.setTo(0.0,0);
    velocityMovement.set(0,0);
    velocityAngleX.set(0,0);
    velocityAngleY.set(0,0);
    velocityAngleZ.set(0,0);
}

/**
 * Zwraca pozostały czas, w którym obiekt będzie w ruchu.
 */
public long getRemainingMoveTime() {
    if (!isMoving()) {
        return 0;
    }
    else {
        return velocityMovement.remainingTime;
    }
}
```

Zwróć uwagę na to, że metoda update() klasy MovingTransform3D powinna być wywoływana dla każdej klatki. Właśnie ta metoda wykonuje całą niezbędną pracę — aktualizuje pozycje obiektu w ruchu postępowym i ustawienie (w osiach x , y i z) obiektu w ruchu obrotowym.

Ponieważ same wielokąty są statyczne, musimy wykorzystać klasę MovingTransform3D za każdym razem, gdy te wielokąty są rysowane. Stworzymy teraz klasę grupującą powiązane ze sobą wielokąty i spróbujemy wprowadzić tę grupę wielokątów w ruch.

Grupy wielokątów

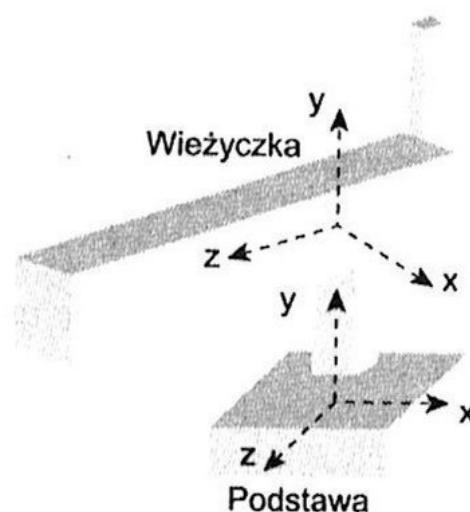
W tym podrozdziale zaimplementujemy kontener dla grup wielokątów, w którym każda taka grupa będzie się poruszała niezależnie od pozostałych.

Do tej pory rozważaliśmy obiekty trójwymiarowe wyłącznie jako całości. W rzeczywistości takie obiekty będą się jednak składały z wielu różnych części, które będą się poruszały niezależnie od siebie. Przykładowo ramię robota może się poruszać niezależnie od jego korpusu, zaś jego palce mogą się poruszać niezależnie od ramienia. Kiedy robot zacznie się przemieszczać, jego ramię będzie oczywiście przemieszczać się razem z nim (chyba że gracz wcześniej „amputuje” to ramię robota).

Aby umożliwić taki ruch obiektów, stworzymy grupy wielokątów z potomnymi grupami wielokątów, z których każda będzie mogła się poruszać niezależnie, co oznacza, że każda grupa wielokątów będzie wykorzystywała własny układ współrzędnych. Przykład takiego złożonego obiektu przedstawia rysunek 9.6. Trójwymiarowy obiekt składa się tam z dwóch grup wielokątów: „wieżyczki” i „podstawy”. Dzięki temu wieżyczka może się obracać i przemieszczać niezależnie od podstawy.

Rysunek 9.6.

Trójwymiarowe obiekty mogą się składać z wielu potomnych grup wielokątów, wykorzystujących własne układy współrzędnych. W tym przykładzie wieżyczka może się poruszać niezależnie od podstawy

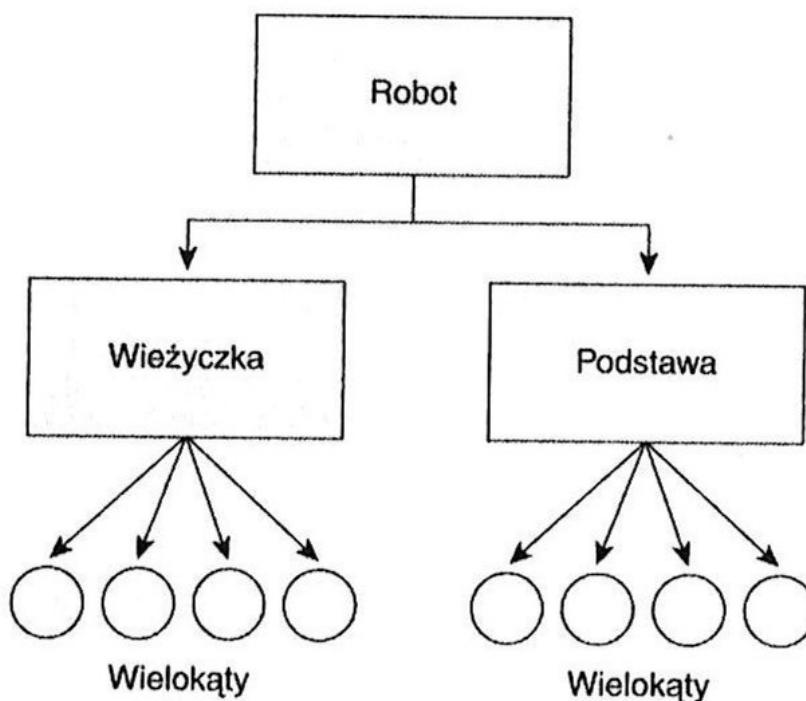


W powyższym przykładzie wieżyczka jest umieszczona centralnie nad grupą wielokątów tworzącą podstawę. Oznacza to, że kiedy wieżyczka obraca się w lewo lub w prawo, obraca się wokół podstawy. Naszym celem jest oczywiście uzyskanie zupełnie innego efektu, ponieważ chcemy, by obracająca się wieżyczka (wraz z anteną) była połączona z podstawą.

Wieżyczka i podstawa poruszają się według własnych schematów, jeszcze inny schemat ruchu możemy zdefiniować dla całego obiektu (robota, czyli podstawy i wieżyczki). Grupy wielokątów są reprezentowane za pomocą hierarchii powiązanych ze sobą i poruszających się obiektów (patrz rysunek 9.7).

Rysunek 9.7.

Trójwymiarowy obiekt jest reprezentowany za pomocą grup wielokątów (prostokąty) i samych wielokątów (kola)



Kiedy rysujesz wielokąt, w pierwszej kolejności powinieneś uwzględnić transformacje dla jego obiektu macierzystego, kolejnego obiektu macierzystego itd. — aż do osiągnięcia transformacji dla korzenia (w tym przypadku całego robota).

Kod implementujący takie grupy wielokątów umieszczony jest w klasie `PolygonGroup` (patrz listing 9.4).

Listing 9.4. *PolygonGroup.java*

```
package com.brackeen.javagamebook.math3D;

import java.util.List;
import java.util.ArrayList;

/**
 * PolygonGroup reprezentuje grupę wielokątów z obiektem klasy
 * MovingTransform3D. Obiekty PolygonGroup mogą także zawierać
 * inne obiekty PolygonGroup.
 */
public class PolygonGroup {

    private String name;
    private List objects;
    private MovingTransform3D transform;

    /**
     * Tworzy nowy pusty obiekt PolygonGroup.
     */
    public PolygonGroup() {
        this("bez_nazwy");
    }

    /**
     * Tworzy nowy pusty obiekt PolygonGroup z określona nazwą.
     */
    public PolygonGroup(String name) {
        setName(name);
        objects = new ArrayList();
        transform = new MovingTransform3D();
    }

    /**
     * Zwraca MovingTransform3D dla tego obiektu PolygonGroup.
     */
    public MovingTransform3D getTransform() {
        return transform;
    }

    /**
     * Zwraca nazwę tego obiektu PolygonGroup.
     */
    public String getName() {
        return name;
    }

    /**
     * Ustawia nazwę tego obiektu PolygonGroup.
     */
    public void setName(String name) {
        this.name = name;
    }
}
```

```
/***
 * Dodaje wielokąt do tej grupy.
 */
public void addPolygon(Polygon3D o) {
    objects.add(o);
}

/***
 * Dodaje obiekt PolygonGroup do tej grupy.
 */
public void addPolygonGroup(PolygonGroup p) {
    objects.add(p);
}

/***
 * Kopiuje tę grupę wielokątów. Obiekty Polygon3D są współdzielone
 * przez tę grupę i kopię grupy; obiekty Transform3D są natomiast kopiowane.
 */
public Object clone() {
    PolygonGroup group = new PolygonGroup(name);
    group.setFilename(filename);
    for (int i=0; i<objects.size(); i++) {
        Object obj = objects.get(i);
        if (obj instanceof Polygon3D) {
            group.addPolygon((Polygon3D)obj);
        }
        else {
            PolygonGroup grp = (PolygonGroup)obj;
            group.addPolygonGroup((PolygonGroup)grp.clone());
        }
    }
    group.transform = (MovingTransform3D)transform.clone();
    return group;
}

/***
 * Zwraca należący do tej grupy obiekt PolygonGroup o podanej
 * nazwie lub - jeśli grupa o takiej nazwie nie istnieje - wartość null.
 */
public PolygonGroup getGroup(String name) {
    // Sprawdza nazwę tej grupy.
    if (this.name != null && this.name.equals(name)) {
        return this;
    }
    for (int i=0; i<objects.size(); i++) {
        Object obj = objects.get(i);
        if (obj instanceof PolygonGroup) {
            PolygonGroup subgroup = ((PolygonGroup)obj).getGroup(name);
            if (subgroup != null) {
                return subgroup;
            }
        }
    }
}
```

```
// Grupa o podanej nazwie nie została znaleziona.  
return null;  
}  
  
/**  
 * Aktualizuje obiekty MovingTransform3D tej grupy i wszystkich  
 * podgrup.  
 */  
public void update(long elapsedTime) {  
    transform.update(elapsedTime);  
    for (int i=0; i<objects.size(); i++) {  
        Object obj = objects.get(i);  
        if (obj instanceof PolygonGroup) {  
            PolygonGroup group = (PolygonGroup)obj;  
            group.update(elapsedTime);  
        }  
    }  
}
```

Klasa `PolygonGroup` utrzymuje listę obiektów będących egzemplarzami albo klasy `Polygon3D`, albo klasy `PolygonGroup`, zatem dowolna grupa wielokątów może zawierać potomne grupy wielokątów.

Obiekty klasy `PolygonGroup` opcjonalnie mogą mieć nazwy. Metoda `getGroup()` umożliwia nam wyszukiwanie obiektu `PolygonGroup` według jego nazwy. Przykładowo moglibyśmy użyć tej metody, gdybyśmy chcieli znaleźć obiekt potomny klasy `PolygonGroup` reprezentujący wieżyczkę (i odpowiednio nazwany) w celu wprowadzenia go w ruch. Metoda `getGroup()` sprawdza nazwę zarówno aktualnej grupy wielokątów, jak i wszystkich potomnych grup wielokątów; metoda zwraca wartość `null`, jeśli żadna z grup nie jest oznaczona podaną nazwą.

Metoda `update()` aktualizuje pozycję obiektu `PolygonGroup` i wszystkich potomnych obiektów `PolygonGroup`.

Metoda `clone()` kopiuje obiekt `PolygonGroup`. Można jej użyć np. do stworzenia wielu różnych robotów w grze. Metoda tworzy nowy obiekt `MovingTransform3D` dla nowego obiektu `PolygonGroup`, jednak pozostawia te same wielokąty (ponieważ i tak pozostają one statyczne).

Iteracyjna obsługa wszystkich wielokątów należących do grupy

Kiedy rysujemy wielokąty reprezentowane przez `PolygonGroup`, nie interesuje nas, który wielokąt należy do której grupy (lub jej potomka); chcemy jedynie narysować wszystkie wielokąty należące do grupy, włącznie z tymi należącymi do podgrup. Chcemy także, by wszystkie wielokąty były rysowane wyłącznie po uwzględnieniu ich ruchu.

Stworzymy teraz kilka metod w klasie PolygonGroup, które będą obsługiwały kolejne (iteracyjne) przechodzenie przez wszystkie wielokąty. Każda klasa reprezentująca grupę wielokątów odpowiada za iteracyjne przechodzenie przez wszystkie swoje wielokąty. Jeśli grupa zawiera jakiekolwiek grupy potomne, każda z tych grup potomnych także odpowiada za zapewnienie iteracyjnej obsługi swoich wielokątów.

```
private int iteratorIndex;

...

/** Zeruje iterator wielokątów dla tej grupy.
 */
public void resetIterator() {
    iteratorIndex = 0;
    for (int i=0; i<objects.size(); i++) {
        Object obj = objects.get(i);
        if (obj instanceof PolygonGroup) {
            ((PolygonGroup)obj).resetIterator();
        }
    }
}

/** Sprawdza, czy istnieje kolejny wielokąt w bieżącej
 * iteracji.
 */
public boolean hasNext() {
    return (iteratorIndex < objects.size());
}

/** Zwraca kolejny wielokąt z bieżącej iteracji, wykonuje operacje
 * ruchu (MovingTransform3D) i zapisuje go w zmiennej cache.
 */
public void nextPolygonTransformed(Polygon3D cache) {
    Object obj = objects.get(iteratorIndex);

    if (obj instanceof PolygonGroup) {
        PolygonGroup group = (PolygonGroup)obj;
        group.nextPolygonTransformed(cache);
        if (!group.hasNext()) {
            iteratorIndex++;
        }
    }
    else {
        iteratorIndex++;
        cache.setTo((Polygon3D)obj);
    }

    cache.add(transform);
}
```

W powyższych metodach każda grupa (obiekt klasy `PolygonGroup`) przechowuje pole `iteratorIndex`, wskazujące na kolejny wielokąt lub grupę potomną, które mają być zwrócone z listy.

Metoda `nextPolygonTransformed()` zwraca wielokąt wskazywany przez aktualną wartość pola `iteratorIndex`. Jeśli na liście na pozycji oznaczonej indeksem `iteratorIndex` znajduje się obiekt klasy `PolygonGroup`, dla tej grupy wywoływana jest jej metoda `nextPolygonTransformed()`.

Metoda `nextPolygonTransformed()` odpowiada także za realizację ruchu grupy. Oznacza to, że grupa potomna wykonuje swój ruch zaraz po tym, jak uczyni to grupa macierzysta. W ten sposób odpowiednie operacje całej hierarchii ruchu są stosowane dla wszystkich wielokątów.

Aby narysować wszystkie wielokąty należące do grupy, dodajemy kolejną metodę `draw()` do klasy `ZBufferedRenderer`:

```
public boolean draw(Graphics2D g, PolygonGroup group) {  
    boolean visible = false;  
    group.resetIterator();  
    while (group.hasNext()) {  
        group.nextPolygonTransformed(temp);  
        visible |= draw(g, temp);  
    }  
    return visible;  
}
```

Osiagnęliśmy już całkiem sporo założonych celów: stworzyliśmy z-bufor, klasę obsługującą ruch obiektów, grupy wielokątów, a także klasę odpowiedzialną za renderowanie grup wielokątów za pomocą z-bufora.

Skupimy się teraz na sposobie odczytywania modelów wielokątów z zewnętrznego pliku.

Wczytywanie grup wielokątów z pliku OBJ

Skoro umiemy już rysować bardziej skomplikowane modele złożone z wielokątów, warto się zastanowić, jak uniknąć pisania zbyt skomplikowanego kodu Javy dla wszystkich wielokątów, które chcemy umieścić w grze. W tym podrozdziale stworzymy parser przetwarzający pliki Alias|Wavefront OBJ, czyli pliki zapisane w popularnym formacie tekstowym, często wykorzystywany właśnie do obiektów trójwymiarowych.

Zastosowanie popularnego formatu, np. *OBJ*, umożliwia twórcom obiektów trójwymiarowych opracowywanie swoich dzieł za pomocą istniejącego oprogramowania i łatwe umieszczanie ich potem w Twojej grze. Ponieważ format *OBJ* opiera się na danych tekstowych, także osoby nie mające dostępu do tego typu specjalistycznego oprogramowania mogą modyfikować istniejące lub tworzyć zupełnie nowe pliki.

W kodzie prezentowanym w tej książce wykorzystujemy tak naprawdę tylko pewien podzbiór specyfikacji plików *OBJ*, jednak nasz kod będzie mógł bez żadnego problemu przetwarzać dowolne istniejące pliki *OBJ*. Jeśli chcesz się zapoznać z kompletną specyfikacją formatu plików *OBJ*, poszukaj odpowiednich materiałów w internecie.

Format pliku *OBJ*

Pliki *OBJ* mają postać tekstu z jedną instrukcją na wiersz. Wiersze rozpoczętymi się od znaku # są komentarzami, natomiast wiersze puste są ignorowane. Nasz parser będzie obsługiwał pięć słów kluczowych:

<code>mtllib <nazwa pliku></code>	Wczytuje materiały z zewnętrznego pliku <i>.mtl</i> .
<code>v <x> <y> <z></code>	Definiuje wierzchołek za pomocą zmiennoprzecinkowych współrzędnych (<i>x</i> , <i>y</i> , <i>z</i>).
<code>f <v1> <v2> <v3> ...</code>	Definiuje nową ścianę. <i>Ścianą</i> nazywamy płaski, wypukły wielokąt z wierzchołkami wypisanymi w porządku odwrotnym do ruchu wskazówek zegara. Ściana może mieć dowolną liczbę wierzchołków. W przypadku każdego wierzchołka liczby dodatnie są indeksami wierzchołków zdefiniowanych w pliku. Liczby ujemne są natomiast względnymi indeksami wierzchołków zdefiniowanych i odczytanych dotychczas, włącznie z ostatnim. Przykładowo 1 oznacza pierwszy wierzchołek w pliku, -1 oznacza ostatni odczytany wierzchołek, a -2 jest indeksem przedostatniego odczytanego wierzchołka.
<code>g <nazwa></code>	Definiuje grupę o podanej nazwie. Kolejno definiowane ściany będą dodawane do tej grupy.
<code>usemtl <nazwa></code>	Używa materiału o podanej nazwie (wczytanego z pliku <i>.mtl</i>) dla ścian definiowanych od tego miejsca.

Format *MTL* definiuje materiały lub (jak w tym przypadku) tekstury. Plikom *MTL* poświęcimy trochę czasu później — najpierw stworzymy przykładowy plik *OBJ* reprezentujący kształt sześcianu (patrz listing 9.5).

Listing 9.5. *cube.obj*

```
# wczytuje materiały
mtllib textures.mtl

# definiuje wierzchołki
v 16 32 16
v 16 32 -16
v 16 0 16
v 16 0 -16
v -16 32 16
v -16 32 -16
v -16 0 16
```

```
v -16 0 -16  
# nazywa nową grupę  
g myCube  
  
# definiuje materiał  
usemtl texture_A  
  
# definiuje wielokąty  
f 1 3 4 2  
f 6 8 7 5  
f 2 6 5 1  
f 3 7 8 4  
f 1 5 7 3  
f 4 8 6 2
```

Ten przykładowy plik *OBJ* zawiera definicje ośmiu wierzchołków i sześciu wielokątów składających się na sześcian. Podobnych obiektów użyjemy w demonstracyjnej grze na końcu tego rozdziału.

Przejdzmy teraz do tworzenia parsera plików *OBJ*. Pierwszą część parsera, klasę *ObjectLoader*, przedstawiamy na listingu 9.6. Nie jest to właściwy parser plików *OBJ*, a jedynie kod przygotowujący system do zarządzania materiałami, wierzchołkami i grupami wielokątów. Klasa *ObjectLoader* odpowiada także za podstawowe przetwarzanie pliku, jak ignorowanie wierszy komentarzy i wierszy pustych; do osobnej klasy właściwego parsera przekazuje pozostałe wiersze.

Listing 9.6. *ObjectLoader.java*

```
package com.brackeen.javagamebook.math3D;  
  
import java.io.*;  
import java.util.*;  
import com.brackeen.javagamebook.graphics3D.texture.*;  
  
/**  
 * Klasa ObjectLoader wczytuje podzbior opracowanej przez  
 * Alias|Wavefront specyfikacji plików OBJ.  
 */  
public class ObjectLoader {  
  
    /**  
     * Klasa Material owija ShadedTexture.  
     */  
    public static class Material {  
        public File sourceFile;  
        public ShadedTexture texture;  
    }  
  
    /**  
     * LineParser jest interfejsem parsera wiersza z pliku tekstowego.  
     * Do przetwarzania plików OBJ i MTL będziemy używać osobnych  
     * parserów LineParser.  
     */
```

```
/*
protected interface LineParser {
    public void parseLine(String line) throws IOException, NumberFormatException,
NoSuchElementException;
}

protected File path;
protected List vertices;
protected Material currentMaterial;
protected HashMap materials;
protected List lights;
protected float ambientLightIntensity;
protected HashMap parsers;
private PolygonGroup object;
private PolygonGroup currentGroup;

/**
 * Tworzy nowy ObjectLoader.
 */
public ObjectLoader() {
    materials = new HashMap();
    vertices = new ArrayList();
    parsers = new HashMap();
    parsers.put("obj", new ObjLineParser());
    parsers.put("mtl", new MtlLineParser());
    currentMaterial = null;
    setLights(new ArrayList(), 1);
}

/**
 * Ustawia światła dla wielokątów tworzących przetwarzane obiekty.
 * Po wywołaniu tej metody wywołania metody loadObject używają
 * tych światel.
 */
public void setLights(List lights,
                      float ambientLightIntensity)
{
    this.lights = lights;
    this.ambientLightIntensity = ambientLightIntensity;
}

/**
 * Wczytuje plik OBJ jako PolygonGroup.
 */
public PolygonGroup loadObject(String filename)
    throws IOException
{
    File file = new File(filename);
    object = new PolygonGroup();
    object.setFilename(file.getName());
    path = file.getParentFile();

    vertices.clear();
    currentGroup = object;
    parseFile(filename);

    return object;
}
```

```
/*
 * Zwraca Vector3D z listy wektorów w pliku. Indeksy ujemne
 * są liczone od końca listy; indeksy dodatnie wskazują natomiast
 * na wektory od początku listy. 1 jest pierwszym indeksem,
 * -1 - ostatnim, natomiast 0 jest indeksem nieprawidłowym,
 * który powoduje wygenerowanie wyjątku.
 */
protected Vector3D getVector(String indexStr) {
    int index = Integer.parseInt(indexStr);
    if (index < 0) {
        index = vertices.size() + index + 1;
    }
    return (Vector3D)vertices.get(index-1);
}

/*
 * Parsuje plik OBJ (z rozszerzeniem ".obj") lub plik MTL
 * (z rozszerzeniem ".mtl").
 */
protected void parseFile(String filename) throws IOException
{
    // Otwórz plik o podanej ścieżce.
    File file = new File(path, filename);
    BufferedReader reader = new BufferedReader(new FileReader(file));

    // Uruchom parser zgodnie z rozszerzeniem pliku.
    LineParser parser = null;
    int extIndex = filename.lastIndexOf('.');
    if (extIndex != -1) {
        String ext = filename.substring(extIndex+1);
        parser = (LineParser)parsers.get(ext.toLowerCase());
    }
    if (parser == null) {
        parser = (LineParser)parsers.get("obj");
    }

    // Parsuj wszystkie wiersze w pliku.
    while (true) {
        String line = reader.readLine();
        // Brak kolejnych wierszy do wczytania.
        if (line == null) {
            reader.close();
            return;
        }

        line = line.trim();

        // Ignoruj puste wiersze i komentarze.
        if (line.length() > 0 && !line.startsWith("#")) {
            // Interpretuj wiersz.
            try {
                parser.parseLine(line);
            }
            catch (NumberFormatException ex) {
                throw new IOException(ex.getMessage());
            }
        }
    }
}
```

```
        catch (NoSuchElementException ex) {
            throw new IOException(ex.getMessage());
        }
    }
}
```

Klasa ObjectLoader zawiera wewnętrzny interfejs, nazwany LineParser. Interfejs udostępnia metodę do parsowania pojedynczego wiersza w pliku wejściowym i zawiera dwie implementujące klasy: ObjLineParser i Mt1LineParser.

Metoda `parseFile()` parsuje kolejno wiersze (pojedynczo) albo pliku *OBJ*, albo pliku *MTL*, ignorując wiersze puste i komentarze. Metoda wykorzystuje właściwą klasę implementującą interfejs `LineParser` do parsowania poszczególnych wierszy — wybór klasy zależy od rozszerzenia przetwarzanego pliku (*obj* lub *.mtl*).

Metoda `getVector()` zapewnia wygodny sposób uzyskiwania wierzchołka o podanym indeksie. Pamiętaj, że indeksy rozpoczynają się od 1, a indeksy ujemne są liczone w odwrotnym kierunku (od ostatniego wierzchołka).

Metoda `loadObject()` została zaprojektowana z myślą o wczytywaniu pliku *OBJ* i zwarcianiu obiektu klasy `PolygonGroup`.

Żadna z opisanych powyżej metod nie działałaby oczywiście bez klasy ObjLineParser (patrz listing 9.7).

Listing 9.7. *ObiLineParser*. wewnętrzna klasa klasy *ObjectLoader*

```
/***
 * Parsuje wiersz z pliku OBJ.
 */
protected class ObjLineParser implements LineParser {
    public void parseLine(String line) throws IOException,
        NumberFormatException, NoSuchElementException
    {
        StringTokenizer tokenizer = new StringTokenizer(line);
        String command = tokenizer.nextToken();
        if (command.equals("v")) {
            // Tworzy nowy wierzchołek.
            vertices.add(new Vector3D(Float.parseFloat(tokenizer.nextToken()),
                Float.parseFloat(tokenizer.nextToken()), Float.parseFloat(tokenizer.
                    nextToken())));
        }
        else if (command.equals("f")) {
            // Tworzy nową ścianę (plaski, wypukły wielokąt).
            List currVertices = new ArrayList();
            while (tokenizer.hasMoreTokens()) {
                String indexStr = tokenizer.nextToken();
                currVertices.add(indexStr);
            }
            faces.add(new Face(currVertices));
        }
    }
}
```

```
// Ignoruje tekstury i dodatkowe współrzędne.  
int endIndex = indexStr.indexOf('/');  
if (endIndex != -1) {  
    indexStr = indexStr.substring(0, endIndex);  
}  
  
currVertices.add(getVector(indexStr));  
}  
  
// Tworzy wielokąt pokryty tekstem.  
Vector3D[] array = new Vector3D[currVertices.size()];  
currVertices.toArray(array);  
TexturedPolygon3D poly = new TexturedPolygon3D(array);  
  
// Ustawia teksturę.  
ShadedSurface.createShadedSurface(poly, currentMaterial.texture, lights,  
ambientLightIntensity);  
  
// Dodaje wielokąt do bieżącej grupy.  
currentGroup.addPolygon(poly);  
}  
else if (command.equals("g")) {  
    // Definiuje bieżącą grupę.  
    if (tokenizer.hasMoreTokens()) {  
        String name = tokenizer.nextToken();  
        currentGroup = new PolygonGroup(name);  
    }  
    else {  
        currentGroup = new PolygonGroup();  
    }  
    object.addPolygonGroup(currentGroup);  
}  
else if (command.equals("mtllib")) {  
    // Wczytuje materiały z pliku.  
    String name = tokenizer.nextToken();  
    parseFile(name);  
}  
else if (command.equals("usemtl")) {  
    // Definiuje bieżący materiał.  
    String name = tokenizer.nextToken();  
    currentMaterial = (Material)materials.get(name);  
    if (currentMaterial == null) {  
        System.out.println("brak materiału: " + name);  
    }  
}  
else {  
    // Nieznane polecenie - ignoruję je.  
}
```

Klasa ObjLineParser wykorzystuje klasę StringTokenizer do otrzymania z parsowanego wiersza słów oddzielonych spacjami. Parsowanie pojedynczego wiersza jest stosunkowo proste. Instrukcja v tworzy nowy wierzchołek, instrukcja f tworzy nowy wielokąt.

(wykorzystujemy klasę ShadedSurface), instrukcja `g` tworzy nową grupę wielokątów (`PolygonGroup`), a instrukcja `usemtl` ustawia bieżący materiał. Znalezienie w wierszu instrukcji `mtllib` powoduje wywołanie metody `parseFile()`, która wczytuje plik *MTL*.

Warto zaznaczyć, że nasz parser w przypadku instrukcji `f` dla każdego wierzchołka ignoruje wszystkie ukośniki (/) oraz znaki następujące po nich. Wynika to z faktu, że specyfikacja plików *OBJ* używa dodatkowych wierzchołków, opisywanych po ukośniku, do definiowania współrzędnych tekstury. Nie używamy tych dodatkowych wierzchołków w naszym kodzie, zatem ignorujemy symbol ukośnika i następujące po nim znaki. Dzięki temu możemy odczytywać zarówno te pliki *OBJ*, które zawierają tego typu definicje, jak i te pozabawione takich zapisów.

Nasza klasa odczytująca i przetwarzająca plik *OBJ* wymaga jeszcze jednego elementu parsera pliku *MTL*.

Format pliku MTL

Format pliku *MTL* został zaprojektowany z myślą o definiowaniu materiałów — jednolitych kolorów, powierzchni odbijających (np. dla lśniących obiektów lub luster), odwzorowań nierówności (symulujących głębię materiału) oraz tekstur. Specyfikacja formatu definiuje rozmaite rodzaje opcji i instrukcji, jednak w tym przypadku skupimy się wyłącznie na mapowaniu tekstur. Oto instrukcje, które będą nas interesowały w przetwarzanym pliku *MTL*:

<code>newmtl <nazwa></code>	Definiuje z nazwy nowy materiał.
<code>map_Kd <nazwa pliku></code>	Dodaje do materiału mapę tekstury.

Stworzona przez nas klasa `MtlLineParser` wczyta tekstury i stworzy nowe obiekty klasy `Material` (patrz listing 9.8).

Listing 9.8. *MtlLineParser*, wewnętrzna klasa w klasie *ObjectLoader*

```

/**
 * Parsuje pojedynczy wiersz pliku materiałów MTL.
 */
protected class MtlLineParser implements LineParser {

    public void parseLine(String line) throws NoSuchElementException
    {
        StringTokenizer tokenizer = new StringTokenizer(line);
        String command = tokenizer.nextToken();

        if (command.equals("newmtl")) {
            // W razie potrzeby tworzy nowy materiał.
            String name = tokenizer.nextToken();
            currentMaterial = (Material)materials.get(name);
            if (currentMaterial == null) {
                currentMaterial = new Material();
                materials.put(name, currentMaterial);
            }
        }
    }
}

```

```
        }
        else if (command.equals("map_Kd")) {
            // Nadaje bieżącemu materialowi teksturę
            String name = tokenizer.nextToken();
            File file = new File(path, name);
            if (!file.equals(currentMaterial.sourceFile)) {
                currentMaterial.sourceFile = file;
                currentMaterial.texture = (ShadedTexture) Texture.createTexture
                    (file.getPath(), true);
            }
        }
        else {
            // Nieznana instrukcja - zignoruj ją.
        }
    }
}
```

To wszystko, co jest nam potrzebne do wczytywania plików *OBJ*. Stworzyliśmy klasę wczytującą plik *OBJ*, która przetwarza tylko wybrany podzbior instrukcji zdefiniowanych dla formatu *OBJ*, uwzględniliśmy jednak wszystkie elementy potrzebne do wyświetlania trójwymiarowych obiektów w tej książce. Jeśli chcesz zmodyfikować klasę wczytującą pliki *OBJ* w taki sposób, by obsługiwała rozszerzony zbiór instrukcji, poszukaj w internecie specyfikacji tego formatu i skorzystaj z dostępnych w sieci obszernych materiałów poświęconych plikom tego typu.

Obiekty w grze

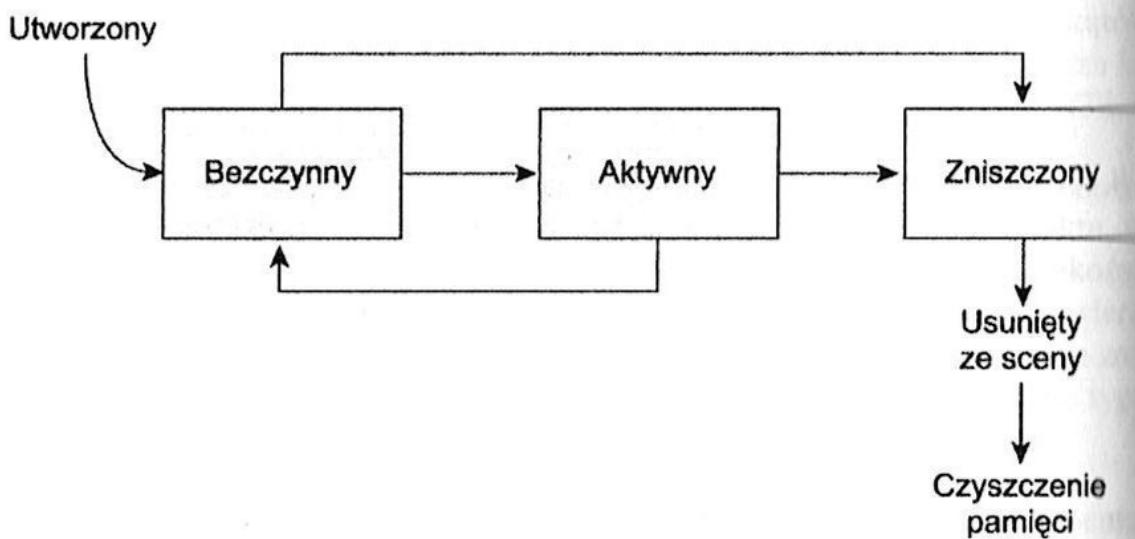
W tym podrozdziale stworzymy kontener dla obiektów klasy *PolygonGroup*, nazwany *GameObject*, który będzie klasą bazową dla wszystkich obiektów w grze i w programach demonstracyjnych prezentowanych w tej książce. Chodzi przede wszystkim o to, by obiekt w grze mógł sam wpływać na swój stan, włącznie z przemieszczaniem grupy wielokątów. W rzeczywistości nasza wersja klasy *GameObject* nie wykonuje żadnych specjalnych działań — zaprogramowanie ewentualnych interesujących ruchów wymaga rozszerzenia kodu tej klasy.

Stworzona w tym rozdziale klasa *GameObject* jest dopiero początkiem naszej drogi do stworzenia zaawansowanego obiektu w grze; w kolejnych rozdziałach będziemy ją rozbudowywać w miarę omawiania takich zagadnień, jak wykrywanie kolizji, znajdowanie ścieżki i sztuczną inteligencję.

Po pierwsze, zajmijmy się podstawowymi stanami dotyczącymi wszystkich obiektów występujących w grach. *Obiektem w grze* może być cokolwiek — obiekt statyczny, pocisk, włącznik lub przeciwnik. Podczas gry obiekt może się znajdować w stanie bezczynności (w bezruchu) lub aktywności (np. krążyć wokół wybranego punktu). Dodatkowo niektóre obiekty mogą być niszczone — nasz przeciwnik może ginąć w wybuchu, a pocisk może np. trafić w ścianę. Mając to na uwadze, opisujemy obiekty w grze za pomocą trzech możliwych stanów: bezczynności (IDLE), aktywności (ACTIVE) i zniszczenia (DESTROYED) — patrz rysunek 9.8.

Rysunek 9.8.

Cykl „życia” obiektu w grze



Kiedy obiekt w grze jest po raz pierwszy tworzony, znajduje się w stanie bezczynności i w dowolnym momencie może przejść w stan aktywności. Podobnie obiekt znajdujący się w stanie aktywności może w dowolnym momencie przejść w stan bezczynności.

Zarówno ze stanu bezczynności, jak i ze stanu aktywności obiekt w grze może przejść do stanu zniszczenia. Obiekt znajdujący się w stanie zniszczenia nie może wrócić do żadnego innego stanu. Menedżer obiektów w grze powinien taki obiekt usunąć ze świata gry — niepotrzebne (w tym zniszczone) obiekty występujące w grze powinny podlegać mechanizmowi usuwania śmieci (przy założeniu, że do likwidowanego obiektu nie istnieją żadne inne odwołania).

Różne typy obiektów w grze realizują przedstawiony cykl „życia” w różny sposób. Przykładowo obiekt statyczny może pozostawać w stanie bezczynności podczas całej gry. Pocisk może w jednej chwili przejść do stanu aktywności i pozostawać w nim do momentu uderzenia w cel lub jakąś przeszkodę, kiedy równie szybko przejdzie w stan zniszczenia. Przeciwnik może przez długi czas pozostawać w stanie bezczynności i przejść do stanu aktywności w momencie, gdy znajdzie się w polu widzenia gracza. Przykładowo jeśli przeciwnik na chwilę znika z ekranu, może wrócić do stanu bezczynności; na końcu przeciwnik przechodzi w stan zniszczenia, kiedy zostanie trafiony przez gracza wystarczającą liczbę razy.

Mając na uwadze wszystkie te założenia, stworzymy teraz klasę `GameObject` (patrz listing 9.9). Klasa zarządza stanem i udostępnia wiele wygodnych metod obsługujących grupę wielokątów i ruch obiektu w grze.

Listing 9.9. `GameObject.java`

```

package com.brackeen.javagamebook.game;

import com.brackeen.javagamebook.math3D.*;

/**
 * Klasa GameObject jest podstawą wszystkich typów obiektów
 * występujących w grze, które są reprezentowane za pomocą obiektu
 * klasy PolygonGroup. Przykładowo GameObject może być obiektem
 * statycznym (np. skrzynią), obiektem w ruchu (np. pociskiem lub
 */
  
```

przeciwniikiem) lub dowolnym innym typem obiektu (np. apteczką). Obiekty klasy GameObject mogą się znajdować w trzech podstawowych stanach: bezczynności (STATE_IDLE), aktywności (STATE_ACTIVE) i zniszczenia (STATE_DESTROYED).

```
/*
public class GameObject {
    protected static final int STATE_IDLE = 0;
    protected static final int STATE_ACTIVE = 1;
    protected static final int STATE_DESTROYED = 2;

    private PolygonGroup polygonGroup;
    private int state;

    /**
     * Tworzy nowy GameObject, reprezentowany przez przekazany obiekt klasy PolygonGroup. Przekazany obiekt może mieć wartość null.
     */
    public GameObject(PolygonGroup polygonGroup) {
        this.polygonGroup = polygonGroup;
        state = STATE_IDLE;
    }

    /**
     * Zwraca położenie tego obiektu GameObject na podstawie danych obiektu klasy Transform3D.
     */
    public Vector3D getLocation() {
        return polygonGroup.getTransform().getLocation();
    }

    /**
     * Zwraca obiekt reprezentujący ruch obiektu w grze.
     */
    public MovingTransform3D getTransform() {
        return polygonGroup.getTransform();
    }

    /**
     * Zwraca obiekt klasy PolygonGroup dla tego obiektu w grze.
     */
    public PolygonGroup getPolygonGroup() {
        return polygonGroup;
    }

    /**
     * Ustawia stan tego obiektu. Parametr state powinien mieć wartość STATE_IDLE, STATE_ACTIVE lub STATE_DESTROYED.
     */
    protected void setState(int state) {
        this.state = state;
    }
}
```

```
/***
 * Ustawia stan wskazanego obiektu. Dzięki temu dowolny obiekt
 * GameObject może zmienić stan dowolnego innego obiektu GameObject.
 * Parametr state powinien mieć wartość STATE_IDLE, STATE_ACTIVE lub
 * STATE_DESTROYED.
 */
protected void setState(GameObject object, int state) {
    object.setState(state);
}

/***
 * Zwraca true, jeśli ten obiekt GameObject jest w stanie bezczynności.
 */
public boolean isIdle() {
    return (state == STATE_IDLE);
}

/***
 * Zwraca true, jeśli ten obiekt GameObject jest w stanie aktywności.
 */
public boolean isActive() {
    return (state == STATE_ACTIVE);
}

/***
 * Zwraca true, jeśli ten obiekt GameObject jest w stanie zniszczenia.
 */
public boolean isDestroyed() {
    return (state == STATE_DESTROYED);
}

/***
 * Jeśli ten GameObject znajduje się w stanie aktywności, ta
 * metoda aktualizuje jego obiekt PolygonGroup. W przeciwnym
 * przypadku metoda nie wykonuje żadnych działań.
 */
public void update(GameObject player, long elapsedTime) {
    if (isActive()) {
        polygonGroup.update(elapsedTime);
    }
}

/***
 * Informuje ten GameObject podczas ostatniej aktualizacji o tym,
 * czy był widoczny czy nie. Domyślnie, jeśli ten GameObject
 * znajduje się w stanie bezczynności i otrzymuje informację,
 * że jest widoczny, przechodzi w stan aktywności.
 */
public void notifyVisible(boolean visible) {
    if (visible && isIdle()) {
        state = STATE_ACTIVE;
    }
}
```

W tej domyślnej klasie reprezentującej obiekt w grze grupa wielokątów jest aktualizowana tylko wtedy, gdy obiekt znajduje się w stanie aktywności. Odwołanie do obiektu gracza jest parametrem metody aktualizującej `update()`, co oznacza, że dowolny obiekt w grze może uzyskać dane na temat położenia gracza. Taka wiedza jest oczywiście bardzo potrzebna występującym w grze przeciwnikom gracza.

Zwróć także uwagę na metodę `notifyVisible()`. Została ona zaprojektowana po to, by w każdej klatce informować obiekt, czy jest widoczny. Jeśli obiekt jest widoczny i znajduje się w stanie bezczynności, automatycznie przejdzie w stan aktywności.

Powyższa klasa nie wymusza ścisłego przestrzegania reguł omawianego cyklu „życia” obiektu w grze — teoretycznie mamy możliwość wyprowadzenia obiektu ze stanu zniszczenia w stan aktywności, byłoby to jednak bezcelowe w przypadku, gdyby dany obiekt został trwale usunięty z wirtualnego świata naszej gry komputerowej.

Zarządzanie obiektami w grze

Aby śledzić zachowanie wszystkich obiektów w grze, stworzymy interfejs `GameObjectManager` (patrz listing 9.10). W tym rozdziale zaprezentujemy dosyć prostą implementację tego interfejsu, jednak w kolejnych rozdziałach spróbujemy stworzyć całkiem nową implementację.

Listing 9.10. `GameObjectManager.java`

```
package com.brackeen.javagamebook.game;

import java.awt.Rectangle;
import java.awt.Graphics2D;
import com.brackeen.javagamebook.game.GameObjectRenderer;

/**
 * Interfejs GameObjectManager zawiera metody umożliwiające
 * śledzenie i rysowanie obiektów GameObject.
 */
public interface GameObjectManager {

    /**
     * Oznacza wszystkie obiekty znajdujące się w wyznaczonym
     * dwuwymiarowym obszarze jako potencjalnie widoczne (czyli
     * takie, które należy narysować).
     */
    public void markVisible(Rectangle bounds);

    /**
     * Oznacza wszystkie obiekty jako potencjalnie widoczne (czyli
     * takie, które należy narysować).
     */
    public void markAllVisible();
```

```

    /**
     * Dodaje obiekt klasy GameObject do tego menadżera.
    */
    public void add(GameObject object);

    /**
     * Dodaje obiekt klasy GameObject do tego menedżera i określa
     * go jako obiekt gracza. Istniejący obiekt gracza, jeśli istnieje,
     * nie jest usuwany.
    */
    public void addPlayer(GameObject player);

    /**
     * Zwraca obiekt określony jako obiekt gracza lub wartość null,
     * jeśli nie wyznaczono żadnego obiektu gracza.
    */
    public GameObject getPlayer();

    /**
     * Usuwa obiekt klasy GameObject z tego menedżera.
    */
    public void remove(GameObject object);

    /**
     * Aktualizuje wszystkie obiekty w oparciu o czas, jaki minął
     * od ostatniej aktualizacji.
    */
    public void update(long elapsedTime);

    /**
     * Rysuje wszystkie widoczne obiekty.
    */
    public void draw(Graphics2D g, GameObjectRenderer r);
}

```

Interfejs `GameObjectManager` udostępnia metody umożliwiające śledzenie zachowania wszystkich obiektów występujących w świecie gry. Zawiera także dwie metody, `markVisible()` i `markAllVisible()`, które oznaczają te obiekty, które są widoczne i powinny być narysowane. Metoda `markVisible()` pobiera jako parametr obiekt klasy `Rectangle`, który określa dwuwymiarowy widoczny obszar (widziany z lotu ptaka). Tylko obiekty oznaczone jako widoczne będą rysowane; zaraz po ich narysowaniu wszystkie obiekty są ponownie oznaczane jako niewidoczne.

Metoda `draw()` interfejsu `GameObjectManager` rysuje wszystkie widoczne obiekty zarządzane przez tego menedżera obiektów w grze i oznacza wszystkie obiekty jako niewidoczne. Do rysowania obiektów metoda wykorzystuje `GameObjectRenderer`, czyli interfejs stworzony właśnie z myślą o rysowaniu obiektów (patrz listing 9.11). Implementacją tego interfejsu jest klasa `ZBufferedRenderer`.

Listing 9.11. *GameObjectRenderer.java*

```
package com.brackeen.javagamebook.game;

import java.awt.Graphics2D;
import com.brackeen.javagamebook.game.GameObject;

/**
 * Interfejs GameObjectRenderer udostępnia metody rysowania
 * obiektu GameObject.
 */
public interface GameObjectRenderer {

    /**
     * Rysuje obiekt i zwraca wartość true, jeśli którakolwiek
     * część obiektu jest widoczna.
     */
    public boolean draw(Graphics2D g, GameObject object);

}
```

Klasa SimpleGameObjectManager implementuje interfejs GameObjectManager, utrzymując listę wszystkich obiektów występujących w grze. Kiedy obiekt jest oznaczany jako widoczny, jest umieszczany na liście obiektów „widocznych”, która jest czyszczona zaraz po narysowaniu wszystkich widocznych obiektów. Bardziej zaawansowaną implementację interfejsu GameObjectManager stworzymy później, w rozdziale 11. Wersja SimpleGameObjectManager na razie wystarczy do naszych celów.

Łączenie elementów

Stworzyliśmy już w tym rozdziale całkiem sporo kodu źródłowego. Zbliżamy się do końca, zatem powinniśmy stworzyć program demonstracyjny, wykorzystujący wszystkie opracowane przez nas składniki.

Program demonstracyjny będzie umożliwiał swobodne poruszanie się po obszarze, na którym będą występować trzy różne obiekty: statyczne skrzynie, pociski i roboty. Każdy z tych obiektów ma własny plik *OBJ*, który definiuje jego trójwymiarowy model.

Statyczne skrzynie są prostymi obiektami klasy GameObject i nie wykonują żadnych działań — są jedynie urozmaiceniem przedstawionego świata.

Robota (obiekt klasy Bot) definiujemy za pomocą kodu z listingu 9.12. Obiekty tego typu wyglądają niemal identycznie, jak przykładowa wieżyczka i podstawa zaprezentowana na rysunku 9.6. Wieżyczka porusza się niezależnie od podstawy i bez przerwy obraca się w taki sposób, by ustawić się przodem do gracza.

Listing 9.12. Bot.java

```
import com.brackeen.javagamebook.math3D.*;
import com.brackeen.javagamebook.game.GameObject;

/**
 * Bot jest obiektem w grze, statycznym robotem z wieżyczką
 * obracającą się w stronę gracza.
 */
public class Bot extends GameObject {

    private static final float TURN_SPEED = .0005f;
    private static final long DECISION_TIME = 2000;

    protected MovingTransform3D mainTransform;
    protected MovingTransform3D turretTransform;
    protected long timeUntilDecision;
    protected Vector3D lastPlayerLocation;

    public Bot(PolygonGroup polygonGroup) {
        super(polygonGroup);
        mainTransform = polygonGroup.getTransform();
        PolygonGroup turret = polygonGroup.getGroup("turret");
        if (turret != null) {
            turretTransform = turret.getTransform();
        } else {
            System.out.println("Nie zdefiniowano wieżyczki!");
        }
        lastPlayerLocation = new Vector3D();
    }

    public void notifyVisible(boolean visible) {
        if (!isDestroyed()) {
            if (visible) {
                setState(STATE_ACTIVE);
            } else {
                setState(STATE_IDLE);
            }
        }
    }

    public void update(GameObject player, long elapsedTime) {
        if (turretTransform == null || isIdle()) {
            return;
        }

        Vector3D playerLocation = player.getLocation();
        if (playerLocation.equals(lastPlayerLocation)) {
            timeUntilDecision = DECISION_TIME;
        } else {
            timeUntilDecision -= elapsedTime;
            if (timeUntilDecision <= 0 ||
                !turretTransform.isTurningY())
        }
    }
}
```

```

    {
        float x = player.getX() - getX();
        float z = player.getZ() - getZ();
        turretTransform.turnYTo(x, z,
            -mainTransform.getAngleY(), TURN_SPEED);
        lastPlayerLocation.setTo(playerLocation);
        timeUntilDecision = DECISION_TIME;
    }
}
super.update(player, elapsedTime);
}
}

```

W obiekcie klasy Bot nie sprawdzamy położenia gracza w każdej klatce; zamiast tego sprawdzamy to położenie co dwie sekundy (częstotliwość sprawdzania pozycji gracza definiujemy w polu DECISION_TIME). W ten sposób robot oblicza kierunek, w którym powinien się zwrócić co kilka sekund, co oznacza, że jego ruchy są opóźnione wobec zachowań gracza.

Kolejnym obiektem w grze jest Blast (patrz listing 9.13), czyli pocisk wystrzeliwany przez gracza. Obiekty tego typu poruszają się w linii prostej (wirując) i ulegają zniszczeniu po upływie pięciu sekund. Pola SPEED i ROT_SPEED definiują szybkość ruchu i wirowania pocisku.

Listing 9.13. Blast.java

```

import com.brackeen.javagamebook.math3D.*;
import com.brackeen.javagamebook.game.GameObject;

/**
 * Obiekt w grze (GameObject) Blast reprezentuje pocisk.
 * Został zaprojektowany w taki sposób, by poruszał się
 * w linii prostej przez pięć sekund i ulegał zniszczeniu.
 * Obiekty Blast niszczą jednym trafieniem roboty (obiekty
 * klasy Bot).
 */
public class Blast extends GameObject {

    private static final long DIE_TIME = 5000;
    private static final float SPEED = 1.5f;
    private static final float ROT_SPEED = .008f;

    private long aliveTime;

    /**
     * Tworzy nowy obiekt Blast z wykorzystaniem wskazanej grupy
     * PolygonGroup i znormalizowanego wektora kierunku.
     */
    public Blast(PolygonGroup polygonGroup, Vector3D direction) {
        super(polygonGroup);
        MovingTransform3D transform = polygonGroup.getTransform();
        Vector3D velocity = transform.getVelocity();
        velocity.setTo(direction);
        velocity.multiply(SPEED);
        transform.setVelocity(velocity);
    }
}

```

```

        //transform.setAngleVelocityX(ROT_SPEED);
        transform.setAngleVelocityY(ROT_SPEED);
        transform.setAngleVelocityZ(ROT_SPEED);
        setState(STATE_ACTIVE);
    }

    public void update(GameObject player, long elapsedTime) {
        aliveTime+=elapsedTime;
        if (aliveTime >= DIE_TIME) {
            setState(STATE_DESTROYED);
        }
        else {
            super.update(player, elapsedTime);
        }
    }
}

```

Skoro mamy już wszystkie obiekty występujące w grze, możemy stworzyć klasę GameObjectTest. Jak zwykle klasa ta rozszerza klasę GameCore3D i przesyła część jej metod. W pierwszej kolejności skupimy się na metodzie createPolygons(), która wczytuje pliki *OBJ* i na ich podstawie konstruuje obiekty w grze:

```

public void createPolygons() {

    // Tworzy ścianę.
    Texture floorTexture = Texture.createTexture("../images/roof1.png", true);
    ((ShadedTexture)floorTexture).setDefaultShadeLevel(ShadedTexture.MAX_LEVEL*3/4);
    Rectangle3D floorTextureBounds = new Rectangle3D(new Vector3D(0,0,0),
        new Vector3D(1,0,0), new Vector3D(0,0,1), floorTexture.getWidth(),
        floorTexture.getHeight());
    float s = GAME_AREA_SIZE;
    floor = new TexturedPolygon3D(new Vector3D[] {new Vector3D(-s, 0, s),
        new Vector3D(s, 0, s), new Vector3D(s, 0, -s), new Vector3D(-s, 0, -s)});
    floor.setTexture(floorTexture, floorTextureBounds);

    // Ustawia lokalne światła dla modelu.
    float ambientLightIntensity = .5f;
    List lights = new LinkedList();
    lights.add(new PointLight3D(-100,100,100, .5f, -1));
    lights.add(new PointLight3D(100,100,0, .5f, -1));

    // Wczytuje modele obiektów
    ObjectLoader loader = new ObjectLoader();
    loader.setLights(lights, ambientLightIntensity);
    try {
        robotModel = loader.loadObject("../images/robot.obj");
        powerUpModel = loader.loadObject("../images/cube.obj");
        blastModel = loader.loadObject("../images/blast.obj");
    }
    catch (IOException ex) {
        ex.printStackTrace();
    }
}

```

```
// Tworzy obiekty w grze
gameObjectManager = new SimpleGameObjectManager();
gameObjectManager.addPlayer(new GameObject(new PolygonGroup("Player")));
gameObjectManager.getPlayer().getLocation().y = 5;
for (int i=0; i<NUM_BOTS; i++) {
    Bot object = new Bot((PolygonGroup)robotModel.clone());
    placeObject(object);
}
for (int i=0; i<NUM_POWER_UPS; i++) {
    GameObject object = new GameObject((PolygonGroup)powerUpModel.clone());
    placeObject(object);
}
```

Kolejną interesującą metodą klasy `GameObjectTest` jest metoda `updateWorld()`, która została zmodyfikowana w porównaniu z oryginalną wersją z klasy `GameCore3D`, dzięki czemu obsługuje także ruch gracza. Metoda umożliwia także wystrzeliwanie pocisków przez gracza:

```
public void updateWorld(long elapsedTime) {

    float angleVelocity;

    // Ogranicza elapsedTime.
    elapsedTime = Math.min(elapsedTime, 100);

    GameObject player = gameObjectManager.getPlayer();
    MovingTransform3D playerTransform = player.getTransform();
    Vector3D velocity = playerTransform.getVelocity();

    playerTransform.stop();
    float x = -playerTransform.getSinAngleY();
    float z = -playerTransform.getCosAngleY();
    if (goForward.isPressed()) {
        velocity.add(x, 0, z);
    }
    if (goBackward.isPressed()) {
        velocity.add(-x, 0, -z);
    }
    if (goLeft.isPressed()) {
        velocity.add(z, 0, -x);
    }
    if (goRight.isPressed()) {
        velocity.add(-z, 0, x);
    }
    if (fire.isPressed()) {
        float cosX = playerTransform.getCosAngleX();
        float sinX = playerTransform.getSinAngleX();
        Blast blast = new Blast((PolygonGroup)blastModel.clone(),
            new Vector3D(cosX*x, sinX, cosX*z));
        // Początkowa pozycja pocisku. Można odnieść wrażenie.
        // że w momencie oddawania strzału pocisk wychodzi
        // z Twojego czoła.
        blast.getLocation().setTo(player.getX(), player.getY() + BULLET_HEIGHT,
            player.getZ());
        gameObjectManager.add(blast);
    }
}
```

```

velocity.multiply(PLAYER_SPEED);
playerTransform.setVelocity(velocity);

// Patrzenie w górę lub w dół (obrót wokół osi x).
angleVelocity = Math.min(tiltUp.getAmount(), 200);
angleVelocity += Math.max(-tiltDown.getAmount(), -200);
playerTransform.setAngleVelocityX(angleVelocity * PLAYER_TURN_SPEED / 200);

// Obracanie się (obrót wokół osi y).
angleVelocity = Math.min(turnLeft.getAmount(), 200);
angleVelocity += Math.max(-turnRight.getAmount(), -200);
playerTransform.setAngleVelocityY(angleVelocity * PLAYER_TURN_SPEED / 200);

// Na razie oznaczamy wszystkie obiekty świata gry jako widoczne w tej klatce,
gameObjectManager.markAllVisible();

// Aktualizuje obiekty.
gameObjectManager.update(elapsedTime);

// Ogranicza kąt widzenia w górę i w dół.
float angleX = playerTransform.getAngleX();
float limit = (float)Math.PI / 2;
if (angleX < -limit) {
    playerTransform.setAngleX(-limit);
}
else if (angleX > limit) {
    playerTransform.setAngleX(limit);
}

// Ustawia pozycję kamery 100 jednostek ponad graczem.
Transform3D camera = polygonRenderer.getCamera();
camera.setTo(playerTransform);
camera.getLocation().add(0,100,0);

}

```

Do nowej klasy GameCore3D dodaliśmy ciekawy efekt, polegający na zanikaniu po kilku sekundach tekstu na górze ekranu. W poprzednich programach demonstracyjnych na górze ekranu zawsze pojawiał się tekst: *Sterowanie ruchem: mysz/klawisze kurSORA. Esc zamyka program.*, jednak w nowej wersji ten sam tekst znika po kilku sekundach od uruchomienia programu. Efekt zanikania uzyskaliśmy dzięki zastosowaniu półprzezroczystego koloru:

```

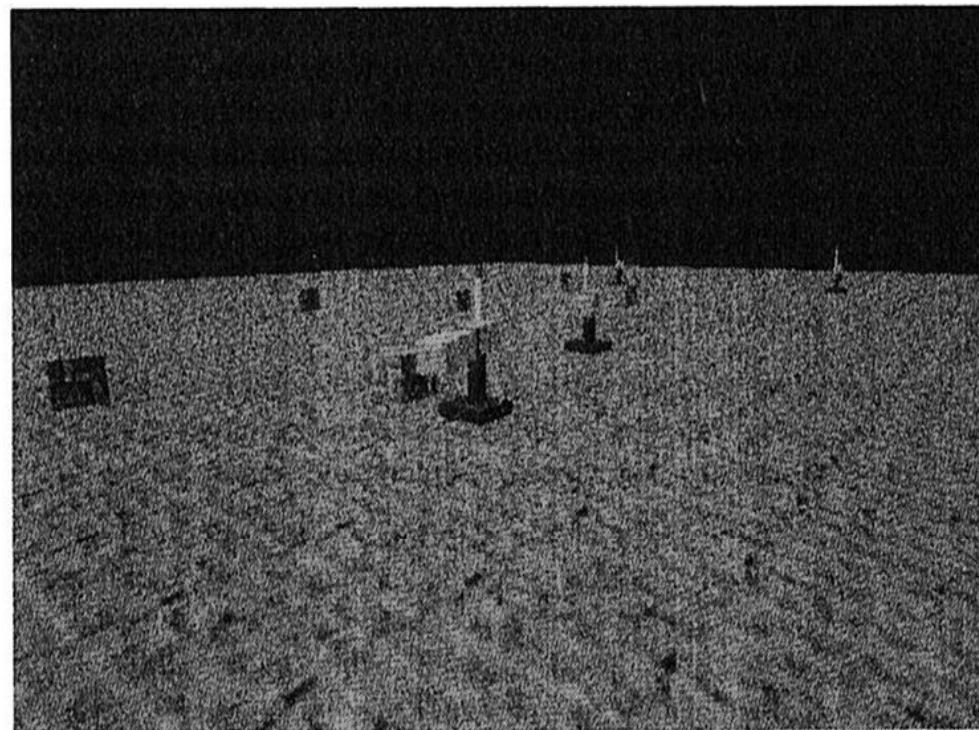
// Płynnie ukrywa tekst przez 500 ms:
long fade = INSTRUCTIONS_TIME - drawInstructionsTime;
if (fade < 500) {
    fade = fade * 255 / 500;
    g.setColor(new Color(0xffffffff | ((int)fade << 24), true));
}
else {
    g.setColor(Color.WHITE);
}

```

Omówiliśmy już wszystkie elementy potrzebne do uruchomienia programu GameObjectTest. Zrzut ekranu z widocznymi obiektami programu demonstracyjnego prezentuje rysunek 9.9.

Rysunek 9.9.

Zrzut ekranu z programu GameObjectTest pokazuje wielokąty wyświetlane we właściwej z-kolejności



Nie zapominaj, że w tym programie demonstracyjnym użyliśmy techniki z-buforowania. Jeśli chcesz się dowiedzieć, jak byłyby wyświetlane obiekty w grze bez z-buforowania, zmodyfikuj metodę checkDepth() klasy ZBuffer w taki sposób, by zawsze zwracała wartość true. Wielokąty będą wówczas rysowane w zupełnie innej kolejności, co spowoduje całkowicie nienaturalny wygląd sceny.

Jeśli w tej wersji programu demonstracyjnego GameObjectTest spojrzyesz w dół tak, że cały ekran będzie wypełniony zieloną powierzchnią, być może zauważysz, że liczba wyświetlanych klatek jest znacznie mniejsza niż w programach prezentowanych w poprzednich rozdziałach. Wynika to z faktu, że dla wszystkich pikseli wyświetlanych na ekranie wykorzystywany jest z-bufor, co powoduje istotne opóźnienia. Jak już wspomnieliśmy, z-buforowanie jest dobrą techniką dla obiektów w grze, ponieważ zajmują one zwykle tylko niewielką część ekranu. W kolejnym rozdziale, kiedy będziemy omawiać drzewa BSP, nauczysz się, jak stosować lepszy algorytm do znacznie szybszego rysowania statycznych wielokątów oraz jak efektywnie dodawać do sceny trójwymiarowe obiekty.

Możliwe rozszerzenia w przyszłości

Oto kilka dodatkowych pomysłów na rozszerzenie kodu zaprezentowanego w tym rozdziale:

- ◆ Dodanie efektu przyspieszenia do klasy MovingTransform3D. Dzięki temu obiekty mogłyby płynnie przyspieszać do osiągnięcia założonej prędkości i dalej utrzymywać tę prędkość. Takie rozwiązanie pozwoliłoby zwiększyć realizm ruchu gracza.
- ◆ Dodanie do klasy MovingTransform3D ruchu po okręgu. Dzięki temu obiekty mogłyby krążyć wokół innych obiektów.

- ◆ Umożliwienie obiektom `GameObject` wykorzystywanie klasy `MovingTransform3D` do płynnego obracania się podczas ruchu. Oczywiście gracz może to robić jednocześnie za pomocą myszy i klawiatury; zupełnie inaczej jest w przypadku obiektów w grze. Zamiast poruszania się, zatrzymywania, obracania i ruszania obiektów znacznie bardziej realistycznym rozwiązaniem byłoby obracanie się obiektów podczas ruchu postępowego. Obiekt mógłby wówczas zwalniać, by zmniejszyć promień skrętu (tak samo, jak zwalniasz, pokonując zakręty samochodem).
- ◆ Rozszerzenie obsługi plików *OBJ* i *MTL*. Obsługę jednolitych kolorów definiowanych w plikach *MTL* można łatwo zaimplementować, tworząc podklasę klasy `Texture`, reprezentującą pojedynczą barwę.
- ◆ Pełne wykorzystanie wyobraźni! Obiekty występujące w grze możesz zmusić do wykonywania znacznie większej liczby ciekawych czynności. Możesz zaimplementować np. poruszanie się obiektów po okręgu, dodać robotom nogi lub stworzyć duchy, które ścigają graczy tylko wówczas, gdy ci nie patrzą w ich stronę. Zachowania Twoich obiektów będą jeszcze ciekawsze, kiedy nauczysz się obsługiwać omówioną w rozdziale 11. technikę wykrywania kolizji.

Podsumowanie

Pozostał nam do omówienia w tej książce jeszcze jeden algorytm usuwania ukrytych powierzchni — drzewa BSP. Technika z-buforowania doskonale się sprawdza w przypadku poruszających się obiektów trójwymiarowych, jednak rysowanie wszystkich pikseli na ekranie z wykorzystaniem z-bufora może się okazać zbyt wolne. Rozwiążemy ten problem w kolejnym rozdziale, w którym nauczysz się wykorzystywać właśnie drzewa BSP.

Pamiętaj jednak o pojęciach, których nauczyłeś się z tego rozdziału. Omówiliśmy kilka technik usuwania ukrytych powierzchni i zaimplementowaliśmy z-bufor z wartościami 1/z. Oczywiście stworzyliśmy także opartą na tej implementacji z-bufora klasę rysującą wielokąty. Opracowaliśmy również system realizujący trójwymiarową animację opartą na grupach wielokątów i stworzyliśmy parser wczytujący trójwymiarowe modele z plików *OBJ*. Na końcu wykonaliśmy klasę bazową dla obiektów występujących w grze, opracowaliśmy kilka przykładowych obiektów i połączylismy wszystkie te elementy w programie demonstracyjnym. Ich połączenie dało imponujący efekt; jednak w kolejnych rozdziałach poszerzysz swoją wiedzę jeszcze bardziej.