

Stwórz wciągającą grę na platformę Android!



Tworzenie gier na platformę Android 4

J.F. DiMarzio



Apress®



Tytuł oryginału: Practical Android 4 Games Development

Tłumaczenie: Szymon Pietrzak

ISBN: 978-83-246-5087-3

Original edition copyright © 2011 by J. F. DiMarzio.
All rights reserved.

Polish edition copyright © 2013 by Helion S.A.
All rights reserved.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiegokolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

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

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

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

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Spis treści

	Przedmowa	11
	O autorze	13
	O recenzentach	14
	O twórcy grafiki do gier	15
	Podziękowania	16
	Wstęp	17
Część I	Planowanie i tworzenie gier 2D	19
Rozdział 1	Witaj w świecie gier na platformę Android	21
	Programowanie gier na platformę Android	21
	Rozpocznij od dobrej historii	22
	Dlaczego historia jest ważna	23
	Pisanie własnej historii	25
	Droga przed Tobą	27
	Przygotowanie narzędzi do programowania na platformę Android	27
	Instalacja OpenGL ES	29
	Wybór wersji platformy Android	30
	Podsumowanie	30
Rozdział 2	Star Fighter — strzelanka 2D	31
	Historia gry Star Fighter	31
	Z czego składa się gra?	33
	Czym jest silnik gry?	34
	Czym jest kod specyficzny dla gry?	35
	Silnik gry Star Fighter	38

Utworzenie projektu gry Star Fighter	38
Podsumowanie	40
Rozdział 3 Naciśnij Start: tworzenie menu	41
Tworzenie ekranu powitalnego	41
Tworzenie czynności	42
Tworzenie obrazu ekranu powitalnego	47
Praca z plikiem R.java	49
Tworzenie pliku szablonu	50
Tworzenie efektów rozmycia	55
Zarządzanie wątkami gry	58
Tworzenie głównego menu	63
Dodawanie obrazów przycisków	63
Ustawianie szablonów	65
Podpinanie przycisków	67
Dodawanie listenerów onClickListeners	69
Dodawanie muzyki	70
Tworzenie usługi muzycznej	72
Odtwarzanie swojej muzyki	77
Podsumowanie	79
Rozdział 4 Wyświetlanie środowiska	81
Renderowanie tła	81
Tworzenie czynności gry	82
Tworzenie renderera	86
Wczytywanie obrazka w OpenGL	92
Przewijanie tła	101
Dodawanie drugiej warstwy	108
Wczytywanie drugiej tekstury	110
Przewijanie drugiej warstwy	111
Praca z macierzami	112
Kończenie metody scrollBackground2()	113
Wyświetlanie gry z szybkością 60 klatek na sekundę	115
Zatrzymywanie pętli gry	116
Czyszczenie buforów OpenGL	118
Modyfikowanie głównego menu	119
Podsumowanie	120
Rozdział 5 Tworzenie postaci gracza	121
Animowanie sprite'ów	121
Wczytywanie postaci gracza	123
Tworzenie tablic mapujących tekstury	124
Nakładanie tekstury na postać gracza	128
Dostosowanie pętli gry	131

Poruszanie postacią	132
Rysowanie domyślnego stanu postaci	133
Oprogramowanie akcji PLAYER_RELEASE	135
Przesuwanie postaci w lewo	137
Wczytywanie odpowiedniego sprite'a	138
Wczytywanie drugiej ramki animacji	141
Przesuwanie postaci w prawo	144
Wczytywanie animacji przechyłu w prawo	145
Poruszanie postacią gracza przy pomocy zdarzenia dotykowego	148
Przetwarzanie zdarzenia MotionEvent	148
Przechwytywanie akcji ACTION_UP i ACTION_DOWN	151
Dostosowanie opóźnienia FPS	153
Podsumowanie	154
Rozdział 6 Dodawanie przeciwników	155
Porządkowanie kodu gry	155
Tworzenie klasy tekstury	156
Tworzenie klasy postaci przeciwnika	160
Dodawanie nowego arkusza sprite'ów	160
Tworzenie klasy SFEnemy	161
Krzywa Béziera	165
Podsumowanie	170
Rozdział 7 Wyposażenie przeciwników w podstawową sztuczną inteligencję	171
Przygotowanie przeciwników na wprowadzenie sztucznej inteligencji	171
Logika tworzenia przeciwników	173
Inicjalizacja przeciwników	175
Wczytywanie arkusza sprite'ów	177
Przegląd sztucznej inteligencji	177
Tworzenie metody moveEnemy()	178
Tworzenie pętli iterującej po tablicy enemies[]	178
Poruszanie każdym z przeciwników przy wykorzystaniu ich logiki	179
Tworzenie sztucznej inteligencji statku przechwytyującego	180
Dostosowywanie wierzchołków	181
Namierzanie pozycji gracza	182
Implementowanie ruchu po prostej pochyłej	184
Tworzenie sztucznej inteligencji statku zwiadowczego	189
Ustalanie losowego punktu docelowego dla statku zwiadowczego	190
Ruch po krzywej Béziera	191
Tworzenie sztucznej inteligencji statku wojennego	194
Podsumowanie	195

Rozdział 8	Broń się!	197
	Tworzenie arkusza sprite'ów uzbrojenia	197
	Tworzenie klasy dla uzbrojenia	198
	Nadawanie trajektorii pociskom	201
	Tworzenie tablicy uzbrojenia	201
	Dodanie drugiego arkusza sprite'ów	201
	Inicjalizowanie uzbrojenia	202
	Ruch pocisków	203
	Wykrywanie krawędzi ekranu	204
	Wywoływanie metody firePlayerWeapons()	206
	Implementacja wykrywania kolizji	207
	Odnoszenie obrazów w wyniku kolizji	207
	Tworzenie metody detectCollisions()	208
	Wykrywanie typów kolizji	209
	Usuwanie wystrzelonych poza ekran pocisków	210
	Dalsze poszerzanie zdobytej wiedzy	211
	Podsumowanie	212
	Przegląd kluczowych fragmentów kodu gry 2D	212
Rozdział 9	Publikowanie swojej gry	229
	Przygotowanie pliku AndroidManifest	229
	Przygotowanie do podpisania, ułożenia i wydania	230
	Sprawdzenie gotowości pliku AndroidManifest	232
	Tworzenie magazynu kluczy	233
	Podsumowanie	235
Część II	Tworzenie gier 3D	237
Rozdział 10	Blob Hunter — tworzenie gier 3D	239
	Porównanie gier 2D i 3D	239
	Tworzenie własnego projektu 3D	240
	BlobhunterActivity.java	240
	BHGameView.java	241
	BHGameRenderer.java	241
	BHEngine.java	242
	Tworzenie testu obiektu 3D	243
	Tworzenie stałej	243
	Tworzenie klasy BHWalls	244
	Tworzenie nowej instancji klasy BHWalls	246
	Mapowanie obrazka	247
	Korzystanie z gluPerspective()	248
	Tworzenie metody drawBackground()	250
	Końcowe poprawki	251
	Podsumowanie	253

Rozdział 11	Tworzenie realistycznego środowiska	255
	Używanie klasy BHWalls	255
	Tworzenie korytarza z wielu instancji klasy BHWalls	256
	Używanie klasy BHCorridor	257
	Tworzenie klasy BHCorridor	257
	Budowanie wielu ścian przy pomocy tablicy vertices[]	258
	Tworzenie tablicy texture[]	260
	Tworzenie metody draw()	263
	Dodawanie tekstury ściany	266
	Wywoływanie klasy BHCorridor	267
	Podsumowanie	268
Rozdział 12	Poruszanie się w trójwymiarowym środowisku	269
	Tworzenie interfejsu sterowania	269
	Modyfikowanie klasy BHEngine	270
	Modyfikowanie klasy BlobhunterActivity	271
	Pozwalanie graczowi na ruch do przodu	272
	Poruszanie się po korytarzu	273
	Dostosowywanie widoku gracza	275
	Podsumowanie	276
	Przegląd kluczowych fragmentów kodu gry 3D	276
	Skorowidz	283

ROZDZIAŁ 5



Tworzenie postaci gracza

Jak do tej pory zdążyłeś już wykonać całkiem sporo programowania i nauczyłeś się wiele o środowiskach OpenGL i Android — na tyle dużo, że powinieneś teraz dobrze znać drobne różnice pomiędzy OpenGL a innymi API, których mogłeś używać w przeszłości.

Nie napisałeś jeszcze powalającej liczby linii kodu, jednak to, co już stworzyłeś, stanowi dobre podwaliny Twojej gry i daje całkiem niezły efekt wizualny. Udało Ci się zaprogramować tło o dwóch przewijających się z różną szybkością warstwach, tło muzyczne, ekran powitalny i główne menu gry. Wszystkie te rzeczy mają jednak, w kontekście grywalności gry, jedną wspólną cechę — są straszliwie nudne.

Oznacza to, że gracz nie kupi Twojej gry tylko po to, by oglądać „odpicowane”, dwuwarstwowe tło przewijające się z góry na dół. Gracz potrzebuje odrobiny akcji i kontroli. Właśnie o tym wszystkim będzie traktował ten rozdział.

Stworzysz w nim swoją postać gracza. Pod koniec rozdziału będziesz miał wyświetloną na ekranie animowaną postać, którą gracz będzie mógł sterować. W pierwszym podrozdziale poznasz podstawowy element programowania gier 2D — animację sprite'ów. Następnie, przy pomocy OpenGL ES, wczytasz różne sprite'y z pełnego ich arkusza, by stworzyć złudzenie animacji postaci. Nauczysz się wczytywać różnorodne sprite'y w kluczowych momentach akcji, aby sprawić, że Twoja postać będzie wyglądała tak, jakby przechylała się podczas lotu.

Animowanie sprite'ów

Jednym z najstarszych narzędzi w warsztacie programisty gier 2D jest animacja sprite'ów. Wróć na chwilę pamięcią do którejkolwiek ze swoich ulubionych gier 2D — istnieje spore prawdopodobieństwo, że animacja dowolnych postaci została w nich wykonana przy pomocy animacji sprite'ów.

Technicznie rzecz biorąc, **sprite** to dowolny element graficzny gry 2D. Zgodnie z tą definicją Twoja postać gracza jest spritem. Sprite'y same w sobie są statycznymi obrazkami, wyświetlanymi na ekranie i niezmiennymi się. Animacja sprite'ów to proces, którego użyjesz do „ożywienia” postaci głównego bohatera, nawet jeśli jest nim statek kosmiczny.

■ **Ostrzeżenie:** Nie myl **animacji** z **poruszaniem**. Poruszanie sprite'a (obrazu, tekstury, wierzchołka czy modelu) po ekranie jest czymś zgoła innym niż jego animacja; te dwa pojęcia i umiejętności są całkowicie rozłączne.

Animacja sprite'a przeprowadzana jest przy pomocy efektu przypominającego przewracanie kartek skoroszytu. Pomyśl o dowolnej dwuwymiarowej grze, np. *Mario Brothers*, który jest jednym z najlepszych przykładów platformówek 2D wykorzystujących animację sprite'ów. W tej grze sterujesz postacią Maria, by poruszała się w prawo albo w lewo po przewijającym się w bok środowisku. Mario chodzi, a czasem biegnie, w kierunku, w którym każesz mu się poruszać. Jego nogi w oczywisty sposób są animowane sekwencją kroków.

Ta animacja kroków składa się w rzeczywistości z serii nieruchomych obrazków. Każdy z nich przedstawia inny moment wykonywania kroku. Kiedy gracz steruje postacią w lewo lub w prawo, różne obrazki są podmieniane, dając złudzenie, że Mario chodzi.

W grze *Star Fighter* wykorzystasz tę samą metodę do stworzenia kilku animacji dla swojego głównego bohatera. Głównym bohaterem, a zarazem postacią gracza jest statek kosmiczny, nie będzie on więc potrzebował animacji chodzenia. Statki kosmiczne wymagają jednak innych animacji. W tym rozdziale stworzysz animację przechyłu lecącego statku w prawo i w lewo, a w kolejnych — animacje wybuchów i kolizji.

Wspaniałą zaletą animacji sprite'ów jest to, że wszystkie umiejętności potrzebne do jej zaimplementowania zdobyłeś już w poprzednim rozdziale. Posiadałeś już bowiem umiejętność wczytywania tekstury do środowiska OpenGL i, co ważniejsze, nauczyłeś się mapować teksturę na zbiór wierzchołków. Klucz do animacji sprite'ów stanowi sposób, w jaki tekstura jest mapowana na wierzchołki.

Tekstury używane w implementacji animacji sprite'a nie są oddzielnymi obrazkami. Czas i moc obliczeniowa wymagane do wczytywania i zmapowania nowej tekstury 60 razy na sekundę — gdyby udało Ci się osiągnąć taką szybkość — przekraczałyby znacznie możliwości urządzenia z systemem Android. Zamiast tego użyjesz więc arkusza sprite'ów.

Arkusz sprite'ów to pojedynczy obrazek zawierający wszystkie odrębne obrazki wymagane do realizacji animacji sprite'a. Rysunek 5.1 przedstawia arkusz sprite'ów statku głównego bohatera gry.



Rysunek 5.1. Arkusz sprite'ów postaci głównego bohatera

■ **Uwaga:** Rysunek 5.1 nie przedstawia arkusza sprite'ów w całości. Rzeczywisty rozmiar wczytywanego do środowiska OpenGL obrazu to 512×512 pikseli. Dolna część obrazka, będąca jedynie przezroczystym obszarem, została przycięta, aby lepiej prezentował się on w książce.

W jaki sposób można więc animować obrazek składający się z mniejszych obrazków? W gruncie rzeczy to łatwiejsze, niż się spodziewasz. Wczytasz obrazek jako jedną teksturę, będziesz jednak wyświetlać jedynie ten jej fragment, który zawiera pokazywany graczowi obrazek. Kiedy będziesz chciał animować obrazek, użyjesz po prostu metody `glTranslatef()`, by przesunąć się do tej części tekstury, którą będziesz chciał wyświetlić.

Nie martw się, jeśli nie rozumiesz jeszcze w pełni tego sposobu animacji — pojmiesz go, składając go w całość w kolejnych częściach tego rozdziału. Pierwszym krokiem jest stworzenie klasy, która będzie obsługiwała wczytywanie i rysowanie postaci gracza.

-
- **Uwaga:** Być może się zastanawiasz, dlaczego statki w arkuszu sprite'ów są skierowane w dół, a nie w górę, tym bardziej że postać gracza ma się znajdować na dole ekranu i lecieć w kierunku jego górnej krawędzi. Odwrotna orientacja sprite'ów występuje ze względu na fakt, iż OpenGL renderuje wszystkie bitmapy od ostatniej linii do pierwszej. Dlatego też kiedy OpenGL wyrenderuje ten arkusz sprite'ów, na ekranie pojawi się on w formie ukazanej na rysunku 5.2.
-



Rysunek 5.2. Wygląd arkusza sprite'ów na ekranie

Oczywiście mógłbyś narysować arkusz sprite'ów poprawnie, a następnie użyć środowiska OpenGL do odwrócenia tekstury w prawidłowy sposób. Odwrócenie arkusza sprite'ów przy pomocy dowolnego narzędzia do obróbki obrazów jest jednak dość proste, a dzięki temu oszczędzasz środowisku OpenGL dodatkowej pracy potrzebnej do odwrócenia za Ciebie tekstury.

Wczytywanie postaci gracza

W poprzednim rozdziale stworzyłeś klasę, która wczytywała obraz tła jako teksturę, a następnie rysowała ten obraz na żądanie. Mechanizmy, których użyłeś do stworzenia tej klasy, są tymi samymi mechanizmami, których będziesz potrzebował, by wczytać i narysować swojego głównego bohatera. Dokonasz w nich drobnych zmian pozwalających Ci na zastosowanie arkusza sprite'ów, poza tym jednak kod powinien Ci wyglądać znajomo.

Rozpocznij od stworzenia w bazowym pakiecie projektu nowej klasy o nazwie `SFGoodGuy`:

```
package com.proandroidgames;

public class SFGoodGuy {

}
```

W klasie tej umieść załączki konstruktora, metody `draw()` i metody `loadTexture()`.

-
- **Wskazówka:** Pamiętaj, że pracując w środowisku Eclipse, możesz użyć skrótu `Alt+Shift+O`, by wykryć wszystkie pominięte przez siebie, a potrzebne do działania importy.
-

```
package com.proandroidgames;

import javax.microedition.khronos.opengl.GL10;

import android.content.Context;

public class SFGoodGuy {
```

```

    public SFGoodGuy() {
    }

    public void draw(GL10 gl) {
    }

    public void loadTexture(GL10 gl, int texture, Context context) {
    }
}

```

Następnie stwórz bufor, których będziesz używał wewnątrz klasy. Powinny one wyglądać identycznie jak te, które w poprzednim rozdziale stosowałeś do wczytywania tła.

Możesz także dodać kod tworzący tablicę `vertices[]`. Tablica ta będzie identyczna z tą używaną w klasie obsługującej tło.

```

package com.proandroidgames;

import java.nio.ByteBuffer;
import java.nio.FloatBuffer;

import javax.microedition.khronos.opengles.GL10;

import android.content.Context;

public class SFGoodGuy {

    private FloatBuffer vertexBuffer;
    private FloatBuffer textureBuffer;
    private ByteBuffer indexBuffer;
    private int[] textures = new int[1];

    private float vertices[] = {
        0.0f, 0.0f, 0.0f,
        1.0f, 0.0f, 0.0f,
        1.0f, 1.0f, 0.0f,
        0.0f, 1.0f, 0.0f,
    };

    public SFGoodGuy() {
    }

    public void draw(GL10 gl) {
    }

    public void loadTexture(GL10 gl, int texture, Context context) {
    }
}

```

Możesz teraz stworzyć tablicę mapującą teksturę.

Tworzenie tablic mapujących tekstury

Mapowanie tekstur to miejsce, w którym klasa `SFGoodGuy` będzie się różniła od klasy wczytującej tło. Tekstura, którą wczytasz do klasy, jest dużym arkuszem `sprite'ów` zawierającym pięć obrazów reprezentujących głównego bohatera. Twoim zadaniem jest wyświetlenie w danej chwili tylko jednego z tych obrazów.

Kluczem do zrozumienia, w jaki sposób należy przekazać środowisku OpenGL lokalizację obrazka, który chcesz wyświetlić, jest rozmieszczenie obrazków w arkuszu sprite'ów. Przyjrzyj się raz jeszcze arkuszowi sprite'ów przedstawionemu na rysunku 5.1. Zauważ, że obrazki rozmieszczone są równomiernie, z czterema obrazkami w pierwszym rzędzie i jednym obrazkiem w drugim. Mając jedynie 4 obrazki w pierwszym rzędzie tekstury i zakładając, że cała tekstura ma długość i wysokość 1 jednostki, możesz łatwo wywnioskować, że będziesz musiał wyświetlić jedynie szesnastą część całej tekstury, by wyświetlić jeden obrazek z pierwszego rzędu.

Oznacza to, że zamiast mapować całą teksturę, od (0, 0) do (1, 1), jak to zrobiłeś w przypadku tła, będziesz mapował jedynie jej szesnastą część, od (0, 0) do (0,25, 0,25). Będziesz mapował, a co za tym idzie, wyświetlał, jedynie pierwszy obrazek statku, używając zaledwie 0,25×0,25, czyli 1/16 tekstury.

Stwórz swoją tablicę tekstury, jak przedstawiono to poniżej:

```
package com.proandroidgames;

import java.nio.ByteBuffer;
import java.nio.FloatBuffer;

import javax.microedition.khronos.opengles.GL10;

import android.content.Context;

public class SFGoodGuy {

    private FloatBuffer vertexBuffer;
    private FloatBuffer textureBuffer;
    private ByteBuffer indexBuffer;
    private int[] textures = new int[1];

    private float vertices[] = {
        0.0f, 0.0f, 0.0f,
        1.0f, 0.0f, 0.0f,
        1.0f, 1.0f, 0.0f,
        0.0f, 1.0f, 0.0f,
    };

    private float texture[] = {
        0.0f, 0.0f,
        0.25f, 0.0f,
        0.25f, 0.25f,
        0.0f, 0.25f,
    };

    public SFGoodGuy() {
    }

    public void draw(GL10 gl) {
    }

    public void loadTexture(GL10 gl, int texture, Context context) {
    }
}
```

Tablica krawędzi, metoda draw() oraz konstruktor są identyczne z tymi użytymi w klasie SFBackground:

```
package com.proandroidgames;

import java.nio.ByteBuffer;
import java.nio.ByteOrder;
```

```

import java.nio.FloatBuffer;

import javax.microedition.khronos.opengles.GL10;

import android.content.Context;

public class SFGoodGuy {

    private FloatBuffer vertexBuffer;
    private FloatBuffer textureBuffer;
    private ByteBuffer indexBuffer;
    private int[] textures = new int[1];

    private float vertices[] = {
        0.0f, 0.0f, 0.0f,
        1.0f, 0.0f, 0.0f,
        1.0f, 1.0f, 0.0f,
        0.0f, 1.0f, 0.0f,
    };

    private float texture[] = {
        0.0f, 0.0f,
        0.25f, 0.0f,
        0.25f, 0.25f,
        0.0f, 0.25f,
    };

    private byte indices[] = {
        0, 1, 2,
        0, 2, 3,
    };

    public SFGoodGuy() {
        ByteBuffer byteBuf = ByteBuffer.allocateDirect(vertices.length * 4);
        byteBuf.order(ByteOrder.nativeOrder());
        vertexBuffer = byteBuf.asFloatBuffer();
        vertexBuffer.put(vertices);
        vertexBuffer.position(0);

        byteBuf = ByteBuffer.allocateDirect(texture.length * 4);
        byteBuf.order(ByteOrder.nativeOrder());
        textureBuffer = byteBuf.asFloatBuffer();
        textureBuffer.put(texture);
        textureBuffer.position(0);

        indexBuffer = ByteBuffer.allocateDirect(indices.length);
        indexBuffer.put(indices);
        indexBuffer.position(0);
    }

    public void draw(GL10 gl) {
        gl.glBindTexture(GL10.GL_TEXTURE_2D, textures[0]);
        gl.glFrontFace(GL10.GL_CCW);
        gl.glEnable(GL10.GL_CULL_FACE);
        gl.glCullFace(GL10.GL_BACK);
        gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
        gl.glEnableClientState(GL10.GL_TEXTURE_COORD_ARRAY);
    }
}

```

```

gl.glVertexPointer(3, GL10.GL_FLOAT, 0, vertexBuffer);
gl.glTexCoordPointer(2, GL10.GL_FLOAT, 0, textureBuffer);
gl.glDrawElements(GL10.GL_TRIANGLES, indices.length,
    GL10.GL_UNSIGNED_BYTE, indexBuffer);
gl.glDisableClientState(GL10.GL_VERTEX_ARRAY);
gl.glDisableClientState(GL10.GL_TEXTURE_COORD_ARRAY);
gl.glDisable(GL10.GL_CULL_FACE);
}

public void loadTexture(GL10 gl, int texture, Context context) {
}
}

```

Zanim zakończysz pracę nad klasą `SFGoodGuy`, musisz dokonać w niej jeszcze jednej zmiany. W klasie `SFBackground` w metodzie `loadTexture()` podawałeś do metody `glTexParameterf()` parametr `GL_REPEAT`, by uruchomić powtarzanie tekstury w miarę przesuwania wierzchołków. Nie jest to jednak potrzebne w przypadku postaci głównego bohatera, dlatego też zmienisz ten parametr na `GL_CLAMP_TO_EDGE`.

Dokończ swoją implementację klasy `SFGoodGuy`, umieszczając w metodzie `loadTexture()` następujący kod:

```

...
public void loadTexture(GL10 gl, int texture, Context context) {
    InputStream imagestream =
        context.getResources().openRawResource(texture);
    Bitmap bitmap = null;
    try {
        bitmap = BitmapFactory.decodeStream(imagestream);
    } catch (Exception e) {
    } finally {
        try {
            imagestream.close();
            imagestream = null;
        } catch (IOException e) {
        }
    }
    gl.glGenTextures(1, textures, 0);
    gl.glBindTexture(GL10.GL_TEXTURE_2D, textures[0]);
    gl.glTexParameterf(GL10.GL_TEXTURE_2D, GL10.GL_TEXTURE_MIN_FILTER,
        GL10.GL_NEAREST);
    gl.glTexParameterf(GL10.GL_TEXTURE_2D, GL10.GL_TEXTURE_MAG_FILTER,
        GL10.GL_LINEAR);
    gl.glTexParameterf(GL10.GL_TEXTURE_2D, GL10.GL_TEXTURE_WRAP_S,
        GL10.GL_REPEAT);
    gl.glTexParameterf(GL10.GL_TEXTURE_2D, GL10.GL_TEXTURE_WRAP_T,
        GL10.GL_REPEAT);
    GLUtils.texImage2D(GL10.GL_TEXTURE_2D, 0, bitmap, 0);
    bitmap.recycle();
}
}

```

Jesteś teraz w posiadaniu w pełni funkcjonalnej klasy, która wczyta teksturę postaci gracza jako arkusz `sprite'ów`, wyświetli pierwszego `sprite'a` z arkusza i nie będzie zawijać tekstury w momencie, gdy postać ta będzie się poruszała.

Nakładanie tekstury na postać gracza

Kolejnym krokiem na drodze do wczytania postaci gracza jest utworzenie nowej instancji klasy `SFGoodGuy` i wczytanie do niej tekstury. Zapisz i zamknij plik klasy `SFGoodGuy` — na razie nie będziesz musiał dodawać do niego więcej kodu.

Dodaj teraz do klasy `SFEngine` kilka prostych zmiennych i stałych. Będziesz z nich korzystał w pętli gry.

W pierwszej kolejności dodasz zmienną o nazwie `playerFlightAction`. Będziesz jej używał do śledzenia akcji, które gracz wykonał, aby odpowiedzieć na nie odpowiednio w pętli gry.

```
package com.proandroidgames;

import android.content.Context;
import android.content.Intent;
import android.view.View;

public class SFEngine {
    ...
    public static int playerFlightAction = 0;

    /* Zamknij wątki gry i wyjdź z niej */
    public boolean onExit(View v) {
        try {
            Intent bgmusic = new Intent(context, SFMusic.class);
            context.stopService(bgmusic);
            musicThread.stop();
            return true;
        } catch (Exception e) {
            return false;
        }
    }
}
```

Następnie dodaj do projektu plik opisanego na początku tego podrozdziału arkusza sprite'ów (*good_sprite.png*) i utwórz w klasie silnika gry stałą wskazującą na odpowiadający mu zasób.

```
package com.proandroidgames;

import android.content.Context;
import android.content.Intent;
import android.view.View;

public class SFEngine {
    ...
    public static int playerFlightAction = 0;
    public static final int PLAYER_SHIP = R.drawable.good_sprite;

    /* Zamknij wątki gry i wyjdź z niej */
    public boolean onExit(View v) {
        try {
            Intent bgmusic = new Intent(context, SFMusic.class);
            context.stopService(bgmusic);
            musicThread.stop();
            return true;
        } catch (Exception e) {
            return false;
        }
    }
}
```


Kolejne trzy stałe będą opisywały akcje, które może wykonać gracz. Ich wartości będą przypisywane do zmiennej `playerFlightAction`, kiedy gracz będzie próbował sterować postacią.

```
package com.proandroidgames;

import android.content.Context;
import android.content.Intent;
import android.view.View;

public class SFEngine {
    ...
    public static int playerFlightAction = 0;
    public static final int PLAYER_SHIP = R.drawable.good_sprite;
    public static final int PLAYER_BANK_LEFT_1 = 1;
    public static final int PLAYER_RELEASE = 3;
    public static final int PLAYER_BANK_RIGHT_1 = 4;

    /* Zamknij wątki gry i wyjdź z niej */
    public boolean onExit(View v) {
        try {
            Intent bgmusic = new Intent(context, SFMusic.class);
            context.stopService(bgmusic);
            musicThread.stop();
            return true;
        } catch (Exception e) {
            return false;
        }
    }
}
```

W zależności od tego, jak spostrzegawczy jesteś, jeśli chodzi o stałe dodane przed chwilą do klasy `SFEngine`, być może już się zastanawiasz, dlaczego `PLAYER_BANK_LEFT_1` ma wartość 1, a `PLAYER_RELEASE` wartość 3. Wartości te będą reprezentowały etapy animacji Twojego sprite'a. W arkuszu sprite'ów znajdują się dwa etapy w animacji przechyłu w lewo i dwa etapy w animacji przechyłu w prawo. W kodzie pętli jednakże zauważysz, że pomiędzy stałymi `PLAYER_BANK_LEFT_1` a stałymi `PLAYER_RELEASE` znajduje się `PLAYER_BANK_LEFT_2` o wartości 2, stała ta nie będzie jednak obecna w klasie `SFEngine`. Rozwiązanie to z pewnością wyda Ci się sensowniejsze, kiedy zobaczysz je w akcji w dalszych częściach książki.

Kolejna stała, której będziesz potrzebował, będzie wskazywała, ile przebiegów pętli będzie odpowiadało jednej klatce animacji sprite'a. Pamiętaj, że wielką różnicę pomiędzy postacią gracza a tłem stanowi fakt, iż animacja postaci zachodzi w momencie jej poruszania się po ekranie. Śledzenie tej animacji jest niełatwym zadaniem. Główna pętla gry działa z szybkością 60 przebiegów na sekundę. Gdybyś uruchamiał nową klatkę animacji sprite'a w każdym przebiegu pętli, Twoja animacja zakończyłaby się, zanim gracz miałby szansę się nią nacieszyć. Stała `PLAYER_FRAMES_BETWEEN_ANI` przyjmie wartość 9, co oznacza, że jedna klatka animacji sprite'a będzie rysowana co dziewięć iteracji głównej pętli gry.

```
package com.proandroidgames;

import android.content.Context;
import android.content.Intent;
import android.view.View;

public class SFEngine {
    ...
    public static int playerFlightAction = 0;
    public static final int PLAYER_SHIP = R.drawable.good_sprite;
    public static final int PLAYER_BANK_LEFT_1 = 1;
```

```

public static final int PLAYER_RELEASE = 3;
public static final int PLAYER_BANK_RIGHT_1 = 4;
public static final int PLAYER_FRAMES_BETWEEN_ANI = 9;

/* Zamknij wątki gry i wyjdź z niej */
public boolean onExit(View v) {
    try {
        Intent bgmusic = new Intent(context, SFMusic.class);
        context.stopService(bgmusic);
        musicThread.stop();
        return true;
    } catch (Exception e) {
        return false;
    }
}
}

```

Na koniec dodaj jeszcze jedną stałą i jedną zmienną. Będą one reprezentowały prędkość, z jaką statek gracza będzie się poruszał od lewej do prawej, oraz aktualną pozycję statku na osi x.

```

package com.proandroidgames;

import android.content.Context;
import android.content.Intent;
import android.view.View;

public class SFEngine {
    ...
    public static int playerFlightAction = 0;
    public static final int PLAYER_SHIP = R.drawable.good_sprite;
    public static final int PLAYER_BANK_LEFT_1 = 1;
    public static final int PLAYER_RELEASE = 3;
    public static final int PLAYER_BANK_RIGHT_1 = 4;
    public static final int PLAYER_FRAMES_BETWEEN_ANI = 9;
    public static final float PLAYER_BANK_SPEED = .1f;
    public static float playerBankPosX = 1.75f;

    /* Zamknij wątki gry i wyjdź z niej */
    public boolean onExit(View v) {
        try {
            Intent bgmusic = new Intent(context, SFMusic.class);
            context.stopService(bgmusic);
            musicThread.stop();
            return true;
        } catch (Exception e) {
            return false;
        }
    }
}

```

Plik klasy SFEngine zawiera teraz cały kod potrzebny Ci do zaimplementowania postaci gracza. Zapisz go i zamknij.

Otwórz plik *SFGameRenderer.java*. Zawiera on kod głównej pętli Twojej gry. W poprzednim rozdziale stworzyłeś pętlę gry i dodałeś dwie metody rysujące i przewijające dwie niezależne warstwy tła. Teraz dodasz do pętli gry kod rysujący postać gracza i poruszający ją.

Dostosowanie pętli gry

Pierwszym krokiem jest stworzenie zmiennej `player1`, przechowującej nową instancję klasy `SFGoodGuy`:

```
...
public class SFGGameRenderer implements Renderer {

    private SFBackground background = new SFBackground();
    private SFBackground background2 = new SFBackground();
    private SFGoodGuy player1 = new SFGoodGuy();

    private float bgScroll1;
    private float bgScroll2;

    ...
}
```

Zmienna `player1` będzie używana w taki sam sposób jak zmienne `background` i `background2`. Wywołasz jej metody `loadTexture()` i `draw()`, by wczytać do gry postać gracza.

Musisz także utworzyć zmienną, która będzie śledziła, ile iteracji pętli gry zostało wykonanych, tak abyś wiedział, kiedy przerzucać ramki w swojej animacji sprite'a.

```
...
public class SFGGameRenderer implements Renderer {

    private SFBackground background = new SFBackground();
    private SFBackground background2 = new SFBackground();
    private SFGoodGuy player1 = new SFGoodGuy();
    private int goodGuyBankFrames = 0;

    private float bgScroll1;
    private float bgScroll2;

    ...
}
```

Następnie znajdź w klasie renderera `SFGGameRenderer` metodę `onSurfaceCreated()`. Obsługuje ona wczytywanie tekstur gry. W poprzednim rozdziale wywołałeś w tej metodzie metody wczytujące tekstury do obiektów `background` i `background2`. Teraz musisz dodać do niej wywołanie metody `loadTexture()` zmiennej `player1`.

```
package com.proandroidgames;

import javax.microedition.khronos.egl.EGLConfig;
import javax.microedition.khronos.opengles.GL10;

import android.opengl.GLSurfaceView.Renderer;

public class SFGGameRenderer implements Renderer {

    private SFBackground background = new SFBackground();
    private SFBackground background2 = new SFBackground();
    private SFGoodGuy player1 = new SFGoodGuy();
    private int goodGuyBankFrames = 0;
    ...
    @Override
    public void onSurfaceCreated(GL10 gl, EGLConfig config) {
        gl.glEnable(GL10.GL_TEXTURE_2D);
        gl.glClearDepthf(1.0f);
        gl.glEnable(GL10.GL_DEPTH_TEST);
    }
}
```

```

        gl.glDepthFunc(GL10.GL_EQUAL);

        gl.glEnable(GL10.GL_BLEND);
        gl.glBlendFunc(GL10.GL_ONE, GL10.GL_ONE);

        background.loadTexture(gl, SFEngine.BACKGROUND_LAYER_ONE,
                               SFEngine.context);
        background2.loadTexture(gl, SFEngine.BACKGROUND_LAYER_TWO,
                                SFEngine.context);
        player1.loadTexture(gl, SFEngine.PLAYER_SHIP, SFEngine.context);
    }
}

```

Jak dotąd cały kod był dość podstawowy — tworzył i wczytywał teksturę. Czas teraz na bardziej treściwą część rozdziału — napisanie metody, która będzie kontrolowała sterowanie postacią gracza.

Poruszanie postacią

Ten podrozdział pomoże Ci stworzyć kod potrzebny do poruszania postacią gracza na ekranie. W tym celu stworzysz nową metodę, wykonującą zadania związane z poruszaniem postacią gracza, którą następnie będziesz wywoływał w głównej pętli swej gry. W klasie `SFGameRenderer` stwórz nową metodę przyjmującą jako parametr referencję do instancji klasy `GL10`.

```

package com.proandroidgames;

import javax.microedition.khronos.egl.EGLConfig;
import javax.microedition.khronos.opengles.GL10;

import android.opengl.GLSurfaceView.Renderer;

public class SFGameRenderer implements Renderer {
    ...

    private void movePlayer1(GL10 gl) {

    }
    ...
}

```

Wewnątrz metody `movePlayer1()` użyjesz instrukcji `switch` sprawdzającej wartość zmiennej całkowitoliczbowej `playerFlightAction`, którą wcześniej w tym rozdziale dodałeś do klasy `SFEngine`. Jeżeli nigdy wcześniej nie używałeś tej instrukcji, `switch` sprawdzi wartość podanego do niego obiektu (`playerFlightAction`) i wykona odpowiedni kod w zależności od wartości tego obiektu. Przypadki obsługiwane w tej instrukcji `switch` to `PLAYER_BANK_LEFT_1`, `PLAYER_RELEASE`, `PLAYER_BANK_RIGHT_1` oraz `default` (domyślny).

```

package com.proandroidgames;

import javax.microedition.khronos.egl.EGLConfig;
import javax.microedition.khronos.opengles.GL10;

import android.opengl.GLSurfaceView.Renderer;

public class SFGameRenderer implements Renderer {
    ...

```

```

private void movePlayer1(GL10 gl) {
    switch (SFEEngine.playerFlightAction) {
        case SFEEngine.PLAYER_BANK_LEFT_1:
            break;
        case SFEEngine.PLAYER_BANK_RIGHT_1:
            break;
        case SFEEngine.PLAYER_RELEASE:
            break;
        default:
            break;
    }
}
...
}

```

Rozpocznijmy od przypadku domyślnego (default). Zostanie on wywołany, gdy gracz nie wykonał swoją postacią żadnej akcji.

Rysowanie domyślnego stanu postaci

W tej chwili wierzchołki mają taki sam rozmiar jak cały ekran. Gdybyś więc teraz narysował postać gracza, zajmowałaby ona cały ekran. Aby postać ta dobrze wyglądała w grze, będziesz ją musiał przeskalować o około 75%.

Aby to zrobić, użyjesz metody `glScalef()`. Przemnożenie skali przez 0,25 zmniejszy rozmiar statku do jednej czwartej jego oryginalnego rozmiaru. Pociąga to za sobą istotne konsekwencje, których musisz być świadomy.

W poprzednim rozdziale miałeś okazję zauważyć, że aby przeskalować wierzchołki lub dokonać ich translacji, musisz pracować w trybie macierzy modelu. Dowolna operacja wykonywana w dowolnym trybie macierzy wpływa na **wszystkie** elementy, które obejmuje ten tryb. Dlatego też jeśli przeskalujesz statek gracza przez 0,25, przeskalujesz także całe osie x i y . Innymi słowy, jeżeli przy domyślnej skali 0 (pełen ekran) osie x i y rozpoczynały się w 0, a kończyły w 1, po przemnożeniu skali przez 0,25 osie te będą rozpoczynały się w 0, a kończyły w 4.

To ważna dla Ciebie informacja, ponieważ próbując śledzić położenie gracza, będziesz musiał pamiętać, że choć tło może się przewijać od 0 do 1, gracz może się poruszać od 0 do 4.

Wczytaj teraz widok macierzy modelu i przeskaluj postać gracza o 0,25 na osiach x i y .

```

package com.proandroidgames;

import javax.microedition.khronos.egl.EGLConfig;
import javax.microedition.khronos.opengles.GL10;

import android.opengl.GLSurfaceView.Renderer;

public class SFGGameRenderer implements Renderer {
    ...

    private void movePlayer1(GL10 gl) {
        switch (SFEEngine.playerFlightAction) {
            case SFEEngine.PLAYER_BANK_LEFT_1:
                break;
            case SFEEngine.PLAYER_BANK_RIGHT_1:
                break;
            case SFEEngine.PLAYER_RELEASE:
                break;
            default:
                gl.glMatrixMode(GL10.GL_MODELVIEW);

```

```

        gl.glLoadIdentity();
        gl.glPushMatrix();
        gl.glScalef(.25f, .25f, 1f);

        break;
    }
}
...
}

```

Następnie dokonaj translacji macierzy modelu na osi x o wartość zmiennej `playerBankPosX`. Zmienna ta będzie przechowywała aktualną pozycję postaci gracza na osi x . Dlatego też za każdym razem, gdy gracz nie podejmie żadnej akcji, jego postać pozostanie w miejscu, w którym znajdowała się ostatnio.

```

package com.proandroidgames;

import javax.microedition.khronos.egl.EGLConfig;
import javax.microedition.khronos.opengles.GL10;

import android.opengl.GLSurfaceView.Renderer;

public class SFGameRenderer implements Renderer {
    ...

    private void movePlayer1(GL10 gl) {
        switch (SFEngine.playerFlightAction) {
            case SFEngine.PLAYER_BANK_LEFT_1:
                break;
            case SFEngine.PLAYER_BANK_RIGHT_1:
                break;
            case SFEngine.PLAYER_RELEASE:
                break;
            default:
                gl.glMatrixMode(GL10.GL_MODELVIEW);
                gl.glLoadIdentity();
                gl.glPushMatrix();
                gl.glScalef(.25f, .25f, 1f);
                gl.glTranslatef(SFEngine.playerBankPosX, 0f, 0f);

                break;
        }
    }
    ...
}

```

Kiedy postać gracza jest w spoczynku, nie ma potrzeby wykonywania dodatkowych akcji, wczytaj więc macierz tekstur i upewnij się, że jest ona w położeniu domyślnym, czyli na pierwszym obrazku w arkuszu sprite'ów. Pamiętaj, iż to właśnie tryb macierzy tekstur będzie przez Ciebie wykorzystywany do zmieniania pozycji tekstury w arkuszu sprite'ów, a co za tym idzie, do „przerzucania” kolejnych klatek animacji. Jeśli gracz nie porusza swoją postacią, nie powinna być ona animowana, dlatego macierz tekstur powinna się znajdować w swojej pierwszej, domyślnej pozycji.

```

package com.proandroidgames;

import javax.microedition.khronos.egl.EGLConfig;
import javax.microedition.khronos.opengles.GL10;

```

```

import android.opengl.GLSurfaceView.Renderer;

public class SFGGameRenderer implements Renderer {
    ...

    private void movePlayer1(GL10 gl) {
        switch (SFEngine.playerFlightAction) {
            case SFEngine.PLAYER_BANK_LEFT_1:
                break;
            case SFEngine.PLAYER_BANK_RIGHT_1:
                break;
            case SFEngine.PLAYER_RELEASE:
                break;
            default:
                gl.glMatrixMode(GL10.GL_MODELVIEW);
                gl.glLoadIdentity();
                gl.glPushMatrix();
                gl.glScalef(.25f, .25f, 1f);
                gl.glTranslatef(SFEngine.playerBankPosX, 0f, 0f);
                gl.glMatrixMode(GL10.GL_TEXTURE);
                gl.glLoadIdentity();
                gl.glTranslatef(0.0f, 0.0f, 0.0f);
                player1.draw(gl);
                gl.glPopMatrix();
                gl.glLoadIdentity();

                break;
        }
    }
    ...
}

```

Kolejnym oprogramowywanym przez Ciebie przypadkiem w instrukcji switch będzie PLAYER_RELEASE. Akcja PLAYER_RELEASE będzie wywoływana, kiedy gracz zwolni sterowanie po przesunięciu postaci. Choć nie stworzyłeś jeszcze faktycznych mechanizmów sterowania grą, gracz będzie dotykał elementu sterującego, aby przesunąć swoją postać. Kiedy gracz puści ten element, przerywając w ten sposób ruch postaci, wywołana zostanie akcja PLAYER_RELEASE.

Oprogramowanie akcji PLAYER_RELEASE

Na tę chwilę przypadek PLAYER_RELEASE będzie wykonywał te same czynności co przypadek domyślny — postać gracza pozostanie tam, gdzie ją pozostawiono na ekranie, i niezależnie od tego, która z tekstur z arkusza sprite'ów była wyświetlana, nastąpi powrót do pierwszej tekstury w arkuszu. Skopiuj i wklej cały blok kodu z przypadku default do PLAYER_RELEASE.

```

package com.proandroidgames;

import javax.microedition.khronos.egl.EGLConfig;
import javax.microedition.khronos.opengles.GL10;

import android.opengl.GLSurfaceView.Renderer;

public class SFGGameRenderer implements Renderer {
    ...

    private void movePlayer1(GL10 gl) {
        switch (SFEngine.playerFlightAction) {

```

```

        case SFEngine.PLAYER_BANK_LEFT_1:
            break;
        case SFEngine.PLAYER_BANK_RIGHT_1:
            break;
        case SFEngine.PLAYER_RELEASE:
            gl.glMatrixMode(GL10.GL_MODELVIEW);
            gl.glLoadIdentity();
            gl.glPushMatrix();
            gl.glScalef(.25f, .25f, 1f);
            gl.glTranslatef(SFEngine.playerBankPosX, 0f, 0f);
            gl.glMatrixMode(GL10.GL_TEXTURE);
            gl.glLoadIdentity();
            gl.glTranslatef(0.0f, 0.0f, 0.0f);
            player1.draw(gl);
            gl.glPopMatrix();
            gl.glLoadIdentity();

            break;
        ...
    }
    ...
}

```

Zanim skończysz pracę nad przypadkiem `PLAYER_RELEASE`, musisz dodać jeszcze jedną linię kodu. Wcześniej w tym rozdziale dowiedziałeś się, że nie możesz zmieniać klatek animacji swojego `sprite'a` z taką samą szybkością, jaką ma główna pętla gry (60 klatek na sekundę), zawierając bowiem jedynie dwie ramki animacji `sprite'a`. Twoja animacja skończyłaby się, zanim gracz by ją w ogóle zauważył. Potrzebujesz więc zmiennej przechowującej liczbę przebiegów głównej pętli gry, które miały już miejsce. Wiedząc, ile razy pętla została już wykonana, możesz tę liczbę porównać z wartością stałej `PLAYER_FRAMES_BETWEEN_ANI`, by określić, kiedy przerzucić klatki animacji `sprite'a`. Utworzona przez Ciebie wcześniej w tym rozdziale zmienna `goodGuyBankFrames` będzie przez Ciebie używana do śledzenia liczby wykonanych iteracji głównej pętli gry.

Wewnątrz kodu obsługującego przypadek `PLAYER_RELEASE` dodaj wyróżnioną poniżej linię kodu, aby zwiększyć o jeden wartość zmiennej `goodGuyBankFrames` w każdej iteracji głównej pętli.

```

package com.proandroidgames;

import javax.microedition.khronos.egl.EGLConfig;
import javax.microedition.khronos.opengles.GL10;

import android.opengl.GLSurfaceView.Renderer;

public class SFGGameRenderer implements Renderer {
    ...

    private void movePlayer1(GL10 gl) {
        switch (SFEngine.playerFlightAction) {
            case SFEngine.PLAYER_BANK_LEFT_1:
                break;
            case SFEngine.PLAYER_BANK_RIGHT_1:
                break;
            case SFEngine.PLAYER_RELEASE:
                gl.glMatrixMode(GL10.GL_MODELVIEW);
                gl.glLoadIdentity();
                gl.glPushMatrix();
                gl.glScalef(.25f, .25f, 1f);
                gl.glTranslatef(SFEngine.playerBankPosX, 0f, 0f);

```



```

        gl.glMatrixMode(GL10.GL_TEXTURE);
        gl.glLoadIdentity();
        gl.glTranslatef(0.0f, 0.0f, 0.0f);
        player1.draw(gl);
        gl.glPopMatrix();
        gl.glLoadIdentity();
        goodGuyBankFrames += 1;

        break;
    }
}

```

Przypadki `PLAYER_RELEASE` i `default` były najłatwiejszymi z czterech możliwych przypadków w Twojej metodzie `movePlayer1()`. Musisz teraz stworzyć kod obsługujący wywołanie akcji `PLAYER_BANK_LEFT_1`.

Akcja `PLAYER_BANK_LEFT_1` jest wywoływana, kiedy gracz użyje elementów interfejsu sterowania, by przechylić statek głównego bohatera w lewo. Oznacza to nie tylko, że musisz przesunąć postać gracza w lewo na osi *x*, lecz także, że musisz stworzyć animację postaci, używając dwóch reprezentujących przechyl w lewo obrazków z arkusza `sprite`’ów.

Przesuwanie postaci w lewo

W środowisku OpenGL operacje przemieszczania postaci wzdłuż osi *x* i zmiana pozycji w arkuszu `sprite`’ów wykorzystują dwa różne tryby macierzy. Do przesunięcia postaci wzdłuż osi *x* będziesz musiał użyć trybu macierzy modelu; przesunięcie tekstury w arkuszu `sprite`’ów, a co za tym idzie, stworzenie animacji przechyłu, będzie wykorzystywało tryb macierzy tekstur. Rozpocznijmy od trybu macierzy modelu.

Pierwszym krokiem jest wczytanie trybu macierzy modelu i ustawienie jego skali na 0,25 na osiach *x* i *y*.

```

package com.proandroidgames;

import javax.microedition.khronos.egl.EGLConfig;
import javax.microedition.khronos.opengl.GL10;

import android.opengl.GLSurfaceView.Renderer;

public class SFGameRenderer implements Renderer {
    ...

    private void movePlayer1(GL10 gl) {
        switch (SFEngine.playerFlightAction) {
            case SFEngine.PLAYER_BANK_LEFT_1:
                gl.glMatrixMode(GL10.GL_MODELVIEW);
                gl.glLoadIdentity();
                gl.glPushMatrix();
                gl.glScalef(.25f, .25f, 1f);

                break;
            ...
        }
    }
}

```

Następnie, używając metody `glTranslatef()`, przesuniesz wierzchołki wzdłuż osi x . Odejmiesz wartość `PLAYER_BANK_SPEED` od aktualnej pozycji postaci gracza na osi x , przechowywanej w zmiennej `playerBankPosX`. (Ponieważ chcesz przesunąć postać w lewo wzdłuż osi x , wykonujesz operację odejmowania w celu uzyskania docelowej pozycji postaci. Gdybyś poruszał postać w prawo, użyłbyś operacji dodawania). Następnie zastosujesz metodę `glTranslatef()`, by przesunąć wierzchołki na pozycję określoną przez `playerBankPosX`.

```
package com.proandroidgames;

import javax.microedition.khronos.egl.EGLConfig;
import javax.microedition.khronos.opengles.GL10;

import android.opengl.GLSurfaceView.Renderer;

public class SFGameRenderer implements Renderer {
    ...

    private void movePlayer1(GL10 gl) {
        switch (SFEngine.playerFlightAction) {
            case SFEngine.PLAYER_BANK_LEFT_1:
                gl.glMatrixMode(GL10.GL_MODELVIEW);
                gl.glLoadIdentity();
                gl.glPushMatrix();
                gl.glScalef(.25f, .25f, 1f);
                SFEngine.playerBankPosX -= SFEngine.PLAYER_BANK_SPEED;
                gl.glTranslatef(SFEngine.playerBankPosX, 0f, 0f);

                break;
            ...
        }
    }
    ...
}
```

Skoro przemieszczasz już postać w lewo wzdłuż osi x , musisz jeszcze wyświetlić kolejną ramkę animacji sprite'a.

Wczytywanie odpowiedniego sprite'a

Przypatrzyć się raz jeszcze arkuszowi sprite'ów przedstawionemu na rysunku 5.1. Zauważ, że dwie klatki animacji odpowiadające przechyłowi na lewą stronę to czwarta klatka w pierwszym rzędzie i pierwsza w drugim (pamiętaj, że choć wydaje Ci się, iż arkusz sugeruje odwrotny kierunek przechyłu, jest on odwracany w pionie tak, że klatki, które sprawiają wrażenie zawierania ruchu w prawo, będą w rezultacie wyrenderowane jako ruch w lewo).

Wczytaj tryb macierzy tekstury i dokonaj translacji tekstury, aby wyświetlić czwarty obrazek w pierwszym rzędzie. Ponieważ translacja tekstury jest dokonywana w oparciu o wartości procentowe, będziesz musiał wykonać trochę obliczeń. Ze względu na to, iż w rzędzie umieszczono tylko 4 obrazki, obliczenia te będą dość proste.

Oś x arkusza sprite'ów ma zakres od 0 do 1. Kiedy podzielisz ten zakres na 4 części, każdy z obrazków zajmie 0,25 osi x . Dlatego też aby przesunąć arkusz stylów na czwarty obrazek w linii, musisz dokonać translacji o 0,75. (Pierwszy z obrazków zajmuje wartości od 0 do 0,24, drugi od 0,25 do 0,49, trzeci od 0,5 do 0,74, a czwarty od 0,75 do 1).

```

package com.proandroidgames;

import javax.microedition.khronos.egl.EGLConfig;
import javax.microedition.khronos.opengles.GL10;

import android.opengl.GLSurfaceView.Renderer;

public class SFGGameRenderer implements Renderer {
    ...

    private void movePlayer1(GL10 gl) {
        switch (SFEngine.playerFlightAction) {
            case SFEngine.PLAYER_BANK_LEFT_1:
                gl.glMatrixMode(GL10.GL_MODELVIEW);
                gl.glLoadIdentity();
                gl.glPushMatrix();
                gl.glScalef(.25f, .25f, 1f);
                SFEngine.playerBankPosX -= SFEngine.PLAYER_BANK_SPEED;
                gl.glTranslatef(SFEngine.playerBankPosX, 0f, 0f);
                gl.glMatrixMode(GL10.GL_TEXTURE);
                gl.glLoadIdentity();
                gl.glTranslatef(0.75f, 0.0f, 0.0f);

                break;
            ...
        }
    }
    ...
}

```

Ostatnim krokiem, który musisz uczynić, zanim zlecisz narysowanie statku, jest zwiększenie licznika `goodGuyBankFrames`, byś mógł rozpocząć wyłapywanie momentów, w których należy zmienić ramkę na kolejną z arkusza `sprite`’ów.

```

package com.proandroidgames;

import javax.microedition.khronos.egl.EGLConfig;
import javax.microedition.khronos.opengles.GL10;

import android.opengl.GLSurfaceView.Renderer;

public class SFGGameRenderer implements Renderer {
    ...

    private void movePlayer1(GL10 gl) {
        switch (SFEngine.playerFlightAction) {
            case SFEngine.PLAYER_BANK_LEFT_1:
                gl.glMatrixMode(GL10.GL_MODELVIEW);
                gl.glLoadIdentity();
                gl.glPushMatrix();
                gl.glScalef(.25f, .25f, 1f);
                SFEngine.playerBankPosX -= SFEngine.PLAYER_BANK_SPEED;
                gl.glTranslatef(SFEngine.playerBankPosX, 0f, 0f);
                gl.glMatrixMode(GL10.GL_TEXTURE);
                gl.glLoadIdentity();
                gl.glTranslatef(0.75f, 0.0f, 0.0f);
                goodGuyBankFrames += 1;

```

```

        break;
    ...
}
}
...
}

```

Rozwiązanie to ma jednak jedną istotną wadę. Gracz może teraz przesuwać postać w lewo wzdłuż osi x , co spowoduje, że sprite statku zmieni się na pierwszy ze sprite'ów animacji przechylu na lewo. Problem w tym, że kod w przedstawionej powyżej formie pozwala przesuwać postać gracza w lewo w nieskończoność. Musisz owinać blok kodu poruszający postacią w instrukcję `if...else`, sprawdzając, czy postać nie osiągnęła pozycji 0 na osi x . Jeśli postać znajduje się na pozycji 0, czyli przy lewej krawędzi ekranu, należy wstrzymać ruch i przywrócić animację do domyślnego sprite'a.

```

package com.proandroidgames;

import javax.microedition.khronos.egl.EGLConfig;
import javax.microedition.khronos.opengl.GL10;

import android.opengl.GLSurfaceView.Renderer;

public class SFGGameRenderer implements Renderer {
    ...

    private void movePlayer1(GL10 gl) {
        switch (SFEngine.playerFlightAction) {
            case SFEngine.PLAYER_BANK_LEFT_1:
                gl.glMatrixMode(GL10.GL_MODELVIEW);
                gl.glLoadIdentity();
                gl.glPushMatrix();
                gl.glScalef(.25f, .25f, 1f);
                if (SFEngine.playerBankPosX > 0) {
                    SFEngine.playerBankPosX -= SFEngine.PLAYER_BANK_SPEED;
                    gl.glTranslatef(SFEngine.playerBankPosX, 0f, 0f);
                    gl.glMatrixMode(GL10.GL_TEXTURE);
                    gl.glLoadIdentity();
                    gl.glTranslatef(0.75f, 0.0f, 0.0f);
                    goodGuyBankFrames += 1;
                } else {
                    gl.glTranslatef(SFEngine.playerBankPosX, 0f, 0f);
                    gl.glMatrixMode(GL10.GL_TEXTURE);
                    gl.glLoadIdentity();
                    gl.glTranslatef(0.0f, 0.0f, 0.0f);
                }

                break;
            ...
        }
    }
    ...
}

```

Możesz teraz zlecić narysowanie postaci poprzez wywołanie metody `draw()`, a następnie odłożyć macierz z powrotem na stos. Ten etap procesu powinien być taki sam jak w przypadku obu warstw tła. Co więcej, etap ten będzie się dość często pojawiać w prawie wszystkich operacjach OpenGL w tej grze.

```

package com.proandroidgames;

import javax.microedition.khronos.egl.EGLConfig;

```

```

import javax.microedition.khronos.opengles.GL10;

import android.opengl.GLSurfaceView.Renderer;

public class SGameRenderer implements Renderer {
    ...

    private void movePlayer1(GL10 gl) {
        switch (SFEngine.playerFlightAction) {
            case SFEngine.PLAYER_BANK_LEFT_1:
                gl.glMatrixMode(GL10.GL_MODELVIEW);
                gl.glLoadIdentity();
                gl.glPushMatrix();
                gl.glScalef(.25f, .25f, 1f);
                if (SFEngine.playerBankPosX > 0) {
                    SFEngine.playerBankPosX -= SFEngine.PLAYER_BANK_SPEED;
                    gl.glTranslatef(SFEngine.playerBankPosX, 0f, 0f);
                    gl.glMatrixMode(GL10.GL_TEXTURE);
                    gl.glLoadIdentity();
                    gl.glTranslatef(0.75f, 0.0f, 0.0f);
                    goodGuyBankFrames += 1;
                } else {
                    gl.glTranslatef(SFEngine.playerBankPosX, 0f, 0f);
                    gl.glMatrixMode(GL10.GL_TEXTURE);
                    gl.glLoadIdentity();
                    gl.glTranslatef(0.0f, 0.0f, 0.0f);
                }
                player1.draw(gl);
                gl.glPopMatrix();
                gl.glLoadIdentity();

                break;
            ...
        }
    }
    ...
}

```

Masz już obsługowany przypadek, w którym kiedy gracz porusza się w lewo, wierzchołki są przesuwane w lewo wzdłuż osi *x*, dopóki nie osiągną zera. Ponadto tekstura rozpoczyna od domyślnego sprite'a (rzut z góry), a kiedy gracz porusza się w lewo, sprite jest podmieniany na pierwszą klatkę animacji przechyłu w lewo.

Wczytywanie drugiej ramki animacji

Kiedy gracz przesunie się wystarczająco daleko w lewo, musisz wyświetlić drugą klatkę animacji przechyłu na lewo. Patrząc na arkusz sprite'ów widoczny na rysunku 5.1, można zauważyć, że druga ramka tej animacji jest pierwszym obrazkiem w drugim rzędzie. Łatwo będzie się więc do niej dostać przy pomocy metody `glTranslatef()`. Problem jednak w tym, skąd mamy wiedzieć, kiedy zmienić wyświetlaną ramkę animacji.

We wcześniejszych częściach tego rozdziału stworzyłeś w klasie `SFEngine` stałą `PLAYER_FRAMES_BETWEEN_ANI` i nadałeś jej wartość 9. Stała ta implikuje, że będziesz przeliczać kolejne klatki animacji postaci co dziewięć klatek animacji gry (tj. 9 iteracji pętli gry). Stworzyłeś także zmienną o nazwie `goodGuyBankFrames`, zwiększaną o 1 za każdym razem, gdy postać gracza jest rysowana.

Porównaj aktualną wartość zmiennej `goodGuyBankFrames` ze stałą `PLAYER_FRAMES_BETWEEN_ANI`. Jeśli wartość `goodGuyBankFrames` jest mniejsza, narysuj pierwszą klatkę animacji. Jeżeli jest ona większa od wartości `PLAYER_FRAMES_BETWEEN_ANI` bądź jej równa, narysuj drugą klatkę animacji. Poniżej znajdziesz fragment kodu przedstawiający, jak powinna wyglądać Twoja instrukcja `if...else`.

```
package com.proandroidgames;

import javax.microedition.khronos.egl.EGLConfig;
import javax.microedition.khronos.opengles.GL10;

import android.opengl.GLSurfaceView.Renderer;

public class SFGameRenderer implements Renderer {
    ...

    private void movePlayer1(GL10 gl) {
        switch (SFEngine.playerFlightAction) {
            case SFEngine.PLAYER_BANK_LEFT_1:
                gl.glMatrixMode(GL10.GL_MODELVIEW);
                gl.glLoadIdentity();
                gl.glPushMatrix();
                gl.glScalef(.25f, .25f, 1f);
                if (goodGuyBankFrames < SFEngine.PLAYER_FRAMES_BETWEEN_ANI &&
                    SFEngine.playerBankPosX > 0) {
                    SFEngine.playerBankPosX -= SFEngine.PLAYER_BANK_SPEED;
                    gl.glTranslatef(SFEngine.playerBankPosX, 0f, 0f);
                    gl.glMatrixMode(GL10.GL_TEXTURE);
                    gl.glLoadIdentity();
                    gl.glTranslatef(0.75f, 0.0f, 0.0f);
                    goodGuyBankFrames += 1;
                } else if (goodGuyBankFrames >=
                    SFEngine.PLAYER_FRAMES_BETWEEN_ANI
                    && SFEngine.playerBankPosX > 0) {
                    SFEngine.playerBankPosX -= SFEngine.PLAYER_BANK_SPEED;
                } else {
                    gl.glTranslatef(SFEngine.playerBankPosX, 0f, 0f);
                    gl.glMatrixMode(GL10.GL_TEXTURE);
                    gl.glLoadIdentity();
                    gl.glTranslatef(0.0f, 0.0f, 0.0f);
                }
                player1.draw(gl);
                gl.glPopMatrix();
                gl.glLoadIdentity();

                break;
            ...
        }
    }
    ...
}
```

W warunku instrukcji `if...else` sprawdzasz, czy wartość zmiennej `goodGuyBankFrames` jest większa od stałej `PLAYER_FRAMES_BETWEEN_ANI`, co wskazywałoby na to, że należy przerzucić kolejną klatkę animacji przechylu w lewo. Napiszmy teraz fragment kodu zmieniający klatki animacji.

Na rysunku 5.1 druga klatka animacji przechylu w lewo znajduje się na pierwszej pozycji w drugim rzędzie. Oznacza to, że lewy górny róg tego sprite'a ma współrzędną 0 na osi *x* (najdalszą lewą) oraz .25 (jedną czwartą) na osi *y*. Wystarczy, że użyjesz metody `glTranslatef()`, aby przesunąć teksturę na odpowiednią pozycję.

■ **Uwaga:** Zanim przesuńiesz teksturę, musisz przejść do trybu macierzy tekstury.

```
package com.proandroidgames;

import javax.microedition.khronos.egl.EGLConfig;
import javax.microedition.khronos.opengles.GL10;

import android.opengl.GLSurfaceView.Renderer;

public class SFGGameRenderer implements Renderer {
    ...

    private void movePlayer1(GL10 gl) {
        switch (SFEngine.playerFlightAction) {
            case SFEngine.PLAYER_BANK_LEFT_1:
                gl.glMatrixMode(GL10.GL_MODELVIEW);
                gl.glLoadIdentity();
                gl.glPushMatrix();
                gl.glScalef(.25f, .25f, 1f);
                if (goodGuyBankFrames < SFEngine.PLAYER_FRAMES_BETWEEN_ANI &&
                    SFEngine.playerBankPosX > 0) {
                    SFEngine.playerBankPosX -= SFEngine.PLAYER_BANK_SPEED;
                    gl.glTranslatef(SFEngine.playerBankPosX, 0f, 0f);
                    gl.glMatrixMode(GL10.GL_TEXTURE);
                    gl.glLoadIdentity();
                    gl.glTranslatef(0.75f, 0.0f, 0.0f);
                    goodGuyBankFrames += 1;
                } else if (goodGuyBankFrames >=
                    SFEngine.PLAYER_FRAMES_BETWEEN_ANI
                    && SFEngine.playerBankPosX > 0) {
                    SFEngine.playerBankPosX -= SFEngine.PLAYER_BANK_SPEED;
                    gl.glTranslatef(SFEngine.playerBankPosX, 0f, 0f);
                    gl.glMatrixMode(GL10.GL_TEXTURE);
                    gl.glLoadIdentity();
                    gl.glTranslatef(0.0f, 0.25f, 0.0f);
                } else {
                    gl.glTranslatef(SFEngine.playerBankPosX, 0f, 0f);
                    gl.glMatrixMode(GL10.GL_TEXTURE);
                    gl.glLoadIdentity();
                    gl.glTranslatef(0.0f, 0.0f, 0.0f);
                    goodGuyBankFrames = 0;
                }
                player1.draw(gl);
                gl.glPopMatrix();
                gl.glLoadIdentity();

                break;
            ...
        }
    }
    ...
}
```

Obsługa w instrukcji switch przypadku ruchu postaci w lewo oraz implementacja dwuklatkowej animacji sprite'a są już kompletne.

Przesuwanie postaci w prawo

Ostatnim wyrażeniem case, które musisz uzupełnić, zanim ukończysz metodę `movePlayer1()`, jest wyrażenie obsługujące akcję `PLAYER_BANK_RIGHT_1`. Przypadek ten jest wywoływany, kiedy gracz chce przesunąć postać na ekranie w prawo, czyli w kierunku dodatnim na osi x .

Szablon kodu obsługującego przypadek będzie wyglądał identycznie jak przypadek ruchu w lewo, będziesz jednak wczytywał inne ramki z arkusza sprite'ów. Po pierwsze ustaw macierz modelu, przeskaluj wierzchołki postaci i stwórz instrukcję warunkową `if...else`, tak jak to uczyniłeś w przypadku `PLAYER_BANK_LEFT_1`.

Wspomniana instrukcja warunkowa będzie się różniła jedną rzeczą od instrukcji użytej w przypadku `PLAYER_BANK_LEFT_1`. Przy ruchu w lewo sprawdzałeś, czy aktualna wartość pozycji wierzchołków na osi x była większa od 0, co wskazywało, że gracz nie znajduje się poza lewą krawędzią ekranu. W przypadku `PLAYER_BANK_RIGHT_1` będziesz musiał sprawdzać, czy postać nie osiągnęła już skrajnej pozycji po prawej stronie ekranu.

Przy ustawieniach domyślnych oś x zaczyna się w 0 i kończy w 1. Aby zmniejszyć postać gracza, przeskalowałeś jednak oś x razy `.25`. Oznacza to, że oś x rozpoczyna się teraz w 0, a kończy w 4. Wynikałoby z tego, iż musisz sprawdzić, czy postać gracza nie przemieściła się o więcej niż 4 jednostki w prawo.

Nie do końca...

OpenGL śledzi lewy górny wierzchołek Twojej figury. Gdybyś więc sprawdzał, czy wartość na osi x osiągnęła 4, postać byłaby poza prawą krawędzią ekranu już w momencie spełnienia tego warunku. Musisz wziąć pod uwagę szerokość postaci. Skrajne wierzchołki (lewy i prawy) są odległe od siebie o 1 jednostkę. Sprawdzanie, czy postać nie przekroczyła wartości 3 na osi x , sprawi, że dla gracza postać zawsze będzie widoczna na ekranie.

```
package com.proandroidgames;

import javax.microedition.khronos.egl.EGLConfig;
import javax.microedition.khronos.opengles.GL10;

import android.opengl.GLSurfaceView.Renderer;

public class SFGameRenderer implements Renderer {
    ...

    private void movePlayer1(GL10 gl) {
        switch (SFEngine.playerFlightAction) {
            ...
            case SFEngine.PLAYER_BANK_RIGHT_1:
                gl.glMatrixMode(GL10.GL_MODELVIEW);
                gl.glLoadIdentity();
                gl.glPushMatrix();
                gl.glScalef(.25f, .25f, 1f);
                if (goodGuyBankFrames < SFEngine.PLAYER_FRAMES_BETWEEN_ANI &&
                    SFEngine.playerBankPosX < 3) {

                    } else if (goodGuyBankFrames >=
                        SFEngine.PLAYER_FRAMES_BETWEEN_ANI
                        && SFEngine.playerBankPosX < 3) {

                    } else {
                        gl.glTranslatef(SFEngine.playerBankPosX, 0f, 0f);
                        gl.glMatrixMode(GL10.GL_TEXTURE);
                        gl.glLoadIdentity();
                        gl.glTranslatef(0.0f, 0.0f, 0.0f);
                        goodGuyBankFrames = 0;
                    }
                }
            }
        }
    }
}
```



```

    }
    player1.draw(gl);
    gl.glPopMatrix();
    gl.glLoadIdentity();

    break;
...
}
}
...
}

```

Ten początkowy blok kodu umieszczony wewnątrz sekcji case wartości `PLAYER_BANK_RIGHT_1` jest praktycznie identyczny z kodem dla przypadku `PLAYER_BANK_LEFT_1`. Dostosowujesz macierz modelu, sprawdzasz pozycję postaci na osi *x* oraz liczbę wykonanych przebiegów pętli gry, by się dowiedzieć, którą klatkę animacji sprite'a wyświetlić.

Możesz teraz wyświetlić w odpowiednich miejscach pierwszą i drugą klatkę animacji przechyłu w prawo.

Wczytywanie animacji przechyłu w prawo

Pierwsza klatka animacji, którą powinieneś wyświetlić, kiedy gracz przechyła swój statek w prawo, znajduje się na drugiej pozycji w pierwszym rzędzie (zgodnie z rysunkiem 5.1). By wyświetlić tę klatkę, musisz więc dokonać translacji macierzy tekstur o 0,25 na osi *x* i 0 na osi *y*.

```

package com.proandroidgames;

import javax.microedition.khronos.egl.EGLConfig;
import javax.microedition.khronos.opengles.GL10;

import android.opengl.GLSurfaceView.Renderer;

public class SFGameRenderer implements Renderer {
    ...

    private void movePlayer1(GL10 gl) {
        switch (SFEngine.playerFlightAction) {
            ...
            case SFEngine.PLAYER_BANK_RIGHT_1:
                gl.glMatrixMode(GL10.GL_MODELVIEW);
                gl.glLoadIdentity();
                gl.glPushMatrix();
                gl.glScalef(.25f, .25f, 1f);
                if (goodGuyBankFrames < SFEngine.PLAYER_FRAMES_BETWEEN_ANI &&
                    SFEngine.playerBankPosX < 3) {
                    SFEngine.playerBankPosX += SFEngine.PLAYER_BANK_SPEED;
                    gl.glTranslatef(SFEngine.playerBankPosX, 0f, 0f);
                    gl.glMatrixMode(GL10.GL_TEXTURE);
                    gl.glLoadIdentity();
                    gl.glTranslatef(0.25f, 0.0f, 0.0f);
                    goodGuyBankFrames += 1;
                } else if (goodGuyBankFrames >=
                    SFEngine.PLAYER_FRAMES_BETWEEN_ANI
                    && SFEngine.playerBankPosX < 3) {

                } else {

```

```

        gl.glTranslatef(SFEngine.playerBankPosX, 0f, 0f);
        gl.glMatrixMode(GL10.GL_TEXTURE);
        gl.glLoadIdentity();
        gl.glTranslatef(0.0f, 0.0f, 0.0f);
        goodGuyBankFrames = 0;
    }
    player1.draw(gl);
    gl.glPopMatrix();
    gl.glLoadIdentity();

    break;
...
}
...
}

```

Zauważ, że w dodanym fragmencie kodu wartość stałej `PLAYER_BANK_SPEED` jest dodawana do aktualnej pozycji postaci gracza, a nie od niej odejmowana. To kluczowy element odróżniający przesuwanie wierzchołków w prawo wzdłuż osi *x* od przesuwania ich w lewo.

Używając ponownie tego kodu do wyświetlenia drugiej ramki animacji przechyłu w prawo, musisz dokonać translacji tekstury o 0,5 wzdłuż osi *x*.

```

package com.proandroidgames;

import javax.microedition.khronos.egl.EGLConfig;
import javax.microedition.khronos.opengl.GL10;

import android.opengl.GLSurfaceView.Renderer;

public class SFGameRenderer implements Renderer {
    ...

    private void movePlayer1(GL10 gl) {
        switch (SFEngine.playerFlightAction) {
            ...
            case SFEngine.PLAYER_BANK_RIGHT_1:
                gl.glMatrixMode(GL10.GL_MODELVIEW);
                gl.glLoadIdentity();
                gl.glPushMatrix();
                gl.glScalef(.25f, .25f, 1f);
                if (goodGuyBankFrames < SFEngine.PLAYER_FRAMES_BETWEEN_ANI &&
                    SFEngine.playerBankPosX < 3) {
                    SFEngine.playerBankPosX += SFEngine.PLAYER_BANK_SPEED;
                    gl.glTranslatef(SFEngine.playerBankPosX, 0f, 0f);
                    gl.glMatrixMode(GL10.GL_TEXTURE);
                    gl.glLoadIdentity();
                    gl.glTranslatef(0.25f, 0.0f, 0.0f);
                    goodGuyBankFrames += 1;
                } else if (goodGuyBankFrames >=
                    SFEngine.PLAYER_FRAMES_BETWEEN_ANI
                    && SFEngine.playerBankPosX < 3) {
                    SFEngine.playerBankPosX += SFEngine.PLAYER_BANK_SPEED;
                    gl.glTranslatef(SFEngine.playerBankPosX, 0f, 0f);
                    gl.glMatrixMode(GL10.GL_TEXTURE);
                    gl.glLoadIdentity();
                    gl.glTranslatef(0.50f, 0.0f, 0.0f);
                } else {

```

```

        gl.glTranslatef(SFEngine.playerBankPosX, 0f, 0f);
        gl.glMatrixMode(GL10.GL_TEXTURE);
        gl.glLoadIdentity();
        gl.glTranslatef(0.0f, 0.0f, 0.0f);
        goodGuyBankFrames = 0;
    }
    player1.draw(gl);
    gl.glPopMatrix();
    gl.glLoadIdentity();

    break;
...
}
...
}

```

Twoja metoda `movePlayer1()` jest już gotowa. Postać gracza będzie się teraz z powodzeniem poruszała w lewo i w prawo w wyniku wywołania odpowiedniej akcji. Wszystko, co musisz teraz zrobić, to wywołać metodę `movePlayer1()` z wnętrza głównej pętli gry i stworzyć mechanizm pozwalający graczowi sterować postacią.

```

package com.proandroidgames;

import javax.microedition.khronos.egl.EGLConfig;
import javax.microedition.khronos.opengles.GL10;

import android.opengl.GLSurfaceView.Renderer;

public class SFGGameRenderer implements Renderer {
    ...

    @Override
    public void onDrawFrame(GL10 gl) {
        try {
            Thread.sleep(SFEngine.GAME_THREAD_FPS_SLEEP);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        gl.glClear(GL10.GL_COLOR_BUFFER_BIT | GL10.GL_DEPTH_BUFFER_BIT);

        scrollBackground1(gl);
        scrollBackground2(gl);
        movePlayer1(gl);

        // Pozostałe metody rysujące elementy gry będą wywoływane tutaj

        gl.glEnable(GL10.GL_BLEND);
        gl.glBlendFunc(GL10.GL_ONE, GL10.GL_ONE_MINUS_SRC_ALPHA);
    }
    ...
}

```

Zapisz i zamknij plik `SFGGameRenderer.java`.

W kolejnym podrozdziale nauczysz się nasłuchiwanie na zdarzenia `TouchEvent` wywoływane przez ekran dotykowy urządzenia z systemem Android. Następnie przetłumaczysz to zdarzenie na odpowiednią akcję gracza, poruszając w ten sposób wyświetlaną na ekranie postać w lewo lub w prawo.

Poruszanie postacią gracza przy pomocy zdarzenia dotykowego

Stworzyłeś już metodę oraz wywołania potrzebne do przesuwania postaci gracza po ekranie. W tej chwili jednak gracz nie może się w żaden sposób komunikować z grą i wskazywać pętli gry, by odwoływała się do mechanizmów poruszających postacią gracza.

W tym podrozdziale stworzysz prosty listener nasłuchujący na zdarzenia dotykowe, który będzie wykrywał, czy gracz dotknął prawej, czy lewej strony ekranu. Listener znajdzie się w czynności, w której znajduje się pętla gry — w tym przypadku będzie to klasa `SFGame`.

Otwórz plik `SFGame.java` i zadeklaruj przeciążenie metody `onTouchEvent()`.

```
package com.proandroidgames;

import android.app.Activity;
import android.os.Bundle;
import android.view.MotionEvent;

public class SFGame extends Activity {
    ...
    @Override
    public boolean onTouchEvent(MotionEvent event) {
        return false;
    }
}
```

Metoda `onTouchEvent()` to standardowy dla platformy Android listener zdarzeń, który będzie nasłuchiwał na dowolne zdarzenie dotykowe, które zajdzie wewnątrz czynności. Ponieważ Twoja gra uruchamiana jest z wnętrza czynności `SFGame`, to właśnie wewnątrz tej czynności musisz nasłuchiwać na zdarzenia dotykowe.

■ **Wskazówka:** Nie myl czynności gry z pętlą gry. Pętla gry znajduje się w klasie `SFGameRenderer`; klasa typu `Activity` uruchamiająca grę to `SFGame`.

Listener `onTouchEvent()` zostanie odpalony jedynie wtedy, gdy użytkownik urządzenia dotknie jego ekranu, przejdzie po nim, przeciągnie go lub puści. W tej grze będziesz się zajmował tylko dotknięciem bądź puszczeniem ekranu oraz tym, po której stronie ekranu te zdarzenia miały miejsce. Aby pomóc nam to określić, Android wysła do listenera `onTouchEvent()` widok `MotionEvent`, zawierający wszystkie informacje potrzebne do zdefiniowania rodzaju zdarzenia dotykowego, które wywołało listenera, oraz wskazania, w którym miejscu ekranu to zdarzenie miało miejsce.

Przetwarzanie zdarzenia `MotionEvent`

Twoim pierwszym zadaniem, jeśli chodzi o tworzenie listenera `onTouchEvent()`, jest pobranie współrzędnych x i y zdarzenia, abyś mógł określić, czy zaszło ono po lewej, czy po prawej stronie ekranu urządzenia. Przekazywany do listenera `onTouchEvent()` obiekt `MotionEvent` posiada metody `getX()` i `getY()`, których możesz użyć do zdefiniowania współrzędnych x i y zdarzenia dotykowego.

■ **Uwaga:** Współrzędne x i y , którymi będziesz się zajmował w listenerze `onTouchEvent()`, są współrzędnymi ekranu, a nie środowiska OpenGL.

```

package com.proandroidgames;

import android.app.Activity;
import android.os.Bundle;
import android.view.MotionEvent;

public class SFGame extends Activity {
    ...
    @Override
    public boolean onTouchEvent(MotionEvent event) {
        float x = event.getX();
        float y = event.getY();

        return false;
    }
}

```

Aby stwierdzić, w którym miejscu ekranu znajdują się dane współrzędne, potrzebny Ci będzie dostęp do jego wymiarów. Możesz je uzyskać przy pomocy klasy `Display`. Utwórz teraz nową zmienną typu `display` w klasie `SFEngine`.

```

package com.proandroidgames;

import android.content.Context;
import android.content.Intent;
import android.view.Display;
import android.view.View;

public class SFEngine {
    /* Stałe używane w grze */
    public static final int GAME_THREAD_DELAY = 4000;
    public static final int MENU_BUTTON_ALPHA = 0;
    public static final boolean HAPTIC_BUTTON_FEEDBACK = true;
    public static final int SPLASH_SCREEN_MUSIC = R.raw.warfieldedit;
    public static final int R_VOLUME = 100;
    public static final int L_VOLUME = 100;
    public static final boolean LOOP_BACKGROUND_MUSIC = true;
    public static final int GAME_THREAD_FPS_SLEEP = (1000 / 60);
    public static Context context;
    public static Thread musicThread;
public static Display display;
    public static final int BACKGROUND_LAYER_ONE = R.drawable.backgroundstars;
    public static float SCROLL_BACKGROUND_1 = .002f;
    public static float SCROLL_BACKGROUND_2 = .007f;
    public static final int BACKGROUND_LAYER_TWO = R.drawable.debris;
    public static int playerFlightAction = 0;
    public static final int PLAYER_SHIP = R.drawable.good_sprite;
    public static final int PLAYER_BANK_LEFT_1 = 1;
    public static final int PLAYER_RELEASE = 3;
    public static final int PLAYER_BANK_RIGHT_1 = 4;
    public static final int PLAYER_FRAMES_BETWEEN_ANI = 9;
    public static final float PLAYER_BANK_SPEED = .1f;
    public static float playerBankPosX = 1.75f;
    ...
}

```

Do zmiennej tej przypisz już na początku gry, w metodzie `onCreate()` czynności `StarfighterService`, pobrany ze środowiska Android obiekt reprezentujący ekran urządzenia:

```
package com.proandroidgames;

import android.app.Activity;
import android.content.Context;
import android.content.Intent;
import android.os.Bundle;
import android.os.Handler;
import android.view.WindowManager;

public class StarfighterActivity extends Activity {
    /** Wywoływane podczas tworzenia czynności. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        SFEngine.display = ((WindowManager) getSystemService(Context.WINDOW_SERVICE))
            .getDefaultDisplay();

        super.onCreate(savedInstanceState);
        ...
    }
}
```

Możesz teraz określić interaktywny obszar ekranu. Nie chcesz reagować na zdarzenia dotykowe w dowolnym miejscu ekranu, określisz więc obszar w dolnej części ekranu, który będzie na nie reagował. Interaktywny obszar będzie się znajdował na samym dole ekranu, aby gracze mogli go dotykać kciukami, trzymając urządzenie w ręce.

Skoro obszar interaktywny zajmuje dolną ćwiartkę ekranu urządzenia, skonfigurujesz ten obszar jako obszar, w którym będziesz reagował na zdarzenia dotykowe.

```
package com.proandroidgames;

import android.app.Activity;
import android.os.Bundle;
import android.view.MotionEvent;

public class SFGGame extends Activity {
    ...
    @Override
    public boolean onTouchEvent(MotionEvent event) {
        float x = event.getX();
        float y = event.getY();
        int height = SFEngine.display.getHeight() / 4;
        int playableArea = SFEngine.display.getHeight() - height;

        return false;
    }
}
```

Masz już współrzędne zdarzenia dotykowego oraz obszar, w którym będziesz reagował na tego typu zdarzenia. Użyj prostej instrukcji warunkowej `if`, by określić, czy powinieneś zareagować na zgłoszone zdarzenie.

```
package com.proandroidgames;

import android.app.Activity;
import android.os.Bundle;
```

```
import android.view.MotionEvent;

public class SFGame extends Activity {
    ...
    @Override
    public boolean onTouchEvent(MotionEvent event) {
        float x = event.getX();
        float y = event.getY();
        int height = SFEngine.display.getHeight() / 4;
        int playableArea = SFEngine.display.getHeight() - height;
        if (y > playableArea) {

        }
        return false;
    }
}
```

Zdarzenie `MotionEvent` posiada bardzo użyteczną metodę `getAction()`, która zwraca wykryty typ wykonanej na ekranie akcji. Na potrzeby tej gry będziesz się zajmował jedynie akcjami `ACTION_UP` oraz `ACTION_DOWN`. Reprezentują one chwile, w których palec gracza rozpoczął dotykanie ekranu (`ACTION_DOWN`), a następnie oderwał się od ekranu (`ACTION_UP`).

Przechwytywanie akcji `ACTION_UP` i `ACTION_DOWN`

Stwórz prostą instrukcję `switch` obejmującą akcje `ACTION_UP` oraz `ACTION_DOWN`. Pozostaw instrukcję bez domyślnego przypadku (`default`), ponieważ chcesz reagować jedynie na te dwie specyficzne akcje.

```
package com.proandroidgames;

import android.app.Activity;
import android.os.Bundle;
import android.view.MotionEvent;

public class SFGame extends Activity {
    ...
    @Override
    public boolean onTouchEvent(MotionEvent event) {
        float x = event.getX();
        float y = event.getY();
        int height = SFEngine.display.getHeight() / 4;
        int playableArea = SFEngine.display.getHeight() - height;
        if (y > playableArea) {
            switch (event.getAction()) {
                case MotionEvent.ACTION_DOWN:

                    break;
                case MotionEvent.ACTION_UP:

                    break;
            }
        }
        return false;
    }
}
```

Wcześniej w tym rozdziale napisałeś kod, który pozwalał Ci poruszać postać gracza na ekranie. Kod ten reagował na trzy utworzone przez Ciebie reprezentujące akcje stałe: `PLAYER_BANK_LEFT_1`, `PLAYER_BANK_RIGHT_1` i `PLAYER_RELEASE`. Akcje te będą ustawiane dla odpowiednich przypadków w metodzie `onTouchEvent()`.

Rozpocznijmy od akcji `PLAYER_RELEASE`. Przypadek ten będzie ustawiany, gdy gracz oderwie palec od ekranu, odpalając tym samym zdarzenie `ACTION_UP`.

```
package com.proandroidgames;

import android.app.Activity;
import android.os.Bundle;
import android.view.MotionEvent;

public class SFGame extends Activity {
    ...
    @Override
    public boolean onTouchEvent(MotionEvent event) {
        float x = event.getX();
        float y = event.getY();
        int height = SFEngine.display.getHeight() / 4;
        int playableArea = SFEngine.display.getHeight() - height;
        if (y > playableArea) {
            switch (event.getAction()) {
                case MotionEvent.ACTION_DOWN:

                    break;
                case MotionEvent.ACTION_UP:
                    SFEngine.playerFlightAction = SFEngine.PLAYER_RELEASE;
                    break;
            }
        }
        return false;
    }
}
```

Na koniec ustawisz akcje `PLAYER_BANK_LEFT_1` i `PLAYER_BANK_RIGHT_1`. Aby to zrobić, znów musisz określić, czy gracz dotknął prawej, czy lewej strony ekranu. Można to łatwo sprawdzić, porównując wartość metody `getX()` klasy `MotionEvent` ze środkową wartością osi `x`. Jeśli wartość ta jest mniejsza od środka osi, akcja została wywołana po lewej stronie ekranu; jeżeli zaś wartość ta jest większa od środka osi, zdarzenie nastąpiło po stronie prawej.

```
package com.proandroidgames;

import android.app.Activity;
import android.os.Bundle;
import android.view.MotionEvent;

public class SFGame extends Activity {
    ...
    @Override
    public boolean onTouchEvent(MotionEvent event) {
        float x = event.getX();
        float y = event.getY();
        int height = SFEngine.display.getHeight() / 4;
        int playableArea = SFEngine.display.getHeight() - height;
        if (y > playableArea) {
            switch (event.getAction()) {
                case MotionEvent.ACTION_DOWN:
```



```

        if (x < SFEngine.display.getWidth() / 2) {
            SFEngine.playerFlightAction = SFEngine.PLAYER_BANK_LEFT_1;
        } else {
            SFEngine.playerFlightAction = SFEngine.PLAYER_BANK_RIGHT_1;
        }
        break;
    case MotionEvent.ACTION_UP:
        SFEngine.playerFlightAction = SFEngine.PLAYER_RELEASE;
        break;
    }
}
return false;
}
}

```

Zapisz i zamknij plik *SFGame.java*. Zakończyłeś właśnie implementację interfejsu użytkownika swojej gry. Gracz może teraz dotknąć prawej lub lewej strony ekranu, by przesunąć swoją postać odpowiednio w prawo albo w lewo.

W końcowym podrozdziale przyjrzymy się ponownie głównemu wątkowi gry i obliczaniu liczby klatek na sekundę.

Dostosowanie opóźnienia FPS

W poprzednim rozdziale dodałeś do pętli opóźnienie, aby wymusić wykonanie jej co najwyżej 60 razy na sekundę (tj. wyświetlanie gry z szybkością 60 klatek na sekundę — 60 FPS). Taka szybkość działania jest najbardziej pożądana przez twórców gier. Jak zapewne zdążyłeś już zauważyć, szybkość ta nie jest jednak zawsze możliwa do osiągnięcia.

Im więcej funkcji wykonuje Twoja pętla gry, tym więcej czasu zajmie jej jeden przebieg i tym wolniej gra będzie działać. Oznacza to, że stworzone przez Ciebie opóźnienie musi zostać dostosowane lub całkowicie wyłączone, w zależności od tego, jak wolno działa gra.

Dla porównania, działająca w obecnym kształcie gra, z dwoma tłami i postacią gracza, osiąga na moim emulatorze pod systemem Windows około 10 klatek na sekundę, około 35 klatek na sekundę na telefonie Droid X i około 43 klatek na sekundę na tablecie Motorola Xoom.

Jednym z problemów jest fakt, iż wielkość stosowanego opóźnienia jest stała i nie zależy od szybkości działania gry. Powinieneś dostosować opóźnienie wątku gry względem ilości czasu potrzebnego na wykonanie jednego przebiegu pętli. Poniższy kod określi, ile czasu zajmuje jedno przejście pętli gry, i odejmie ten czas od czasu opóźnienia. Jeśli przebieg pętli zajmuje więcej czasu, niż wynosi opóźnienie, opóźnienie jest wyłączane.

```

package com.proandroidgames;

import javax.microedition.khronos.egl.EGLConfig;
import javax.microedition.khronos.opengles.GL10;

import android.opengl.GLSurfaceView.Renderer;

public class SFGameRenderer implements Renderer {

    private SFBackground background = new SFBackground();
    private SFBackground background2 = new SFBackground();
    private SFGoodGuy player1 = new SFGoodGuy();
    private int goodGuyBankFrames = 0;

    private long loopStart = 0;
    private long loopEnd = 0;

```

```

private long loopRunTime = 0;

private float bgScroll1;
private float bgScroll2;

@Override
public void onDrawFrame(GL10 gl) {
    loopStart = System.currentTimeMillis();
    try {
        if (loopRunTime < SFEngine.GAME_THREAD_FPS_SLEEP) {
            Thread.sleep(SFEngine.GAME_THREAD_FPS_SLEEP - loopRunTime);
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    gl.glClear(GL10.GL_COLOR_BUFFER_BIT | GL10.GL_DEPTH_BUFFER_BIT);

    scrollBackground1(gl);
    scrollBackground2(gl);
    movePlayer1(gl);

    // Pozostałe metody rysujące elementy gry będą wywoływane tutaj

    gl.glEnable(GL10.GL_BLEND);
    gl.glBlendFunc(GL10.GL_ONE, GL10.GL_ONE_MINUS_SRC_ALPHA);
    loopEnd = System.currentTimeMillis();
    loopRunTime = ((loopEnd - loopStart));
}
...
}

```

Skompiluj i uruchom swoją grę. Spróbuj poruszać postacią po ekranie i obserwuj zmiany w animacji.

Podsumowanie

W tym rozdziale poczyniłeś kolejny wielki krok w tworzeniu gry *Star Fighter*. Do swoich osiągnięć możesz teraz dodać następujące umiejętności:

- tworzenie postaci gracza,
- animacja postaci przy pomocy tekstur z arkusza sprite'ów,
- wykrywanie na ekranie urządzenia poleceń dotykowych,
- przesuwanie i animacja postaci w oparciu o wywołane przez gracza zdarzenia dotykowe,
- dostosowywanie liczby klatek na sekundę, by gra działała najszybciej, jak to możliwe.

Skorowidz

A

AI, Artificial Intelligence, 171
akcja
 ACTION_DOWN, 151
 ACTION_UP, 151
 PLAYER_BANK_LEFT_1, 137, 152
 PLAYER_BANK_RIGHT_1, 152
 PLAYER_RELEASE, 135
aktualizowanie przeciwników, 179
Android SDK, 21
animacja
 kroków, 122
 sprite'a, 121, 129
arkusz sprite'ów, 122, 160, 167, 197, 201
atrybut
 layout_height, 52
 layout_width, 52
atrybuty
 czynności, 45
 FrameLayout, 52

B

biblioteka graficzna, 29
blendowanie przezroczystości, 119
błędy aplikacji, 47
bufory OpenGL, 118

C

CA, certificate authority, 230
CAD, Computer-Aided Design, 29
czynności
 gry, 33
 silnika, 38

czynność, activity, 43, 82
 SFMainMenu, 45, 62
 StarfighterActivity, 42, 84
czyszczenie buforów OpenGL, 118

D

detekcja kolizji, 207, 275
dodawanie
 arkusza sprite'ów, 160
 drugiej warstwy, 108
 listenerów, 69
 muzyki, 70
 obrazów, 63
 tekstury, 266
dostawca certyfikatów, CA, 230
dostęp do tekstur, 158
dostosowywanie
 widoku gracza, 275
 wierzchołków, 181

E

Eclipse, 27
edytor tekstowy, 51
efekt rozmycia, 55
ekran powitalny, 41, 48, 66
element
 gry, 37
 silnika, 37
 TextView, 52

F

flaga isDestroyed, 179, 207
 folder drawable-hdpi, 49
 format nine-patch, 48
 FPS, First-Person Shooter, 34
 funkcja
 Destroy(), 36
 Move(), 36
 TestForCollision(), 36
 funkcje
 silnika, 34
 wyświetlające grafikę, 35

G

generator liczb losowych, 163
 gra Blob Hunter, 239, 276
 gra Spy Hunter, 25
 gra Star Fighter, 31, 38
 ekran powitalny, 41
 projekt, 38
 publikacja, 229
 rozszerzanie wersji, 211
 silnik, 38
 szczegóły, 32
 gry 2D, 239
 gry 3D, 239

H

historia gry Star Fighter, 31

I

IDE, Integrated Development Environment, 27
 identyfikator splashScreenImage, 52
 implementacja
 uzbrojenia, 207
 ruchu, 184
 importowanie obrazka, 48
 inicjalizacja
 przeciwników, 175
 uzbrojenia, 202
 instalacja OpenGL ES, 29
 instancja klasy BHWalls, 246
 instrukcja switch...case, 274
 interfejs sterowania, 269
 iteracja pętli głównej, 129

J

języki niskiego poziomu, 22

K

katalog drawable-nodpi, 160, 243
 kierunek ataku, 163, 174

klasa

Activity, 43, 83
 BHCorridor, 257, 261, 264, 267
 BHEngine, 241, 270, 276
 BHGameRenderer, 241, 267
 BHGameView, 241
 BHWalls, 244–246, 255
 BlobhunterActivity, 240, 269–271
 GLSurfaceView, 241
 Handler, 60
 Intent, 60
 MotionEvent, 152
 SFBackground, 108, 244
 SFEnemy, 161, 163
 SFEngine, 242
 SFGame, 83
 SFGameRenderer, 86, 171, 201, 216
 SFGameView, 84
 SFGoodGuy, 123, 127
 SFMainMenu, 42, 45, 77
 SFTextures, 157, 172
 SFWeapon, 200
 StarfighterActivity, 59
 tekstury, 156
 kod specyficzny dla gry, 36
 kojarzenie czynności z projektem, 43
 kolizje, 207, 209
 detekcja, 207, 275
 typy kolizji, 209
 korytarz, 256, 257
 krawędź ekranu, 204
 kreator
 eksportu, 231
 projektu, 39
 krzywa Béziera, 165, 191
 kształt korytarza, 258
 kwadrat, 244

L

listener onClickListener, 69, 119

Ł

łączenie czynności, 54

M

macierz, 112
 OpenGL, 103
 modelu, 112
 tekstur, 103, 105
 magazyn kluczy, 233
 mapowanie
 obrazka, 247
 tekstur, 93, 98, 124
 tekstur 2D, 87
 menu główne, 67, 119

metoda

appyDamage(), 208
 detectCollisions(), 208
 draw(), 99, 125, 263
 drawBackground(), 250, 251
 drawCorridor(), 267, 274
 firePlayerWeapon(), 203–205
 firePlayerWeapons(), 206
 getX(), 152
 glBindTextures(), 158
 glGenTextures(), 157
 glMatrixMode(), 91
 glOrthof(), 91, 248
 glRotatf(), 250
 glScale(), 112
 glTranslate(), 103
 glTranslatef(), 122, 256
 gluLookAt(), 251, 267
 gluPerspective(), 248
 glViewport(), 90
 loadTexture(), 96–100, 110, 127, 156, 262
 moveEnemy(), 178
 movePlayer1(), 132
 onClick(), 69
 onCreate(), 74
 onDraw(), 200
 onDrawFrame(), 87, 115, 195
 onPause(), 85
 onResume(), 85
 onSurfaceChanged(), 87, 90
 onSurfaceCreated(), 87–89, 131, 175, 203
 onTouchEvent(), 152, 270–272
 overridePendingTransition(), 62
 postDelay(), 59, 60
 run(), 59
 scrollBackground1(), 101, 104, 111
 scrollBackground2(), 111, 113
 setContentView(), 54, 84
 SFBackground.loadTexture(), 93
 Thread.sleep(), 118

metody OpenGL, 30

muzyka, 70

N

nachylenie prostej, 185
 nakładanie tekstury, 128
 namierzanie
 pozycji gracza, 182
 statku, 177

O

obiekt ImageButton, 69

obrazek

ekranu powitalnego, 51
 statku, 243

obrazy, 47

obrazy przycisków, 63

obrót

modelu, 275
 w 3D, 246, 250
 widoku, 271

obsługa ruchu, 270

OpenGL, 21, 27

OpenGL ES, 29, 82

OpenGL ES 1.0, 30

OpenGL for Embedded Systems, 29

opóźnienie, 153

orientacja ekranu, 46

P

parametr textureNumber, 158

pętla gry, 116, 131

plik

AndroidManifest.xml, 43, 76, 229
 BHCorridor.java, 264, 277
 BHEngine.java, 242, 270, 276
 BHGameRenderer.java, 241, 273, 279
 BHGameView.java, 241
 BlobhunterActivity.java, 240
 debris.png, 108
 fadein.xml, 56
 fadeout.xml, 56
 R.java, 49
 SFBadGuy.java, 224
 SFEnemy.java, 224, 225
 SFEngine.java, 60, 67, 77, 198, 212
 SFGGame.java, 153
 SFGGameRenderer.java, 130, 216
 SFGGoodGuy.java, 224
 SFMainMenu.java, 62, 67, 77
 SFMusic.java, 72
 SFTextures.java, 215
 SFWeapon.java, 213
 splashscreen.xml, 51
 starfighter.apk, 235
 StarfighterActivity.java, 54, 58
 warfieldedit.ogg, 71

pliki .apk, 234

pliki muzyczne, 70

podpisywanie pliku, 234

poruszanie

postacią gracza, 132, 273
 sprite'a, 121
 statkiem, 165

postać

gracza, 121, 123
 przeciwnika, 160

pozycja gracza, 182

projekt
 blobhunter, 240
 starfighter, 42
 przechwytywanie akcji, 151
 przesuwanie modelu, 275
 przewijana strzelanka, scrolling shooter, 18
 przewijanie, 107, 111
 przewijanie tła, 101
 przezroczystość, 119
 przyciski, 67
 przyśpieszenie statku, 185
 publikacja aplikacji, 230
 punkt namierzania, 190

R

rachunek macierzy, 29
 renderer, 86, 240
 renderer BHGamerRenderer, 268
 renderowanie, 239
 gry, 267
 powierzchni, 91
 tła, 81
 resetowanie macierzy, 113
 ruch po krzywej, 191
 rysowanie postaci, 133

S

silnik gry, 34, 35, 37
 silnik Unreal, 34
 skalowanie obrazów, 47
 specyfikacja ikony aplikacji, 230
 sprawdzanie
 projektu, 231
 manifestu, 232
 sprite, 121
 stała
 fill_parent, 52
 INTERCEPTOR_SHIELDS, 198
 match_parent, 52
 PLAYER_BULLET_SPEED, 198, 205
 PLAYER_SHIP, 160
 SCOUT_SHIELDS, 198
 SPLASH_SCREEN_MUSIC, 71
 WARSHIP_SHIELDS, 198
 WEAPONS_SHEET, 198
 statek
 przechwytyjący, 180, 185
 wojenny, 194
 zwiadowczy, 189, 243
 sterowanie ruchem, 273
 szablon, 50
 exitselector.xml, 65
 FrameLayout, 51
 graficzny, 51

LinearLayout, 51
 RelativeLayout, 64
 startselector.xml, 65
 szczegóły gry, 32
 sztuczna inteligencja, AI, 171, 177, 180, 189, 194

Ś

śledzenie
 akcji gracza, 128
 elementów sterujących, 271
 intencji gracza, 270
 środowisko programistyczne, 27
 Eclipse, 27
 NetBeans, 38

T

tablica
 enemies[], 173, 178
 indices[], 94
 spriteSheets[], 172, 201
 textures[], 94, 260
 vertices[], 94, 124, 258
 tablice mapujące tekstury, 124
 tekstura, 89, 110, 159
 tekstura ściany, 266
 tło, 92
 tło elementu ImageButton, 67
 tożsamość, 91
 trajektoria pocisków, 201, 203
 tworzenie
 arkusza sprite'ów, 197
 czynności, 42, 82
 efektów rozmycia, 55
 ekranu powitalnego, 41
 elementu Activity, 44
 głównego menu, 63
 gry Star Fighter, 38
 interfejsu sterowania, 269
 katalogu raw, 71
 klasy, 42
 klasy tekstury, 156
 klucza, 234
 korytarza, 256, 257
 kwadratu, 244
 magazynu kluczy, 233
 obrazu, 47
 pliku szablonu, 50
 powierzchni, 87
 projektu 3D, 240
 przeciwników, 169, 173
 renderera, 86
 sztucznej inteligencji, 180, 189, 194
 tablic mapujących, 124
 usługi muzycznej, 72

- uzbrojenia, 198
- wątku gry, 58
- widoku gry, 83
- typ statku, 163

U

- uruchamianie gry, 58
- ustawianie orientacji ekranu, 46
- usuwanie pocisków, 210
- uzbrojenie, 197–202

W

- warstwy tła, 82, 108
- wątek gry, 59
- wczytywanie
 - arkusza sprite'ów, 177
 - tekstur, 110, 131
- węzeł .StarfighterActivity, 43
- węzły aplikacji, 43
- widok SFGameView, 85
- właściwość enemyType, 179
- współrzędne punktu, 166
- wybór magazynu kluczy, 233

- wykrywanie
 - dotknięcia, 272
 - importów, 123
 - kolizji, 207, 275
 - krawędzi, 204
 - typów kolizji, 209
- wyświetlanie gry, 115
- wzór na współrzedną punktu, 166

Z

- zabijanie czynności, 61
- zatrzymywanie wątku, 116
- zmiana klasy w czynność, 43
- zmienna
 - corridorZPosition, 274
 - display, 271
 - isRunning, 75
 - nextShot, 204
 - playerFlightAction, 128
 - playerMovementAction, 273
- znacznik ImageView, 52

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

GRUPA WYDAWNICZA

 **Helion SA**

System operacyjny Android podbił rynek smartfonów, a obecnie walczy o panowanie na tabletach. Swoją pozycję zawdzięcza niezwykle intuicyjnemu interfejsowi użytkownika, szerokim możliwościom dostosowania do własnych potrzeb, genialnej wręcz integracji z usługami firmy Google oraz niewyobrażalnej liczbie dostępnych aplikacji. Te wszystkie możliwości czynią z niego idealną platformę dla wszystkich programistów chcących stworzyć nową grę i zdobyć popularność. Jak się do tego zabrać?

Odpowiedzi dostarcza ta książka. W trakcie lektury poznasz cały proces tworzenia gry działającej zarówno na smartfonie, jak i na tablecie. Już tylko mały krok dzieli Cię od stworzenia pierwszej strzelanki 2D z tłem przewijanym z góry do dołu, a następnie czegoś bardziej zaawansowanego w trójwymiarze. Grafika 3D, sztuczna inteligencja przeciwników, zaawansowane efekty graficzne — to wszystko masz na wyciągnięcie ręki. Ponadto dowiesz się, jak wykrywać kolizje, sterować postaciami oraz zapewnić najwyższą wydajność Twojej grze. Książka ta poprowadzi Cię krok po kroku poprzez rozwój dwóch różnych gier komórkowych, poczynsz od pomysłu, a na kodzie skończywszy. Sięgnij po nią i opublikuj swoją pierwszą grę w Google Play!

Wykorzystaj potencjał platformy Android i:

- **zaprojektuj swoją pierwszą grę**
- **stwórz zaawansowane efekty graficzne**
- **obdarz przeciwników sztuczną inteligencją**
- **rzuć wyzwanie użytkownikom!**

Doskonałe źródło informacji dla pasjonatów platformy Android!

helion.pl
księgarnia internetowa

Nr katalogowy: 11717



Księgarnia internetowa
<http://helion.pl>



Zamówienia telefoniczne:
0 801 339900
0 601 339900

Apress®



Helion

Sprawdź najnowsze promocje:
• <http://helion.pl/promocje>
Książki najchętniej czytane:
• <http://helion.pl/bestsellery>
Zamów informacje o nowościach:
• <http://helion.pl/novosci>

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

sięgnij po **WIECEJ**



KOD KORZYŚCI

ISBN 978-83-246-5087-3



Cena: 49,00 zł

Informatyka w najlepszym wydaniu