

Popularna platforma — profesjonalne aplikacje!



# Java ME

Tworzenie zaawansowanych aplikacji na  
smartfony

Ovidiu Iliescu

Tytuł oryginału: Pro Java ME Apps

Tłumaczenie: Paweł Koronkiewicz (wstęp, rozdz. 1 – 8), Robert Górczyński (rozdz. 9 – 16)

ISBN: 978-83-246-3589-4

Original edition copyright © 2011 by Ovidiu Iliescu.

All rights reserved.

Polish edition copyright © 2012 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](mailto:helion@helion.pl)

WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:

<ftp://ftp.helion.pl/przyklady/jamets.zip>

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

<http://helion.pl/user/opinie/jamets>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

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

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

# Spis treści

	<b>O autorze .....</b>	<b>13</b>
	<b>O recenzencie technicznym .....</b>	<b>14</b>
	<b>Podziękowania .....</b>	<b>15</b>
	<b>Wstęp .....</b>	<b>17</b>
	Co znajdziesz w tej książce? .....	18
	Czego potrzebujesz? .....	18
	Zabawę czas zacząć .....	19
<b>Rozdział 1</b>	<b>Pierwsze kroki .....</b>	<b>21</b>
	Java ME i współczesne urządzenia przenośne .....	21
	Zalety platformy Java ME .....	21
	Wady platformy Java ME .....	22
	Podsumowanie .....	23
	Budowanie aplikacji Javy ME .....	24
	Pomysł .....	24
	Cele, funkcje, źródła przychodów i urządzenia docelowe .....	25
	Wybór urządzeń docelowych .....	26
	Identyfikacja ograniczeń technicznych aplikacji .....	27
	Aplikacje amatorskie i aplikacje profesjonalne .....	33
	Aplikacje Javy ME a elastyczność kodu .....	35
	Programowanie defensywne .....	36
	Unikanie błędnych założeń .....	38
	Zarządzanie złożonością .....	39
	Zastępowanie zasobów .....	40
	Luźne powiązania i decentralizacja architektury .....	42
	Unikanie niepotrzebnego obciążenia .....	43
	Podsumowanie .....	44
<b>Rozdział 2</b>	<b>Platforma aplikacji Javy ME .....</b>	<b>45</b>
	Znaczenie platformy aplikacji .....	45
	Dostosowanie platformy do aplikacji .....	46
	Definiowanie struktury platformy .....	47

Podstawowe typy obiektów .....	47
Zdarzenia .....	49
Obserwatory zdarzeń .....	50
Dostawcy danych .....	51
Odbiorcy (konsumenci) .....	52
Menedżery .....	52
Modele i kontrolery .....	53
Widoki .....	54
Standardowe obiekty .....	55
Klasa EventController .....	55
Menedżer kontrolerów zdarzeń .....	57
Główne obiekty i klasy .....	59
Klasa Application .....	59
Klasa EventManagerThreads .....	61
Klasa Bootstrap .....	64
Prosta aplikacja testowa .....	65
Podsumowanie .....	69
<b>Rozdział 3</b> <b>Definicje danych .....</b>	<b>71</b>
Po co implementujemy interfejs Model? .....	71
Niemodyfikowalne typy danych .....	72
Definiowanie typu Tweet .....	73
Definiowanie typu TwitterUser .....	74
Definiowanie interfejsu TwitterServer .....	75
Definiowanie typu UserCredentials .....	76
Definiowanie typu TweetFilter .....	77
Definiowanie interfejsu Timeline .....	78
Inteligentna reprezentacja danych .....	79
Podsumowanie .....	80
<b>Rozdział 4</b> <b>Moduł sieciowy .....</b>	<b>81</b>
Instalowanie i konfigurowanie biblioteki .....	81
Obiekty wysokiego poziomu .....	84
Własne typy danych .....	84
Budowa implementacji obiektu TwitterServer .....	85
Ogólna struktura klasy .....	85
Inicjalizacja obiektu .....	86
Logowanie .....	87
Wysyłanie wiadomości .....	91
Pobieranie wiadomości .....	92
Metody getMyProfile() i getProfileFor() .....	103
Mechanizmy pracy sieciowej w aplikacjach Javy ME .....	104
Nie wyważaj otwartych drzwi .....	104
Mobilny internet ma swoją specyfikę .....	105
Pamiętaj o ograniczeniach urządzeń docelowych .....	106

	Zapewnij obsługę dławienia komunikacji i trybu uśpienia .....	106
	Efektywne przesyłanie danych .....	107
	Unikanie nadmiernej rozbudowy warstwy sieciowej .....	108
	Zachowuj niezależność .....	109
	Podsumowanie .....	109
<b>Rozdział 5</b>	<b>Moduł pamięci trwałej .....</b>	<b>111</b>
	Java ME a trwałe zapis danych .....	111
	Projektowanie modułu pamięci trwałej .....	113
	Dostawcy usług pamięci trwałej .....	114
	Obiekty zapisujące i odczytujące rekordy .....	115
	Obiekty serializujące i deserializujące .....	117
	Obiekty pomocnicze .....	118
	Implementowanie podstawowej architektury modułu .....	118
	Implementowanie obiektów serializujących i deserializujących (Serializer i Deserializer) .....	118
	Implementowanie obiektów odczytujących i zapisujących (RecordReader i RecordWriter) .....	121
	Implementowanie dostawcy usługi pamięci trwałej (PersistenceProvider) .....	122
	Testowanie kodu .....	124
	Pisanie obiektów pomocniczych .....	125
	Moduł pamięci trwałej w praktyce .....	130
	Rozwijanie modułu .....	131
	Podsumowanie .....	132
<b>Rozdział 6</b>	<b>Moduł interfejsu użytkownika .....</b>	<b>133</b>
	Po co budować moduł UI od podstaw? .....	134
	Wprowadzenie .....	135
	Widżety .....	135
	Kontenery .....	137
	Prostokąty obcinania .....	138
	Widoki .....	142
	Motywy (kompozycje) .....	143
	Obsługa interakcji z użytkownikiem .....	144
	Podstawowe mechanizmy widżetów .....	146
	Klasa BaseWidget .....	146
	BaseContainerWidget i BaseContainerManager .....	148
	Klasy konkretne widżetów .....	154
	Klasy VerticalContainer i HorizontalContainer .....	154
	Klasa SimpleTextButton .....	158
	Klasa StringItem .....	160
	Klasa InputStringItem .....	163
	Klasa GameCanvasView .....	165

Testowanie modułu interfejsu użytkownika .....	167
Implementowanie UI dla urządzeń z ekranem dotykowym .....	169
Kilka ważnych porad .....	171
Podsumowanie .....	173
<b>Rozdział 7    Moduł lokalizacji .....</b>	<b>175</b>
Charakterystyka dobrego modułu lokalizacji .....	175
Standardowe mechanizmy lokalizacji platformy Java ME .....	176
Budowa własnego mechanizmu lokalizacji aplikacji Javy ME .....	177
Przetwarzanie plików lokalizacji .....	179
Ładowanie danych lokalizacji .....	181
Testowanie modułu lokalizacji .....	185
Implementowanie zaawansowanych mechanizmów lokalizacji .....	186
Podsumowanie .....	188
<b>Rozdział 8    Łączenie elementów w gotową aplikację .....</b>	<b>189</b>
Uruchamianie aplikacji .....	189
FlowController — kontroler przepływu sterowania .....	190
TweetsController — kontroler wiadomości tweet .....	193
WelcomeScreenController i WelcomeForm — kontroler i formularz ekranu powitalnego .....	196
MainScreenController i MainForm — kontroler i formularz ekranu głównego .....	200
SettingsScreenController i SettingsForm — kontroler i formularz ekranu ustawień konfiguracyjnych .....	204
Klasa EVT .....	208
Modyfikowanie i rozbudowa aplikacji .....	210
Poprawienie obsługi błędów .....	210
Rozbudowa zakresu funkcji .....	211
Dopracowanie platformy UI .....	212
Podsumowanie .....	212
<b>Rozdział 9    Fragmentacja urządzenia .....</b>	<b>213</b>
Fragmentacja sprzętowa .....	215
Procesor .....	215
RAM .....	218
Ekran .....	220
Inne rozważania dotyczące sprzętu .....	222
Fragmentacja możliwości .....	224
Niezgodność API .....	228
Lokalna niezgodność API .....	229
Globalna niezgodność API .....	231
Niezgodność wynikająca ze swobody interpretacji .....	234

Struktury przenoszenia kodu .....	235
Preprocesor .....	236
Baza danych informacji o urządzeniu .....	237
Silnik kompilacji .....	238
Abstrakcja API .....	239
Obsługa wielu platform .....	239
Kod pomocniczy i narzędzia .....	240
Biblioteka interfejsu użytkownika .....	240
Pomoc techniczna .....	241
Licencja na kod .....	242
Programowanie na różne platformy i narzędzia	
służące do przenoszenia kodu .....	242
Podsumowanie .....	243
<b>Rozdział 10 Optymalizacja kodu .....</b>	<b>245</b>
Krótkie wprowadzenie do optymalizacji kodu .....	246
Techniki optymalizacji kodu .....	248
Szybkie przełączanie ścieżki wykonywania kodu .....	248
Unikanie powielania kodu .....	249
Wykorzystanie zalet płynących z lokalizacji .....	249
Optymalizacja operacji matematycznych .....	250
Rezygnacja z użycia pętli .....	253
Kod osadzony .....	254
Optymalizacja pętli wykonujących operacje matematyczne .....	254
Usunięcie warunków z pętli .....	256
Eliminowanie iteracji specjalnych z pętli .....	256
Rozbicie pętli .....	257
Unikanie wysokiego poziomu funkcji języka .....	258
Pozostawanie przy podstawach .....	259
Unikanie tworzenia niepotrzebnych obiektów .....	259
Optymalizacja dostępu do pamięci .....	263
Techniki optymalizacji algorytmu .....	263
Porównywanie algorytmów .....	264
Usprawnianie algorytmów .....	266
Podsumowanie .....	273
<b>Rozdział 11 Dopracowanie aplikacji i usprawnienia interfejsu użytkownika ....</b>	<b>275</b>
Dopracowanie aplikacji w każdym szczególe .....	276
Omówienie dopracowywania szczegółów aplikacji .....	276
Prawidłowe umieszczenie pomocy w aplikacji .....	278
Dodawanie informacji kontekstowych .....	279
Dodawanie poprawnego systemu informacji zwrotnych .....	280
Dodawanie możliwości obsługi tekstu adaptacyjnego .....	281
Dodawanie obsługi historii i automatycznego uzupełniania .....	283
Dodawanie wykrywania intencji .....	284
Synchronizacja danych pomiędzy urządzeniami .....	286

Usprawnienie interakcji z użytkownikiem .....	287
Unikanie zmylenia użytkownika .....	287
Stosowanie jak najprostszego interfejsu użytkownika .....	289
Ułatwienie klientom dotarcia do Ciebie .....	290
Utworzenie nie tylko przenośnych wersji aplikacji .....	291
Dostarczanie częstych uaktualnień aplikacji .....	292
Dodawanie obsługi motywów .....	293
Reklamowanie podobnych produktów .....	294
Wybór dopracowanych szczegółów do zaimplementowania .....	295
Podsumowanie .....	295
<b>Rozdział 12 Testowanie aplikacji Javy ME .....</b>	<b>297</b>
Zbieranie informacji procesu usuwania błędów .....	297
Wykonywanie testów jednostkowych .....	299
Rozwiązywanie najczęstszych problemów z testami jednostkowymi ...	300
Zbieranie wysokiej jakości danych procesu usuwania błędów .....	301
Przeprowadzanie testów w środowisku biurkowym .....	302
Wizualny proces usuwania błędów .....	303
Przeprowadzanie testów baterii .....	304
Testowanie aplikacji w różnych sytuacjach .....	306
Testowanie usprawnień w zakresie wydajności i technik optymalizacji .....	307
Podsumowanie .....	308
<b>Rozdział 13 Zaawansowana grafika w Javie ME .....</b>	<b>309</b>
Używanie przygotowanej wcześniej grafiki .....	310
Używanie maski obrazu .....	314
Używanie technik mieszania obrazów .....	316
Obracanie obrazów .....	321
Zmiana wielkości obrazu .....	326
Implementacja innych efektów graficznych .....	328
Połączenie kilku efektów graficznych .....	329
Podsumowanie .....	330
<b>Rozdział 14 Odpowiednie nastawienie programisty Javy ME .....</b>	<b>331</b>
Możliwości Javy ME nierozzerwalnie łączą się z urządzeniem działającym pod jej kontrolą .....	332
Najlepsze praktyki dotyczące optymalizacji aplikacji .....	335
Trzymaj się priorytetów .....	337
Ważne jest myślenie nieszablonowe .....	340
Pamiętaj o prostocie .....	342
Ustalenie standardu opisującego sposób działania aplikacji .....	344
Planowanie dla najgorszej z możliwych sytuacji .....	347
Określenie ograniczeń aplikacji .....	347
Podsumowanie .....	350



---

<b>Rozdział 15</b>	<b>Przyszłość Javy ME .....</b>	<b>351</b>
	Ewolucja sprzętu działającego pod kontrolą Javy ME .....	351
	Ewolucja API Javy ME .....	353
	Ewolucja nastawienia programisty Javy ME i filozofii tworzenia oprogramowania .....	354
	Rynek docelowy dla Javy ME .....	355
	Java ME i inne platformy .....	356
	Rodzaje aplikacji Javy ME .....	358
	Innowacje Javy ME .....	359
	Śmierć Javy ME .....	360
	Podsumowanie .....	361
<b>Rozdział 16</b>	<b>Zakończenie .....</b>	<b>363</b>
	Materiał przedstawiony w książce .....	363
	Co dalej? .....	364
	Na koniec .....	365
	<b>Skorowidz .....</b>	<b>367</b>



# Zaawansowana grafika w Javie ME

Tradycyjnie grafika nigdy nie była mocną stroną Javy ME. Opracowana we wczesnych dniach urządzeń przenośnych platforma Java ME była przeznaczona do działania na maksymalnej liczbie urządzeń, co wiąże się z wieloma kompromisami. Jednym z takich kompromisów jest ograniczenie w postaci wyświetlania grafiki jedynie za pomocą wyjątkowo okrojonego zestawu API 2D.

**Uwaga!**

Wprawdzie w ostatnich latach Java ME została wyposażona w obsługę grafiki 3D, to jednak w większości urządzeń nadal można używać tylko okrojonego zestawu API 2D. Wynika to stąd, że ich wirtualne maszyny Javy nie zawierają obsługi grafiki 3D, lub jest powodowane naprawdę kiepską wydajnością grafiki 3D, jaka jest osiągnięta w rzeczywistości. Co więcej, na platformie Java ME obsługa grafiki 3D jest odpowiednia jedynie dla gier i na dodatek dość ograniczona.

Na początku oferowane możliwości były wystarczające, ale wraz z nadejściem ery smartfonów i zwiększaniem się ich popularności aplikacje Javy ME zaczęły konkurować z aplikacjami utworzonymi z użyciem graficznych API o znacznie większych możliwościach, a tym samym prezentujących się o wiele lepiej.

Na szczęście, choć API graficzne Javy ME pozostało w ogromnej mierze niezmiennione od wielu lat, urządzenia działające pod kontrolą Javy ME zostały usprawnione i teraz oferują znacznie potężniejsze możliwości graficzne i obliczeniowe niż kiedykolwiek wcześniej. Dzięki temu możemy rozbudować istniejące możliwości graficzne poprzez utworzenie własnych procedur graficznych oraz zastosowanie pewnych sztuczek i technik, które były niemożliwe do wykorzystania kilka lat temu. Tak więc świetnie wyglądające aplikacje Javy ME nie są już mrzonką — obecnie są możliwe do utworzenia.

W tym rozdziale zostaną przedstawione pewne najważniejsze sztuczki i techniki dotyczące tworzenia grafiki na platformie Java ME. Ich opanowanie pozwoli Ci na tworzenie zadziwiających efektów wizualnych dla aplikacji Javy ME, które mogą zachęcić użytkowników i dać Ci przewagę na konkurencją.

Na początek trzeba wspomnieć, że niektóre z omówionych tutaj technik do prawidłowego działania wymagają funkcji MIDP 2.0, takich jak `Image.getRGB()`. To niestety oznacza, że nie mogą być używane w urządzeniach oferujących jedynie MIDP 1.0. Z drugiej strony urządzenia MIDP 1.0 zwykle i tak nie posiadają wystarczającej mocy obliczeniowej do wykorzystania wspomnianych technik.

Kolejna ważna kwestia wiąże się z wydajnością. Fragmenty kodu przedstawione w rozdziale zostały utworzone w taki sposób, aby były maksymalnie czytelne, a nie w celu zapewnienia jak największej wydajności. Z tego powodu ich wydajność jest daleka od optymalnej. Jednak dzięki zastosowaniu rozwiązań omówionych w rozdziale 10. można znacznie zwiększyć wydajność kodu prezentowanego w tym rozdziale. W rzeczywistości optymalizacja tych procedur stanowi prawdopodobnie najlepsze ćwiczenie dla technik przedstawionych w rozdziale 10. i daje Ci możliwość wypróbowania większości z nich.

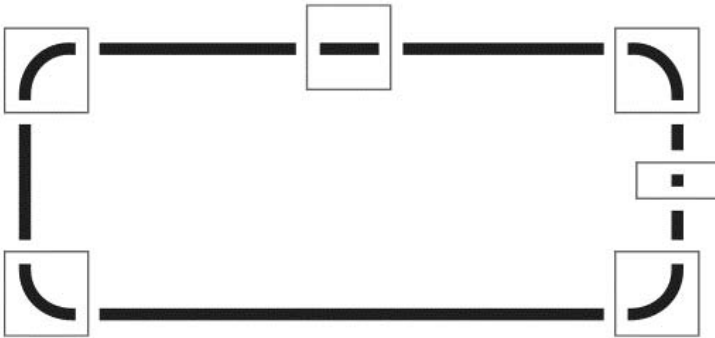
## Używanie przygotowanej wcześniej grafiki

O ile urządzenia docelowe naprawdę nie są niskiej klasy, maksymalna wielkość pliku JAR dla takich urządzeń prawdopodobnie będzie znacznie większa niż rzeczywista wielkość tworzonych przez Ciebie plików JAR. Innymi słowy, zwykle masz sporo wolnego miejsca, więc dlaczego tego nie wykorzystać? Jednym z najlepszych sposobów zagospodarowania tych dodatkowych bajtów jest przechowywanie *przygotowanej wcześniej grafiki* dla Twoich aplikacji (na przykład elementów interfejsu użytkownika, czcionek, animacji itd.). To brzmi sensownie: dlaczego obciążać urządzenie dynamicznym generowaniem grafiki podczas działania aplikacji, skoro można ją po prostu wcześniej przygotować? Oprócz tego przygotowana wcześniej grafika zwykle charakteryzuje się większą jakością niż generowana dynamicznie, podobnie jak film zrealizowany w technologii 3D pod względem technicznym jest lepszy od gry wideo.

W pierwszej kolejności powinieneś spróbować umieścić w aplikacji wygenerowane już elementy interfejsu użytkownika. W przypadku elementów o stałej wielkości, takich jak pola wyboru, wystarczy przygotować oddzielne obrazy dla poszczególnych stanów elementu (na przykład naciśnięty, zaznaczony, niedostępny, wyłączony, zaznaczony i niedostępny), a następnie wyświetlać je bezpośrednio na ekranie w metodzie `paint()`. Takie rozwiązanie jest wyjątkowo proste, a otrzymany wynik przedstawia się imponująco.

Natomiast w przypadku elementów o zmiennej wielkości sytuacja jest nieco bardziej skomplikowana. Pomysł polega na wykorzystaniu techniki o nazwie *łączenie* — element interfejsu jest dzielony na dwa lub więcej fragmentów, z których część jest o stałej wielkości, a część może być wielokrotnie powielana. Następnie tak przygotowane fragmenty służą do zbudowania obrazużądanego elementu o wskazanej wielkości. Przykładowo przyjmujemy założenie, że chcesz wyświetlić element abstrakcyjnie zdefiniowany jako prostokąt o zaokrąglonych wierzchołkach — przycisk lub okno dialogowe. Na rysunku 13.1 pokazano przykładowe fragmenty, z których można zbudować tego rodzaju element.

Widać wyraźnie, że cztery fragmenty tworzące wierzchołki mają niezmiennie wymiary. Natomiast inne fragmenty, na przykład tworzące poziome lub pionowe linie obramowania, mogą być powielane wielokrotnie aż do utworzenia linii łączącej wierzchołki. W zależności od wymagań istnieje możliwość dodania kolejnych fragmentów; jeśli na przykład górna i dolna linia są inne lub gdy linie mają zawierać jakieś skomplikowane wzory (w takim przypadku trzeba będzie dostarczyć odpowiednie fragmenty używane do budowy takich linii).



**Rysunek 13.1.** Fragmenty, z których można zbudować prostokąt z zaokrąglonych wierzchołkach

Przygotowaną wcześniej grafikę można wykorzystać także do przechowywania animacji interfejsu użytkownika, której wygenerowanie w czasie działania aplikacji zajmowałoby dużo czasu lub byłoby wręcz niemożliwe. Niemalże taka sama technika jest stosowana w grach bazujących na elementach o nazwie *sprite*. W przypadku interfejsu użytkownika animowany *sprite* jest używany na przykład w tle przycisku jako ikona powiadamiająca o otrzymaniu wiadomości e-mail lub jako pasek wczytywania. Wspomniane animacje stanowią dopracowane szczegóły, które są tak ważne w aplikacjach Javy ME. Dzięki nim wygląd interfejsu użytkownika wydaje się bardziej naturalny i przypominający ten z urządzeń biurkowych lub smartfonów. Przykładem może być sytuacja, gdy po umieszczeniu kursora nad przyciskiem dany przycisk powoli się rozjaśnia, zamiast natychmiast przejść do stanu „wybrany”.

Przygotowane wcześniej obrazy to również doskonały sposób tworzenia motywów aplikacji. W takim przypadku zmiana elementów interfejsu użytkownika sprowadza się jedynie do użycia innego pliku obrazu, nie jest wymagane wprowadzanie zmian w kodzie.

Wiele innych małych elementów graficznych można wcześniej przygotować, aby zapewnić aplikacji lepszą wydajność działania i ładniejszy wygląd. Przykładowo w wykresie liniowym różne segmenty są najczęściej ograniczone za pomocą pewnego rodzaju kuli. Zamiast wspomniane kule generować w czasie rzeczywistym, używając funkcji niskiego poziomu (takich jak `fillArc()`), można je po prostu wcześniej przygotować. Takie rozwiązanie będzie nie tylko szybsze (zazwyczaj), ale również lepsze pod względem wizualnym, ponieważ przygotowana wcześniej grafika kuli może mieć półprzezroczyste piksele wokół krawędzi kuli, co spowoduje, że sprawi ona wrażenie bardziej wygładzonej.

Inny doskonały sposób wykorzystania przygotowanej wcześniej grafiki to użycie jej jako podstawy dla efektów graficznych. Jak się wkrótce przekonasz, wiele efektów graficznych, które możesz zaimplementować w Javie ME, wymaga użycia dwóch plików obrazów: obrazu źródłowego i drugiego wykorzystywanego jako parametr (na przykład maski lub filtru koloru). Dzięki wcześniejszemu przygotowaniu tego drugiego obrazu zamiast jego generowania w trakcie działania aplikacji można znacznie zwiększyć wydajność tak tworzonych efektów graficznych.

Czasami można nawet przygotować wcześniej całe dane wyjściowe efektu graficznego, zwłaszcza gdy z natury nie są one dynamiczne (na przykład danymi wyjściowymi nie jest klatka animacji). Przypuśćmy, że chcesz wykorzystać obraz maski do utworzenia logo Twojej firmy z teksturą nieba. Taki efekt można wygenerować całkowicie w trakcie działania aplikacji lub wcześniej przygotować obraz maski dla logo i umieścić wygenerowaną maskę w wynikowym

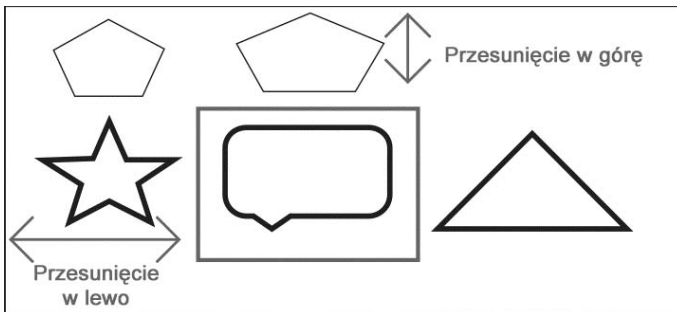
pliku JAR. To drugie rozwiązanie na pewno będzie szybsze, biorąc pod uwagę wydajność działania aplikacji, ale jego oczywistym efektem ubocznym jest zwiększenie wielkości wynikowego pliku JAR.

**Wskazówka**

Dobłą strategią jest mieszanie grafiki dynamicznej i wcześniej przygotowanej. Przykładowo, jeżeli tworzysz grę bazującą na elementach sprite, które muszą obracać się w szesnastu kłatkach, rozważ przygotowanie ośmiu wymaganych klatek i wygenerowanie pozostałych ośmiu. W ten sposób zachowasz równowagę pomiędzy wielkością wynikowego pliku JAR i wydajnością aplikacji w trakcie działania. Co więcej, po zastosowaniu techniki *sprite mirroring* musisz przygotować i wygenerować jedynie połowę klatek.

Wreszcie jednym z najlepszych sposobów wykorzystania wcześniej przygotowanej grafiki jest przechowywanie czcionek. Rodzime czcionki urządzenia zwykle nie przedstawiają się najlepiej. Co więcej, ze względu na różnice w wielkości i kształcie mogą zepsuć wygląd interfejsu użytkownika w różnych urządzeniach. Z tego powodu większość wysokiej jakości aplikacji Javy ME wykorzystuje przygotowane wcześniej czcionki bitmapowe. W dobrze wyglądającym interfejsie użytkownika aplikacji zwykle potrzebujesz kilku czcionek lub co najmniej tej samej czcionki w różnych wielkościach i stylach (pogrubiona, pochylona itd.) — jeśli ilość wolnego miejsca pozwala, wykorzystaj czcionki bitmapowe.

Ostatnią poruszoną tutaj kwestią dotyczącą przygotowanej wcześniej grafiki jest prawidłowy sposób jej przechowywania. Umieszczenie każdego elementu graficznego w oddzielnym pliku jest bardzo nieefektywne, zarówno pod względem wydajności, jak i miejsca wymaganego w wynikowym pliku JAR oraz potrzebnej ilości pamięci. Najlepszym sposobem przechowywania przygotowanej wcześniej grafiki jest jej umieszczenie jako części mapy obrazów. Mówiąc dokładniej, tworzysz jeden większy *obraz nadrzędny* (zobacz rysunek 13.2). Odległości pomiędzy górną i lewą krawędzią obrazu nadrzędnego i elementami w poszczególnych sekcjach noszą nazwę odpowiednio *przesunięcia w górę* oraz *przesunięcia w lewo*.



**Rysunek 13.2.** Mapa obrazu, na której zaznaczony został jeden element. Na rysunku pokazano także odległości przesunięcia w górę i w lewo

**Wskazówka**

Istnieje możliwość optymalizacji wielkości pliku poprzez konwersję plików PNG na format używający palety kolorów. To będzie użyteczne rozwiązanie w przypadku obrazów wykorzystujących jedynie ograniczoną liczbę kolorów, ale już mniej przydatne w obrazach używających wielu kolorów. Warto również pamiętać, że poza formatem PNG, który jest wymagany przez specyfikację Javy ME, pozostałe formaty graficzne są opcjonalne. Dlatego też, rozważając zastosowanie alternatywnego formatu pliku graficznego (takiego jak GIF lub JPEG), upewnij się, że będzie on obsługiwany w urządzeniu docelowym.

W celu wyświetlenia elementu graficznego będącego częścią mapy obrazów konieczne jest obliczenie jego współrzędnych w obrazie nadrzędnym z uwzględnieniem *przesunięcia do elementu graficznego*. Kolejny krok to przycięcie obrazu do obszaru zawierającego jedynie żądany element graficzny. Następnie trzeba *wyświetlić obraz nadrzędny o obliczonych wcześniej wymiarach*. Ponieważ wymiary mapy obrazów są obliczane z uwzględnieniem obszaru wybranego elementu graficznego, rzeczywisty element będzie wyświetlony dokładnie w żądanej pozycji. Wyjaśnienie za pomocą słów brzmi nieco zawile, ale przedstawiony poniżej fragment kodu powinien wyjaśnić omówioną koncepcję (zobacz listing 13.1).

**Listing 13.1.** Fragment kodu pozwalający na wyświetlenie elementu, który stanowi część mapy obrazów

```
function drawAt( Image imageMap, Graphics target, int x, int y, int
↳elementOffsetLeft,
int elementOffsetTop, int elementWidth, int elementHeight)
{
    // Obliczenie docelowych wymiarów mapy obrazów z uwzględnieniem
    // przesunięcia względem żadanego elementu graficznego.
    imageX = x - elementOffsetLeft;
    imageY = y - elementOffsetTop;

    // Przycięcie obrazu wokół elementu przeznaczonego do wyświetlenia.
    target.clip(x, y, elementWidth, elementHeight);

    // Wyświetlenie mapy obrazów o obliczonych wymiarach.
    // To gwarantuje, że element faktycznie będzie wyświetlony dokładnie w pozycji (x, y),
    // a ponieważ mapa została przycięta i zawiera jedynie obszar z wyświetlanym elementem,
    // to na ekranie zostanie wyświetlony jedynie żądany element graficzny.
    target.drawImage(imageMap, imageX, imageY);
}
```

Używanie map obrazów to bardzo ważna technika dla aplikacji Javy ME. Nie stanowi zbyt dużego obciążenia pod względem zasobów (konieczne jest utworzenie i zarządzanie mniejszą liczbą obiektów Image, a ponadto trzeba obsłużyć mniejszą ilość poszczególnych plików graficznych). To także całkiem szybkie rozwiązanie, ponieważ większość telefonów zawiera sprzętową obsługę przycinania. W rzeczywistości, jeśli używasz dużej liczby elementów graficznych w pojedynczej operacji rysowania (na przykład podczas odświeżania ekranu), istnieje duże prawdopodobieństwo, że użycie mapy obrazów będzie *szybsze* niż wyświetlenie poszczególnych obrazów. Co więcej, wysokiej klasy telefony działające pod kontrolą Javy ME są wyposażone w dedykowane układy graficzne, które są używane do przyspieszania operacji graficznych Javy ME. Z powodu sposobu działania wspomnianych układów rysowanie różnych fragmentów tego samego obrazu jest niemal zawsze szybsze niż wyświetlanie poszczególnych obrazów, ponieważ w tym drugim przypadku układ musi przeprowadzać operacje przełączania buforów (obrazów). Różnica jednak będzie niezauważalna.

## Używanie maski obrazu

*Maskowanie obrazu* to technika podobna do przycinania. Podczas gdy przycinanie oznacza „przycięcie” obrazu do wskazanego obszaru prostokątnego, maskowanie obrazu jest przycięciem obrazu do dowolnej maski, którą najczęściej jest inny obraz.

Maskowanie obrazu jest niezwykle użyteczne w wielu sytuacjach, począwszy od nakładania efektów (na przykład teksturowane czcionki — obraz jest teksturą, maską jest czcionka) aż do praktycznych, pomysłowych sztuczek (na przykład dynamiczne tła dla komponentów niebędących prostokątami; operacja bazuje na rzeczywistym kształcie i wielkości komponentu — w tym przypadku kształt komponentu jest maską).

Oprócz tego, że maskowanie oferuje potężne możliwości, jest to także operacja niezwykle łatwa do implementacji. Wszystko sprowadza się do prawidłowego zdefiniowania maski. Mówiąc dokładniej, maska musi być obrazem, którego część pikseli będzie całkowicie nieprzezroczysta, a część w pewnym stopniu lub w całości przezroczysta. Po zastosowaniu maski względem obrazu źródłowego piksele nieprzezrocyste spowodują utworzenie całkowicie nieprzezroczystych pikseli w obrazie docelowym, natomiast przezrocyste i półprzezrocyste piksele maski utworzą przezrocyste i półprzezrocyste piksele w obrazie wynikowym. Bardzo ważna jest przezroczystość pikseli w obrazie maski, natomiast ich kolor jest nieistotny z punktu widzenia naszych potrzeb.

Wiadomo już, że MIDP 2.0 pozwala na pobieranie poszczególnych pikseli obrazu za pomocą metody `Image.getRGB()`. Wynikiem jest tablica liczb całkowitych, a każda liczba w tablicy odpowiada pikselowi w obrazie źródłowym. Kanał danych każdego piksela (to znaczy wartości alfa i kolorów czerwonego, zielonego oraz niebieskiego) jest zapisywany poprzez zarezerwowanie ośmiu bitów liczby całkowitej dla każdego kanału. Po zastosowaniu trybu szesnastkowego zapis przyjmuje postać `0xAARRGGBB`.

Mając to wszystko na uwadze, Twoim zadaniem jest pobranie kanału alfa obrazu maski i jego połączenie z kanałami RGB obrazu źródłowego. To całkiem łatwe zadanie, jak pokazano na listingu 13.2 (pogrubione wiersze kodu). Po połączeniu otrzymujemy dane ARGB, które można po prostu wyświetlić na ekranie lub umieścić w dowolnym innym obiekcie `Graphics`.

### Listing 13.2. Operacja maskowania obrazu w Javie ME

```
public void drawMaskedImage(Image source, Image mask, Graphics g, int x, int y)
{
    // Zarezerwowanie tablicy dla pikseli danych każdego obrazu.
    int [] sourceData = new int[source.getHeight()*source.getWidth()];
    int [] maskData = new int[mask.getHeight()*mask.getWidth()];

    // Pobranie poszczególnych pikseli każdego obrazu (źródło, maska).
    source.getRGB(sourceData, 0, source.getWidth(), 0, 0, source.getWidth(),
    source.getHeight());
    mask.getRGB(maskData, 0, mask.getWidth(), 0, 0, mask.getWidth(),
    mask.getHeight());

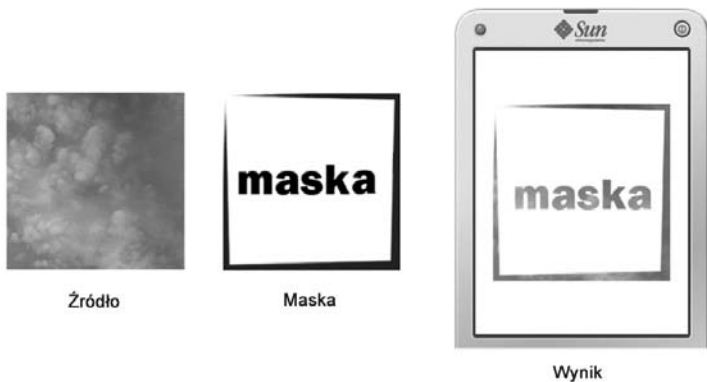
    // Połączenie kanału alfa maski z kanałami kolorów obrazu źródłowego.
    for (int i=0;i<sourceData.length;i++) {
        sourceData[i] = (maskData[i] & 0xFF000000) |
        (sourceData[i] & 0x00FFFFFF) ;
    }
}
```



```
// Wyświetlenie wygenerowanego obrazu.
g.drawRGB(sourceData, 0, source.getWidth(), x, y, source.getWidth(),
          source.getHeight(), true);
}
```

Warto tutaj wspomnieć o jednej bardzo ważnej kwestii: ze względu na czytelność i zwięzłość w kodzie przedstawionym na listingu 13.2 przyjęto założenie, że obrazy maski i źródła mają takie same wymiary. Jeżeli obrazy mają różne wymiary, skutkiem będzie nieprawidłowo wygenerowany obraz wynikowy lub nastąpi zgłoszenie wyjątku `ArrayIndexOutOfBoundsException`. Jednakże istnieje możliwość (i to wcale nie taka trudna do zrealizowania) przystosowania kodu do obsługi obrazów źródłowych i maski o innych wymiarach. Wprowadzenie takiej modyfikacji może być dla Ciebie dobrym ćwiczeniem.

I na tym kończy się implementacja maskowania obrazu w Javie ME! Otrzymany wynik jest całkiem dobry, o czym można się przekonać, patrząc na rysunek 13.3.



**Rysunek 13.3.** Maskowanie obrazu w działaniu

Zabawa wcale nie musi się na tym zakończyć. Za pomocą omówionej techniki można osiągnąć jeszcze więcej. Przykładowo tablica `maskData` może być generowana dynamicznie. To daje możliwość wykorzystania jej do utworzenia różnego rodzaju efektów, począwszy od efektów typu *Film grain* lub *Snow* aż po animowane przejścia podobne do stosowanych w efektach *Checkerboard* lub *Dissolve* w programie PowerPoint.

Trzeba mieć świadomość, że choć większość nowoczesnych urządzeń MIDP 2.0 obsługuje przezroczystość i stosowanie kanału alfa, to jednak ich obsługa nie jest wymagana przez standard Java ME. Co więcej, poziom przezroczystości może być odmienny w różnych urządzeniach i wahać się od 2 do 256. Dlatego też przed użyciem przezroczystości i maskowania obrazu upewnij się, że wymienione elementy zostały przetestowane w urządzeniu docelowym. Przykładowo obraz wykorzystujący pełne 256 poziomów przezroczystości będzie wyglądał kiepsko po wyświetleniu go w urządzeniu obsługującym zaledwie 4 poziomy przezroczystości.

## Używanie technik mieszania obrazów

*Mieszanie obrazów* w zasadzie odnosi się do połączenia dwóch obrazów, co skutkuje powstaniem obrazu wynikowego zawierającego informacje kolorów z obydwu obrazów. Z jednej strony to technika w pewnym sensie podobna do maskowania obrazu, gdyż dwa obrazy są używane do wygenerowania wynikowego. Z drugiej strony to zupełnie odmienna technika, ponieważ w przeciwieństwie do stosowania maski tutaj w obrazie wynikowym znajdują się informacje o kolorach z obydwu obrazów.

Istnieje wiele różnych rodzajów mieszania obrazów, począwszy od bardzo zaawansowanych (rodzaj efektu „miś polarny na słonecznej plaży”, które można osiągnąć w programach typu Photoshop) aż po proste (na przykład łączenie kanałów alfa; w efekcie dwa obrazy są nakładane na siebie; obraz na górze jest półprzezroczysty i pozwala na „prześwitwanie” fragmentów obrazu znajdującego się poniżej — ten rodzaj efektu był często stosowany w teledyskach produkowanych w latach osiemdziesiątych ubiegłego stulecia).

W tej sekcji zostaną przedstawione proste techniki mieszania obrazów. Bardziej zaawansowane tak naprawdę nie są odpowiednie dla Javy ME z powodu ograniczonych zasobów urządzeń. Warto tutaj przypomnieć, że proste niekoniecznie oznacza prymitywne. Jak się przekonasz, nawet proste techniki mieszania obrazów mogą dać interesujące wyniki.

W pierwszej kolejności poznamy wspomniane już wcześniej mieszanie kanałów alfa dwóch obrazów. Ta technika jest stosowana w wielu aplikacjach, począwszy od płynnych przesłań ekranu (ekran znajdujący się na górze powoli zanika, odsłaniając znajdujący się poniżej) aż do tworzenia lepszych i bardziej intuicyjnych narzędzi analizy danych (na przykład dwie wizualne reprezentacje danych — takie jak wykresy lub mapy kolorów — można na siebie nałożyć i sprawdzić, które obszary są takie same, a które inne).

Warto tutaj wspomnieć, że taka technika mieszania (podobnie jak wszystkie proste techniki mieszania obrazów) jest operacją przeprowadzaną „na poziomie piksela”. Oznacza to, że każdy piksel w obrazie wynikowym zależy tylko i wyłącznie od odpowiadającego mu piksela w obrazie źródłowym i jest niezależny od pikseli sąsiednich. Takie rozwiązanie jest więc łatwe w implementacji, a sam kod działa całkiem szybko, o ile nie są używane wyjątkowo duże obrazy.

Dla porównania bardziej zaawansowane techniki mieszania obrazów wymagają przetworzenia dwóch lub większej liczby pikseli obrazu źródłowego podczas generowania każdego piksela obrazu wynikowego. Niektóre techniki dla każdego piksela wynikowego wymagają użycia dziesiątek pikseli obrazu źródłowego. To oczywiście powoduje, że są znacznie wolniejsze, a przez to nieodpowiednie do zastosowania na platformie Java ME.

Powracając do tematu, najlepszym porównaniem efektu mieszania kanałów alfa jest... mieszanie farby. Kiedy chcesz wymieszać dwa kolory, to umieszczasz je na palecie i po prostu mieszasz. W zależności od ilości danego koloru w mieszance otrzymany wynik jest bliższy pierwszemu lub drugiemu kolorowi. Dokładnie tego samego oczekujemy od naszego efektu: w zależności od wskazanego poziomu przezroczystości obraz wynikowy ma w większym stopniu odzwierciedlać obraz znajdujący się na górze i w mniejszym ten na dole.

Na szczęście istnieje wzór matematyczny pozwalający na wykonanie powyższego zadania. Przy założeniu, że wszystkie wartości pochodzą z zakresu od 0 do 255 (to znacznie ułatwi nam pracę w Javie ME), wzór będzie miał postać przedstawioną na listingu 13.3.

**Listing 13.3.** Wzór opisujący mieszanie kanałów alfa

---

```
wynik = ( kolor1 * przezroczystość + kolor2 * ( 255-przezroczystość ) ) / 255
```

---

Ponieważ każdy kolor w rzeczywistości powstaje z czterech poszczególnych kanałów (wartość alfa i kolory czerwony, zielony i niebieski), powyższy wzór musi być zastosowany względem wszystkich kanałów (zobacz listing 13.4).

**Listing 13.4.** Zastosowany względem poszczególnych kanałów wzór opisujący mieszanie kanałów alfa

---

```
R.czerwony = ( kolor1.czerwony * przezroczystość + kolor2.czerwony *
↳( 255-przezroczystość ) ) / 255
R.zielony = ( kolor1.zielony * przezroczystość + kolor2.zielony *
↳( 255-przezroczystość ) ) / 255
R.niebieski = ( kolor1.niebieski * przezroczystość + kolor2.niebieski *
↳( 255-przezroczystość ) ) / 255
R.alfa = ( kolor1.alfa * przezroczystość + kolor2.alfa * ( 255-przezroczystość ) ) / 255
```

---

Ponieważ jest to operacja przeprowadzana „na poziomie piksela”, powyższy wzór musi być zastosowany względem wszystkich pikseli obrazu wynikowego. Odpowiedzialny za to kod Javy ME został przedstawiony na listingu 13.5, na którym *coeff* oznacza współczynnik przezroczystości w zakresie od 0 do 255.

**Listing 13.5.** Zastosowanie mieszania kanałów alfa w kodzie Javy ME

---

```
public void drawBlendedImage(Image bottom, Image top, Graphics g,
int coeff, int x, int y)
{
    // Zarezerwowanie tablicy pikseli danych dla każdego obrazu.
    int [] bottomData = new int[bottom.getHeight()*bottom.getWidth()];
    int [] topData = new int[top.getHeight()*top.getWidth()];

    // Pobranie poszczególnych pikseli każdego obrazu (źródło, maska).
    bottom.getRGB(bottomData, 0, bottom.getWidth(), 0, 0, bottom.getWidth(),
bottom.getHeight());
    top.getRGB(topData, 0, top.getWidth(), 0, 0, top.getWidth(), top.getHeight());

    // Zdefiniowanie wymaganych wartości piksela.
    int alpha1, alpha2;
    int red1, red2;
    int green1, green2;
    int blue1, blue2;
    int resultA, resultR, resultG, resultB;

    // Iteracja przez wszystkie piksele obrazów: „górnego” i „dolnego”.
    for (int i=0; i<bottomData.length; i++) {

        // Pobranie wartości poszczególnych kanałów dla każdego piksela (górnego, dolnego).
        alpha1 = (bottomData[i] & 0xFF000000) >>> 24;
        alpha2 = (topData[i] & 0xFF000000) >>> 24;
        red1 = (bottomData[i] & 0x00FF0000) >> 16;
        red2 = (topData[i] & 0x00FF0000) >> 16;
        green1 = (bottomData[i] & 0x0000FF00) >> 8;
```

```

green2 = (topData[i] & 0x0000FF00) >> 8;
blue1 = (bottomData[i] & 0x000000FF);
blue2 = (topData[i] & 0x000000FF);

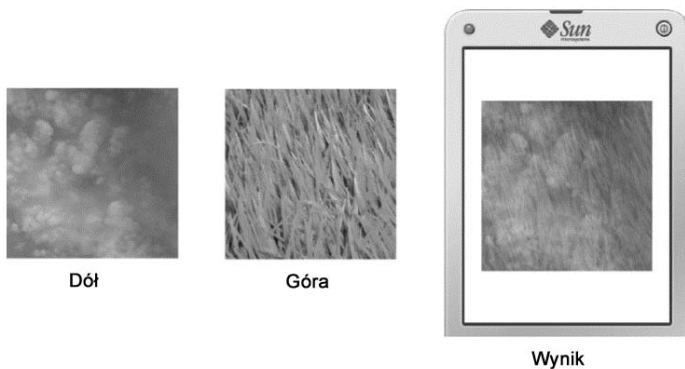
// Użycie wzoru mieszania obrazów.
resultA = ( alpha1 * coeff + alpha2 * (255 - coeff) ) / 255;
resultR = ( red1 * coeff + red2 * (255 - coeff) ) / 255;
resultG = ( green1 * coeff + green2 * (255 - coeff) ) / 255;
resultB = ( blue1 * coeff + blue2 * (255 - coeff) ) / 255;

// Utworzenie ostatecznej wartości piksela.
bottomData[i] = resultA << 24 | resultR << 16 | resultG << 8 | resultB ;
}

// Wyświetlenie wygenerowanego obrazu.
g.drawRGB(bottomData, 0, bottom.getWidth(), x, y, bottom.getWidth(),
bottom.getHeight(), true);
}

```

Pogrubione wiersze na powyższym listingu oznaczają te, w których przeprowadzana jest właściwa operacja. Przede wszystkim dla każdego piksela z dwóch obrazów źródłowych następuje wyodrębnienie wartości liczb całkowitych (integer) poszczególnych kanałów: wartości alfa i kolorów czerwonego, zielonego oraz niebieskiego. Odbywa się to za pomocą zastosowania klasycznej maski bitowej i przesuwania bitów. Następnie przedstawiony wcześniej wzór jest stosowany względem każdego kanału, co skutkuje otrzymaniem wartości ARGB dla kanału. Po połączeniu otrzymanych wartości do postaci pojedynczej liczby całkowitej efektem jest wartość piksela. Przykład działania powyższego kodu został pokazany na rysunku 13.4.



**Rysunek 13.4.** Wynik mieszania dwóch półprzezroczystych obrazów

Najzabawniejsza w efektach typu „na poziomie piksela” jest możliwość zmiany po prostu wzoru mieszania, co powoduje otrzymanie innego, interesującego wyniku. Przykładowo jeden z oferujących potężne możliwości typów mieszania nosi nazwę *Mnożenie*. Jak widać na listingu 13.6, wzór tego typu mieszania jest całkiem prosty, jeszcze mniej skomplikowany od przedstawionego wcześniej.

**Listing 13.6.** Wzór opisujący mieszanie typu Mnożenie

---

```
wynik = ( kolor1 * kolor2 ) / 255
```

---

Na listingu 13.7 przedstawiłem kod Javy ME wykorzystujący ten typ mieszania. Poniższy fragment kodu jest bardzo podobny do przedstawionego wcześniej na listingu 13.5, a jedyna rzeczywista różnica polega na zmianie we wzorze mieszania, co widać w pogrubionych wierszach kodu.

**Listing 13.7.** Kod Javy ME wykorzystujący mieszanie typu Mnożenie

---

```
public void drawMultipliedImage(Image firstImage, Image secondImage,
Graphics g, int x, int y)
{
    // Zarezerwowanie tablicy pikseli danych dla każdego obrazu.
    int [] bottomData = new int[firstImage.getHeight()*firstImage.getWidth()];
    int [] topData = new int[secondImage.getHeight()*secondImage.getWidth()];

    // Pobranie poszczególnych pikseli każdego obrazu (źródło, maska).
    firstImage.getRGB(bottomData, 0, firstImage.getWidth(), 0, 0,
    firstImage.getWidth(), firstImage.getHeight());
    secondImage.getRGB(topData, 0, secondImage.getWidth(), 0, 0,
    secondImage.getWidth(), secondImage.getHeight());

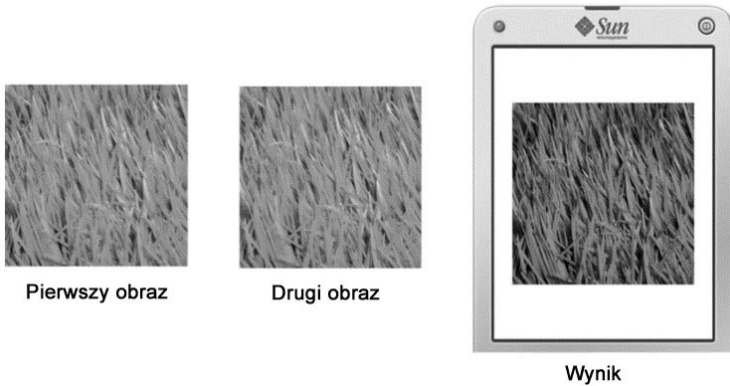
    // Zdefiniowanie wymaganych wartości piksela.
    int alpha1, alpha2;
    int red1, red2;
    int green1, green2;
    int blue1, blue2;
    int resultA, resultR, resultG, resultB;

    for (int i=0; i<bottomData.length; i++) {
        // Pobranie wartości poszczególnych kanałów dla każdego piksela (górze, dół).
        alpha1 = (bottomData[i] & 0xFF000000) >>> 24;
        alpha2 = (topData[i] & 0xFF000000) >>> 24;
        red1 = (bottomData[i] & 0x00FF0000) >> 16;
        red2 = (topData[i] & 0x00FF0000) >> 16;
        green1 = (bottomData[i] & 0x0000FF00) >> 8;
        green2 = (topData[i] & 0x0000FF00) >> 8;
        blue1 = (bottomData[i] & 0x000000FF);
        blue2 = (topData[i] & 0x000000FF);
        resultA = alpha1 * alpha2 / 255 ;
        resultR = red1 * red2 / 255 ;
        resultG = green1 * green2 / 255 ;
        resultB = blue1 * blue2 / 255 ;
        // Utworzenie ostatecznej wartości piksela.
        bottomData[i] = resultA << 24 | resultR << 16 | resultG << 8 | resultB ;
    }
    // Wyświetlenie wygenerowanego obrazu.
    g.drawRGB(bottomData, 0, firstImage.getWidth(), x, y, firstImage.getWidth(),
    firstImage.getHeight(), true);
}

```

---

Do czego można wykorzystać mieszanie typu Mnożenie? Jedną z możliwości jest jego zastosowanie względem tego samego obrazu, co powoduje zwiększenie kontrastu obrazu i poprawienie jego przejrzystości. Ten efekt pokazano na rysunku 13.5.



**Rysunek 13.5.** Zastosowanie mieszania typu Mnożenie względem tego samego obrazu

Inna możliwość to zastosowanie mieszania typu Mnożenie względem obrazu i przejścia pomiędzy kolorami. W ten sposób powstaje ładny efekt wyłaniania się obrazu z cienia, co pokazano na rysunku 13.6.



**Rysunek 13.6.** Zastosowanie mieszania typu Mnożenie względem obrazu i przejścia pomiędzy kolorami

Są jeszcze inne efekty mieszania obrazów, które można osiągnąć poprzez zmianę stosowanego wzoru — na przykład mieszanie typu Ekran, którego wzór został przedstawiony na listingu 13.8. Ten rodzaj mieszania jest w pewien sposób przeciwieństwem mieszania typu Mnożenie i ma tendencje do generowania obrazu wynikowego o nieco mniejszym kontraście.

**Listing 13.8.** Wzór mieszania typu Ekran

---

```
wynik = 255 - ( ( 255 - kolor1 ) * ( 255 - kolor2 ) ) / 255 )
```

---

Techniki mieszania obrazów to narzędzia o potężnych możliwościach. Rozsądnie stosowane względem elementów interfejsu użytkownika, a także w logach, obrazach użytkownika lub jako fragment mniejszych animacji mogą nadać aplikacji styl i zmniejszyć wizualną różnicę pomiędzy interfejsem użytkownika aplikacji Javy ME i smartfonów. Dzięki tym technikom wizualny efekt końcowy będzie jeszcze przyjemniejszy dla użytkowników niż zwykle oczekiwany od aplikacji Javy ME, co spowoduje, że wykorzystująca je aplikacja wyróżni się na tle pozostałych. Omówione techniki są szczególnie efektywne po zastosowaniu względem małych obrazów (na przykład o wymiarach  $64 \times 64$  lub  $32 \times 128$  pikseli), ponieważ w tych przypadkach są na tyle szybkie, że mogą być używane w czasie rzeczywistym bez powodowania widocznego spowolnienia obecnie dostępnych urządzeń działających pod kontrolą Javy ME.

## Obracanie obrazów

Na początek trzeba wspomnieć, że prawidłowe zrozumienie tego tematu wymaga dość dobrej wiedzy matematycznej (przede wszystkim z trygonometrii). W tym miejscu dołożę wszelkich starań, aby zagadnienie objaśnić w możliwie najbardziej niematematyczny sposób.

*Obracanie obrazów* to jedna z najczęściej używanych operacji na wszystkich platformach, zarówno biurkowych, jak i przenośnych. Jest powszechnie stosowana w grach i aplikacjach multimedialnych, choć przydaje się również w aplikacjach biznesowych. Niestety standardowo wbudowana w Javę ME obsługa obracania obrazów jest niezwykle ograniczona. Obrazy można więc obracać jedynie pod kątem będącym wielokrotnością kąta 90 stopni, obrót obrazu o dowolny kąt jest niemożliwy. To największe ograniczenie, ponieważ czasami zachodzi potrzeba obrócenia obrazu o kąt inny niż będący wielokrotnością kąta 90 stopni.

Przykładowo, jeżeli chcesz utworzyć widżet w postaci licznika dla swojej aplikacji, wskazówka licznika będzie musiała być wyświetlana pod dowolnym kątem. Za pomocą standardowych funkcji możesz to osiągnąć na trzy sposoby. Pierwszy, utworzyć obraz każdego możliwego położenia wskazówki licznika, co spowoduje znaczne zwiększenie wielkości wynikowego pliku JAR. Drugi, utworzyć mniejszą liczbę obrazów i pogodzić się z mniejszą dokładnością licznika. Trzeci, do narysowania wskazówki licznika wykorzystać funkcje niskiego poziomu, takie jak `drawLine()`, ale otrzymany w ten sposób efekt będzie po prostu brzydki. Żadna z wymienionych opcji nie jest szczególnie interesująca.

Z tego powodu najlepszym rozwiązaniem będzie samodzielne przygotowanie implementacji funkcji obracania obrazu, która umożliwi obrót o dowolny kąt. Być może takie rozwiązanie brzmi przerażająco, ale naprawdę nie jest trudne, zwłaszcza po poznaniu operacji matematycznych wymaganych do jego implementacji. Operacja obrócenia obrazu jest tak naprawdę całkiem prosta. Ponieważ każdy obraz składa się z poszczególnych pikseli, w celu przeprowadzenia obrotu obrazu konieczne jest obrócenie wszystkich jego pikseli wokół tego samego punktu odniesienia, którym najczęściej jest środek obrazu. Ogólnie rzecz biorąc, ta sama prosta operacja jest wykonywana wielokrotnie.

W celu zaimplementowania własnego rozwiązania przede wszystkim trzeba więc zdefiniować wzór obrotu punktu wokół punktu początkowego w systemie współrzędnych kartezjańskich. Zaangażowane w to operacje matematyczne są trudne do wyjaśnienia w kilku akapitach, a ich przedstawienie wykracza poza zakres tematyczny niniejszej książki. (Wspomniane operacje matematyczne są jednak bardzo ciekawe i dostarczają wielu informacji na temat przekształceń 2D/3D i ich implementacji w komputerze). Dlatego też na listingu 13.9 po prostu przedstawiam wzór, który będziemy wykorzystywać.

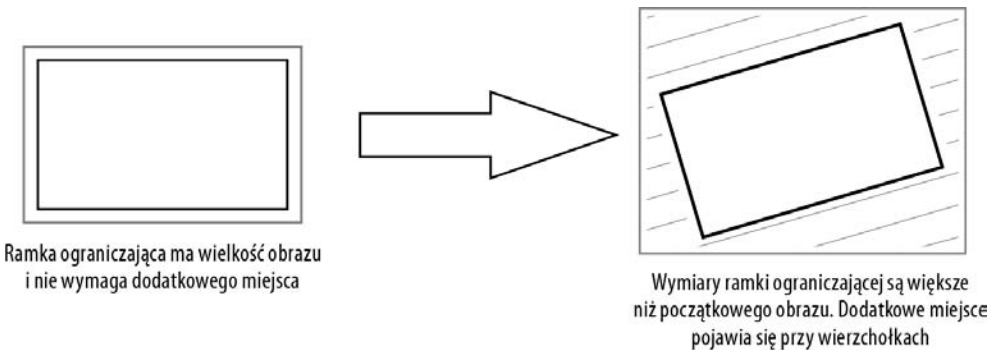
**Listing 13.9.** Wzór obrotu punktu wokół punktu początkowego w systemie współrzędnych kartezjańskich

$$x' = x * \cos(a) - y * \sin(a)$$

$$y' = x * \sin(a) + y * \cos(a)$$

W powyższym wzorze  $(x, y)$  to współrzędne punktu początkowego, natomiast  $(x', y')$  to współrzędne punktu po jego obrocie o zdefiniowany kąt wokół punktu początkowego. Podobnie jak w przypadku większości praktycznych przekształceń 2D/3D stosowany wzór jest zadziwiająco prosty.

Każdy obraz można wpisać w prostokąt, który nazywamy *ramką ograniczającą*. W przypadku obrazu, który nie jest obrócony, ramka ograniczająca będzie miała taką samą wielkość jak wpisany w nią obraz. Jednak po obrocie obrazu o dowolny kąt ramka ograniczająca będzie musiała być większa, co pokazano na rysunku 13.7.



**Rysunek 13.7.** Po obrocie obrazu wielkość ramki ograniczającej zwiększa się

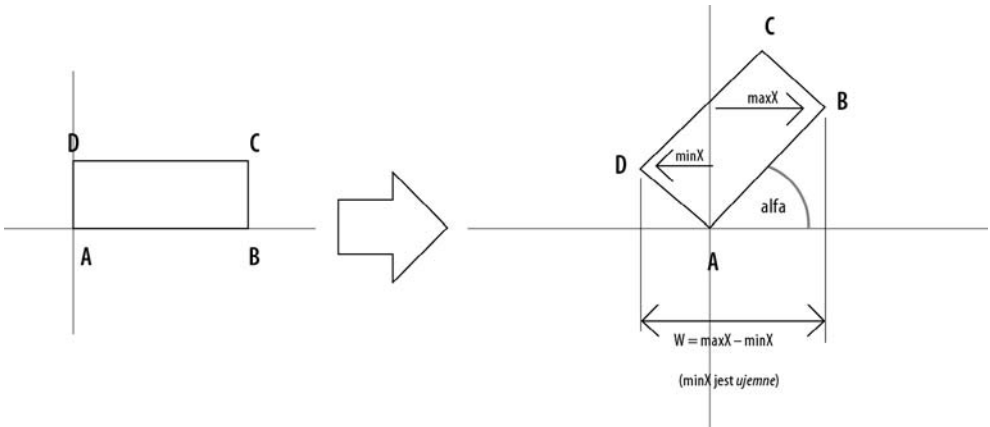
Kolejnym krokiem jest ustalenie wielkości ramki ograniczającej, w którą wpisany jest obrócony obraz. Istnieje kilka sposobów wykonania tego zadania, więc wybierzemy ten, którego użycie ma największy sens (choć niekoniecznie będzie on najszybszy bądź najefektywniejszy).

W celu ustalenia wielkości ramki ograniczającej pierwszym krokiem jest sprawdzenie wielkości ramki ograniczającej początkowego obrazu i obrócenie jej czterech wierzchołków wokół punktu początkowego podobnie jak w przypadku samego obrazu. Otrzymujemy w ten sposób współrzędne czterech punktów; pamiętając o wartościach minimum i maksimum współrzędnych  $x$  i  $y$ , w efekcie dysponujemy wartościami  $\min X$ ,  $\max X$ ,  $\min Y$  i  $\max Y$ . Różnica pomiędzy maksimum i minimum współrzędnej  $x$  jest szerokością ramki ograniczającej, natomiast różnica pomiędzy maksimum i minimum współrzędnej  $y$  jest jej wysokością.

Aby jeszcze bardziej ułatwić sobie operacje matematyczne, jeden z wierzchołków możemy uznać za początek systemu współrzędnych, zamiast zastosować bardziej intuicyjne podejście i za początek systemu współrzędnych uznać centrum ramki ograniczającej. Oznacza to konieczność obliczenia współrzędnych jedynie dla trzech pozostałych wierzchołków, ponieważ wierzchołek początkowy z definicji ma współrzędne  $(0, 0)$ .

Cały proces został pokazany na rysunku 13.8.





**Rysunek 13.8.** Obliczenie wielkości obróconej ramki ograniczającej dla obrazu. Rysunek przedstawia obliczenie szerokości ramki ograniczającej

Po przełożeniu powyższego objaśnienia na rzeczywisty kod otrzymamy kod przedstawiony na listingu 13.10.

**Listing 13.10.** Kod odpowiedzialny za obliczenie wielkości obróconej ramki ograniczającej obraz

```
// Obliczenie wielkości obróconego obrazu.
// W tym celu w pierwszej kolejności przyjmujemy założenie, że lewy dolny wierzchołek to punkt
// o współrzędnych (0,0).
// Następnie obliczamy pozostałe trzy wierzchołki.
double point1x = originalW * Math.cos(angle);
double point1y = originalW * Math.sin(angle);
double point2x = -originalH * Math.sin(angle);
double point2y = originalH * Math.cos(angle);
double point3x = originalW * Math.cos(angle) - originalH * Math.sin(angle);
double point3y = originalW * Math.sin(angle) + originalH * Math.cos(angle);

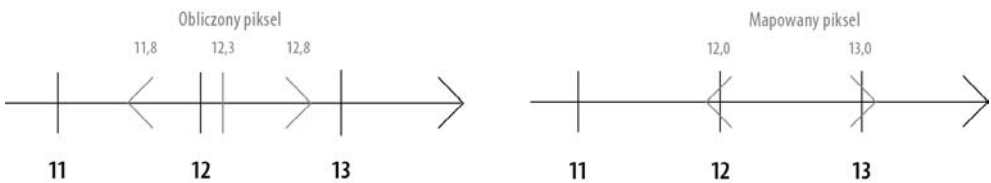
// Kolejnym krokiem jest znalezienie wartości minimum i maksimum współrzędnych wierzchołków.
double minx = Math.min( 0, Math.min(point1x , Math.min(point2x , point3x)));
double miny = Math.min( 0, Math.min(point1y , Math.min(point2y , point3y)));
double maxx = Math.max( 0, Math.max(point1x , Math.max(point2x , point3x)));
double maxy = Math.max( 0, Math.max(point1y , Math.max(point2y , point3y)));

// Ostatnim krokiem jest obliczenie faktycznej szerokości i wysokości obróconego obrazu.
int rotatedW = (int) Math.floor(Math.abs(maxx - minx));
int rotatedH = (int) Math.floor(Math.abs(maxy - miny));
```

Na tym etapie, gdy znamy już wymiary ramki ograniczającej, możemy przystąpić do „przenoszenia” pikseli z obrazu początkowego do obróconego. W tym celu zostanie zastosowane *mapowanie odwrócone*. Oznacza to, że zamiast przechodzenia przez wszystkie piksele obrazu początkowego i sprawdzania, czy będą odpowiednie w obrazie obróconym, następuje przejście przez wszystkie piksele obrazu obróconego i sprawdzenie, które piksele obrazu początkowego (o ile w ogóle) im odpowiadają.

Powód zastosowania takiego podejścia jest prosty: zwykłe mapowanie spowoduje powstanie „dziur” w obróconym obrazie. Wynika to stąd, że piksele zawsze mają współrzędne podawane w liczbach całkowitych, podczas gdy piksele obliczone mają współrzędne wyrażone w liczbach rzeczywistych. To prowadzi do powstania różnic pomiędzy współrzędnymi obliczonymi i rzeczywistymi pikseli.

Aby to zilustrować, przyjmujemy założenie, że po zastosowaniu względem piksela obrazu początkowego wzoru opisującego obrót wartością  $x'$  jest 12,3. Prawidłowe odwzorowanie piksela źródłowego na obróconym obrazie oznacza konieczność „wypełnienia” przestrzeni pomiędzy  $x' = 11,8$  i  $x' = 12,8$ , więc wartość  $x' = 12,3$  znajduje się pośrodku. Jednak w trakcie mapowania piksela w obróconym obrazie wartość 12,3 zostaje zaokrąglona do 12 i następuje wypełnienie przestrzeni pomiędzy  $x' = 12$  i  $x' = 13$ : różnica 0,4 (0,2 + 0,2) pomiędzy obliczonymi i rzeczywistymi wartościami  $x'$  pozostaje niewypełniona. Taka sytuacja została pokazana na rysunku 13.9.



**Rysunek 13.9.** Różnica pomiędzy obliczonymi i mapowanymi współrzędnymi piksela

Jeżeli obraz miałby tylko jeden wymiar, to nie stanowiłoby żadnego problemu, ponieważ puste miejsce zostałoby wypełnione przez inne piksele. Jednak w dwóch wymiarach prowadzi to do sytuacji, w której piksele w obrazie wynikowym pozostałyby niewypełnione, czyli obraz obrócony zawierałby dziury.

**Uwaga!**

Wspomniane „dziury” często pojawiają się w przekształceniach obrazów, których wzory generują współrzędne w postaci liczb rzeczywistych (a nie całkowitych). Aby uniknąć powstawania dziur, w takich przypadkach zawsze powinno być stosowane mapowanie odwrotne.

Poza użyciem mapowania odwrotnego trzeba pamiętać o jeszcze jednym: należy wziąć pod uwagę punkt centralny, wokół którego przeprowadzany jest obrót. W przypadku obrazów wspomnianym punktem centralnym jest środek obrazu, co oznacza, że środek obrazu musi mieć współrzędne (0,0), aby przedstawiony wzór obrotu obrazu funkcjonował prawidłowo.

Jednak Java ME przypisuje współrzędne (0,0) lewemu górnemu wierzchołkowi obrazu. Tę różnicę w mapowaniu współrzędnych trzeba wziąć pod uwagę i odpowiednio dostosować wzór obrotu obrazu. Różnica pomiędzy środkiem obrazu i jego lewym górnym wierzchołkiem wynosi (-szerokość/2, -wysokość/2). Ostateczna postać wzoru została przedstawiona na listingu 13.11.

**Listing 13.11.** Wzory obliczania obrotu po dostosowaniu ich do różnic w mapowaniu

```
x' = (x - szerokość/2) * cos(a) - (y - wysokość/2) * sin(a)
y' = (x - szerokość/2) * sin(a) + (y - wysokość/2) * cos(a)
```

Tak więc zebraliśmy już wszystkie komponenty potrzebne do utworzenia pełnej procedury obrotu obrazu. Jej kod został przedstawiony na listingu 13.12.

**Listing 13.12.** Pełna procedura obrotu obrazu przeznaczona dla platformy Java ME

```
public void drawRotatedImage(Image image, Graphics g, double angle, int x, int y)
{
    // W tym miejscu wstaw kod z listingu 13.8.

    // Obliczenie „punktu początkowego” (w omawianym przykładzie środka) obróconego obrazu.
    int referenceX = rotatedW / 2;
    int referenceY = rotatedH / 2;

    // Zarezerwowanie tablicy na dane piksela obrazu początkowego i obróconego.
    int [] sourceData = new int[originalW * originalH];
    int [] rotatedData = new int[rotatedW * rotatedH];

    // Pobranie pikseli obrazu początkowego.
    image.getRGB(sourceData, 0, originalW, 0, 0, originalW, originalH);

    // Zmienne przeznaczone do oznaczenia położenia X, Y pikseli w obrazach początkowym i obróconym.
    int rotX, rotY;
    int origX, origY;

    // Zmienne przechowujące pozycje indeksu w tablicach RGB.
    int origPos, rotatedPos;

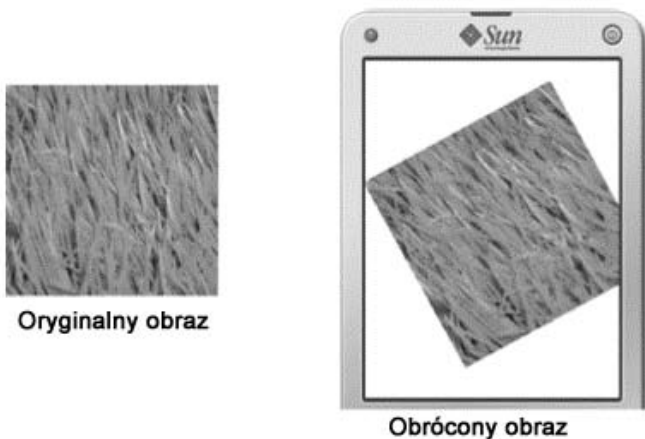
    // Przetworzenie każdego piksela obrazu obróconego.
    for (rotX=0; rotX<rotatedW; rotX++)
    {
        for (rotY=0; rotY<rotatedH; rotY++)
        {
            // Dla bieżącego „obróconego” piksela obliczamy współrzędną
            // X piksela w obrazie początkowym. Dla tej operacji
            // za punkt początkowy przyjmuje się środek
            // tego punktu odniesienia.
            origX = (int) ( (rotX - referenceX) * Math.cos(angle) - (rotY
            - referenceY) * Math.sin(angle) + originalW / 2);
            // Sprawdzenie, czy otrzymana wartość X mieści się w obrazie
            // początkowym, czy nie. Jeżeli tak jest, przechodzimy do kolejnego piksela.
            if ( origX < 0 || origX >= originalW)
            {
                continue;
            }
            // Następnym krokiem jest obliczenie współrzędnej Y.
            origY = (int) ( (rotY - referenceY) * Math.cos(angle) +
            (rotX - referenceX) * Math.sin(angle) + originalH / 2);
            // Obliczenie przyszłej pozycji piksela w tablicy obrazu źródłowego.
            origPos = origY * originalW + origX ;
            // Sprawdzenie, czy pozycja jest prawidłowa.
            // Jeżeli pozycja jest nieprawidłowa, przechodzimy do kolejnego piksela.
            if ( origPos < 0 || origPos >= sourceData.length )
            {
                continue;
            }
        }
    }
}
```

```
    }  
    // Obliczenie pozycji piksela „obróconego” w tablicy obrazu obróconego.  
    rotatedPos = rotY * rotatedW + rotX;  
    // Przeniesienie danych pikseli z tablicy początkowej do tablicy obrazu obróconego.  
    rotatedData[rotatedPos] = sourceData[origPos];  
    }  
}  
// Wyświetlenie wygenerowanego obrazu.  
g.drawRGB(rotatedData, 0, rotatedW, x, y, rotatedW, rotatedH, true);  
}
```

---

Powyższy listing jest pokaźny, ale sam sposób działania procedury jest całkiem prosty. Najbardziej skomplikowane fragmenty, takie jak obliczanie wielkości obróconego obrazu oraz mapowanie odwrotne pikseli (wraz z niezbędnym dostosowaniem współrzędnych), zostały już wcześniej omówione. Jednak w tym miejscu warto zwrócić uwagę na pogrubione wiersze kodu na listingu. Odpowiadają one za sprawdzenie, czy współrzędne mapowanych pikseli mieszczą się w ramach obrazu źródłowego. Jeżeli sprawdzenie zakończy się niepowodzeniem, kod przechodzi do kolejnego piksela obróconego obrazu. Ten krok jest niezbędny, ponieważ, jak mogłeś zobaczyć na rysunku 13.7, niektóre piksele w rogach obróconego obrazu są po prostu pustą przestrzenią, więc w ich przypadku nie ma potrzeby mapowania jakichkolwiek pikseli obrazu źródłowego. Próba ich mapowania spowoduje zgłoszenie wyjątku `ArrayIndexOutOfBoundsException`.

Na rysunku 13.10 pokazano powyższą procedurę w działaniu.



**Rysunek 13.10.** Omówiona procedura obracania obrazu w działaniu

## Zmiana wielkości obrazu

Inną często stosowaną techniką przekształcania obrazu jest *zmiana wielkości obrazu* — to znaczy jego zwiększenie lub zmniejszenie. Zmiana wielkości obrazu jest użyteczna w wielu różnych sytuacjach, począwszy od tworzenia efektu przybliżenia w aplikacji galerii obrazów aż do przeprowadzanej „w locie” zmiany wielkości elementów interfejsu użytkownika.

Domyślnie Java ME nie jest wyposażona w żadne możliwości z zakresu zmiany wielkości obrazu. Na szczęście odpowiednia procedura jest całkiem łatwa do zaimplementowania, ponieważ w zasadzie to jedynie operacje pomnożenia obrazu źródłowego przez określony współczynnik. Jeżeli wartość współczynnika jest mniejsza niż 1,0, obraz zostanie zmniejszony. Natomiast wartość współczynnika większa niż 1,0 oznacza powiększenie obrazu.

Pierwszym krokiem jest ustalenie wielkości zmodyfikowanego obrazu. Jest to łatwe zadanie, ponieważ wystarczy wielkość obrazu początkowego pomnożyć przez określoną wartość współczynnika. Kolejnym krokiem jest wykorzystanie mapowania odwrotnego do sprawdzenia, który piksel obrazu początkowego odpowiada konkretnemu pikselowi obrazu po zmianie jego wielkości. W tym celu wystarczy *podzielić* współrzędne każdego piksela docelowego przez wartość współczynnika.

Cały proces przedstawiono na listingu 13.13.

---

### Listing 13.13. Zmiana wielkości obrazów na platformie Java ME

---

```
public void drawResizedImage(Image image, Graphics g, double factor, int x, int y)
{
    // Zmienne oznaczające pozycje X, Y w obrazie początkowym i zmienionym.
    int xpos, ypos;
    int origx, origy;

    // Pozycje w tablicy RGB.
    int origpos, zoompos;

    // Obliczona wielkość obrazu po zmianie.
    int originalW = image.getWidth();
    int originalH = image.getHeight();
    int zoomW = (int) (originalW * factor);
    int zoomH = (int) (originalH * factor);

    // Zarezerwowanie tablicy na dane pikseli obrazu początkowego i zmienionego.
    int [] sourceData = new int[originalW * originalH];
    int [] zoomData = new int[zoomW * zoomH];

    // Pobranie pikseli obrazu początkowego.
    image.getRGB(sourceData, 0, originalW, 0, 0, originalW, originalH);

    // Przetworzenie każdego piksela w obrazie docelowym.
    for (xpos=0; xpos<zoomW; xpos++)
    {
        for (ypos=0; ypos<zoomH; ypos++)
        {
            // Obliczenie odpowiadającego mu piksela w obrazie początkowym.
            origx = (int) (xpos / factor);
            origy = (int) (ypos / factor);

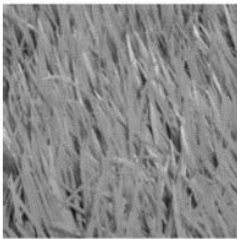
            // Mapowanie obydwu pikseli (początkowy, zmieniony) w tablicach danych.
            origpos = origy * originalW + origx;
            zoompos = ypos * zoomW + xpos;

            // Przeniesienie danych pikseli z tablicy obrazu początkowego do tablicy obrazu zmienionego.
            zoomData[zoompos] = sourceData[origpos];
        }
    }
}
```

```
    }  
}  
  
// Wyświetlenie wygenerowanego obrazu  
g.drawRGB(zoomData, 0, zoomW, x, y, zoomW, zoomH, true);  
}
```

---

Powyższa procedura w działaniu została pokazana na rysunku 13.11.



Obraz początkowy



Obraz po zmianie wielkości

**Rysunek 13.11.** Przykład zmiany wielkości obrazu na platformie Java ME

## Implementacja innych efektów graficznych

Istnieje znacznie więcej efektów graficznych, które mógłbyś zaimplementować. Przykładowo możesz zaimplementować system bazujący na obsłudze cząstek i utworzyć efekt „eksplozji” obrazu na wiele mniejszych fragmentów (lub efekt odwrotny, gdy kompletny obraz powstaje w wyniku połączenia wielu mniejszych fragmentów).

Kluczem i jedynym ograniczeniem w dodawaniu kolejnych efektów graficznych do arsenału dostępnych jest Twoja wyobraźnia oraz sprytnie używanie możliwości oferowanych przez Javę ME. Przykładowo wspomniany wcześniej efekt eksplozji w rzeczywistości nie wymaga ogromnej mocy procesora, ponieważ Twoim zadaniem jest jedynie obliczenie położenia każdego fragmentu obrazu, a następnie użycie obrazu źródłowego jako mapy obrazów podczas wyświetlania każdego fragmentu w odpowiedniej pozycji.

Trzeba również pamiętać, że istnieją dwa typy efektów, które możesz utworzyć: odpowiednie oraz nieodpowiednie do użycia w czasie rzeczywistym. Maskowanie obrazu to doskonały przykład efektu odpowiedniego do zastosowania w czasie rzeczywistym, ponieważ ta operacja jest prosta i szybka. Jednak jeśli spróbujesz zaimplementować filtry obrazu (na przykład filtr Watercolor znany z programu Photoshop), prawdopodobnie nie będziesz mógł ich używać w czasie rzeczywistym. To oczywiście nie oznacza, że nie można dla nich znaleźć zastosowania. Przykładowo personalizacja aplikacji na podstawie własnych obrazów użytkownika jest mile widzianą funkcją, a w takim przypadku filtry mogą się okazać przydatne.

Implementacja własnych efektów graficznych jest zawsze cennym doświadczeniem. Praca z pikselami dostarcza radości, a programista może przy okazji poznać wiele zagadnień związanych z grafiką komputerową, optymalizacją kodu, technikami programowania oraz operacjami matematycznymi stosowanymi w trakcie tego procesu. Ponadto efekty uzyskane za pomocą samodzielnie utworzonego kodu mogą zadziwić Twoich przyjaciół, współpracowników, a nawet szefa.

Poniżej wymieniono kilka efektów graficznych, których implementację możesz rozważyć:

- wspomniany już wcześniej system bazujący na cząstkach;
- rozmywanie i wyostrażanie obrazu;
- antyaliasing obrazu;
- efekty zniekształcania obrazu (na przykład obraz wyświetlany jako trapez lub trzepocząca flaga);
- desaturacja obrazu (zamiana obrazu kolorowego na czarno-biały);
- filtry artystyczne, takie jak Watercolor i Bas Relief;
- morfing obrazu (płynne przejście między zawartością dwóch obrazów, a nie ich pikseli — na przykład zamiana twarzy małżonki na twarz męża). Warto pamiętać, że jest to filtr wyjątkowo trudny do implementacji.

## Połączenie kilku efektów graficznych

Jeżeli zostaną prawidłowo zoptymalizowane, efekty przedstawione w rozdziale można ze sobą łączyć, osiągając w ten sposób jeszcze bardziej spektakularne efekty. Przykładowo istnieje możliwość utworzenia obróconej i teksturowanej wersji logo Twojej firmy, w której tekstura powstaje poprzez zmieszanie dwóch obrazów. Takie rozwiązanie będzie efektywniejsze po zróżnicowaniu parametrów stosowanych tutaj efektów (na przykład zmiana kąta obrotu obrazu oraz współczynnika przezroczystości w operacji mieszania obrazów) i utworzeniu na tej podstawie animacji.

Na tym etapie możesz się zastanawiać, czy jest to w ogóle wykonalne. Jeszcze kilka lat temu zadawałem sobie to samo pytanie. Dlatego też rozpocząłem prace nad utworzeniem biblioteki efektów graficznych dla Javy ME, która byłaby zarówno szybka, jak i oferowałaby potężne możliwości. Zastanawiałem się, ile pod tym względem można „wycisnąć” z Javy ME, i byłem całkowicie zadowolony z wyników uzyskanych za pomocą utworzonej biblioteki (J2ME Army Knife).

Najbardziej zaskakujący (przynajmniej dla mnie) okazał się poziom, do którego można zoptymalizować szybkość działania efektów graficznych. Przykładowo po optymalizacji kodu odpowiedzialnego za obracanie obrazu obraz o wielkości 64×64 pikseli mógł być obracany prawie sto razy na sekundę w telefonie Nokia E50. Najzabawniejsze jest, że od ponad dwóch lat (co jest wiecznością podczas tworzenia oprogramowania) nie prowadziłem żadnych poważnych prac nad tą biblioteką, a pomimo to nadal otrzymuję wiadomości e-mail od użytkowników zaskoczonych szybkością jej działania i oferowanymi możliwościami.

### Uwaga!

Więcej informacji na temat biblioteki J2ME Army Knife znajduje się w witrynie <http://www.j2mearmyknife.com/>. W wymienionej witrynie można zobaczyć bibliotekę w działaniu (demo) oraz pobrać ją. Biblioteka w postaci binarnej jest całkowicie bezpłatna, nawet dla projektów komercyjnych, więc warto ją wypróbować!

Połączenie wielu efektów razem jest jeszcze bardziej wykonalne w sytuacjach niewymagających dynamiki. Wprawdzie pojedyncza klatka animacji powinna być obliczona w ciągu maksymalnie 100 milisekund, aby uzyskać efekt „płynności”, ale w przypadku grafiki statycznej ten czas może wynosić nawet 500 milisekund (być może nawet dłużej, jeżeli przetwarzanie odbywa się w oddzielnym wątku w tle). Oznacza to, że można pracować nawet z większymi obrazami źródłowymi lub łączyć ze sobą większą liczbę efektów.

Inna technika oferująca potężne możliwości to użycie dynamicznych efektów graficznych w połączeniu z mapą obrazów. Powróćmy do wspomnianego wcześniej przykładu animacji tekstuowanego logo. Zamiast za każdym razem generować każdą klatkę animacji, klatki mogą być wygenerowane jedynie za pierwszym razem, a następnie umieszczone w mapie obrazów. Następnie w trakcie trwania animacji zamiast ponownie generować klatki, wykorzystujesz dane przechowywane w mapie obrazu. Ogólnie rzecz biorąc, to rozwiązanie charakteryzuje się dużą szybkością działania, ale wymaga całkiem sporej ilości pamięci RAM w przypadku używania dużych obrazów. Wyniki osiągnęte podczas stosowania tej techniki będą znacznie różniły się w zależności od wykorzystywanego urządzenia docelowego.

Wreszcie łączenie efektów graficznych można przeprowadzić na dwa sposoby. Pierwszym jest zastosowanie każdego efektu oddzielnie: dane wyjściowe pierwszego efektu stają się danymi wejściowymi dla kolejnego itd. Wprawdzie takie rozwiązanie jest elastyczne, ponieważ efekty pozostają samodzielne, ale wiąże się z ogólnie większym obciążeniem dla urządzenia. Przykładowo, jeżeli dwa efekty z rzędu będą musiały pracować z poszczególnymi wartościami kanałów alfa oraz kolorów czerwonego, zielonego i niebieskiego dla każdego piksela, wówczas każdy efekt będzie musiał oddzielnie przeprowadzać wyodrębnianie wymienionych wartości.

Drugim sposobem jest utworzenie efektów ostatecznych. Przykładowo operacje obrotu i zmiany wielkości można ze sobą połączyć. Zaletą takiego podejścia jest wyeliminowanie niepotrzebnego obciążenia, ponieważ wiele obliczeń wymaganych w obydwóch efektach zostanie przeprowadzonych tylko jednokrotnie. Ponadto można wówczas ponownie wykorzystywać obiekty znajdujące się w pamięci, takie jak tablice wartości ARGB. Wzrost wydajności będzie szczególnie widoczny, jeśli wyeliminowanie obciążenia związanego z efektami nastąpi na poziomie piksela (to znaczy w wewnętrznej pętli, w której są przeprowadzane operacje na poszczególnych pikselach).

## Podsumowanie

W tym rozdziale omówiliśmy zaawansowane techniki graficzne na platformie Java ME: mieszanie obrazów, maskowanie obrazów, zmianę wielkości obrazów i obracanie obrazów. Przedstawione zostały reguły dotyczące wymienionych technik oraz różne przykłady ich wykorzystania. W kolejnym rozdziale zaprezentuję kilka opowieści dotyczących programowania na platformie Java ME oraz wnioski z nich płynące.



# Skorowidz

2G, 32  
3G, 32, 106

## A

abstrakcja API, 239  
addWidget(), 138  
algorytm Levenshteina do obliczania edycji odległości, 282  
AMR, 26  
Android, 21  
antyaliasing obrazu, 329  
API 2D, 309  
API File Connection, 333  
API Location, 224  
API Reflection, 302  
API Wireless Messaging, 226  
aplikacje  
  amatorskie, 33  
  profesjonalne, 33, 34  
App Store, 294  
app.files, 48  
app.media, 48  
app.models, 48  
app.module.<NAZWA>, 48  
app.module.<NAZWA>.classes, 48  
app.module.<NAZWA>.controllers, 48  
app.module.<NAZWA>.helpers, 48  
app.module.<NAZWA>.managers, 48  
app.module.<NAZWA>.models, 48  
app.module.<NAZWA>.views, 48  
Apple, 294  
Application, 59  
APPLICATION\_EXIT, 192  
APPLICATION\_START, 190, 192  
arraycopy(), 298  
ArrayIndexOutOfBounds, 28, 326  
assign(), 53, 57  
automatyczne uzupełniania, 283

## B

backBuffer, 100  
Bas Relief, 329  
BaseContainerManager, 148  
BaseContainerWidget, 148  
BaseWidget, 146  
BEGIN\_LOGIN, 88, 90  
biblioteka  
  instalowanie, 81  
  konfigurowanie, 81  
  UI, 134  
Big O, 247, 264  
Blackberry, 21, 28  
Bootstrap, 59  
borders, 136  
ByteArrayOutputStream, 120  
ByteDeserializer, 118, 120  
ByteRecordReader, 121  
ByteRecordWriter, 121  
ByteSerializer, 118

## C

CallbackHandler, 145  
CameraFrameProvider, 51  
Cell Broadcast Service, 226  
checksum(), 54  
ciąg Fibonacciego, 270, 271  
CLDC 1.0, 213  
ClipHelper, 141  
clipping rectangle, 138  
ClipRect, 140  
CLOSED, 232  
cloud, 112  
com.apress.framework.objecttypes, 49  
CommandAction(), 57  
CommandListener, 57  
common objects, 54

Consumer, 49  
consumers, 52  
ContactIntegrityManager, 52  
ContactSyncManager, 52  
Container, 137, 142  
ContentWidth(), 136  
Controller, 49  
controllers, 54  
Controls, 232  
CPU, central processing unit, 30  
createFromHashtable(), 84  
createFromRawBytes(), 84  
currentEventCount(), 62

## D

DataInputStream, 181, 211  
DataSources, 232  
dateOfBirth, 260  
deallocate(), 233  
desaturacja obrazu, 329  
Deserializer, 117  
Display, 59  
dławnienie komunikacji, 107  
doCallback(), 145  
doInit(), 260  
doLayout(), 149  
dostawcy danych, 51  
Dostawcy usług pamięci trwałej, 114  
dostawcy usługi pamięci trwałej, 122  
drawRGB(), 28  
dziedziczenie, 39

## E

EDGE, 106  
efekty zniekształcania obrazu, 329  
ekran, 196, 220  
elastyczność aplikacji, 36  
elastyczność kodu, 35  
element osiowy, 266  
eliminowanie niepotrzebnej rekurencji, 270  
emulator WTK, 26  
Enough Software J2ME Polish, 242  
EnterPassword, 145  
EOFException, 183  
EvenetControllerManager, 58  
Event, 48  
event controller managers, 50

event controllers, 50  
event listeners, 50  
Event.Environment.LOW\_MEMORY, 43  
EventController, 48, 55, 56, 57  
EventControllerManager, 57  
EventListener, 48  
EventManagerThreads, 59  
EVT, 65, 208

## F

FILL\_IN\_THE\_BLANKS, 63  
fillArc(), 311  
fillPolygon(), 333  
filtry artystyczne, 329  
fireEvent(), 164  
firmware, 225  
FlowController, 190, 191  
form, 196  
formatDate(), 187  
formularz, 196  
    ekranu głównego, 200  
    ekranu powitalnego, 196  
    ekranu ustawień konfiguracyjnych, 204  
forwardBuffer, 100  
FPS, 216  
fragmentacja  
    API, 214  
    możliwości, 214, 224  
    sprzętowa, 214, 215  
    urządzenia, 213  
Frame Per Second, 216  
framework.common, 48  
framework.core, 48  
framework.helpers, 48  
framework.objecttypes, 48

## G

GameCanvas, 54  
GameCanvasView, 165  
garbage collector, 31  
Garbage Collector, 216, 260, 262  
geolokalizacja, 214  
getAngleForTile(), 41  
getContentWidth(), 136  
getCurrentResult(), 116  
getFirst(), 79  
getLast(), 79

getMyProfile(), 103  
 getNumberOfAvailableObjects(), 52  
 getPreferredContentWidth(), 136  
 getProfileFor(), 103  
 getter, 260  
 getTimelineForFilter(), 92  
 getYScroll(), 167  
 GIF, 312  
 goBack(), 79  
 goForward(), 79  
 GPRS, 106  
 grafika SVG, 26  
 Graphics.translate(), 228  
 GraphicsWrapper, 229

## H

HANDLE\_PAINT, 195  
 handleEvent(), 56, 90  
 Hashtable, 181, 259  
 hashtag, 77  
 hasMore(), 52  
 hasMoreEvents(), 62  
 heapsort, 268  
 Hessian, 222  
 hideNotify(), 233  
 HighLevelSerializer, 127  
 home timeline, 78  
 HorizontalContainer, 154  
 HorizontalManager, 158  
 HTMLParser, 35  
 HttpConnection, 301  
 HTTPS, 106

## I

Image.getRGB(), 310, 314  
 IMEI, 25  
 implementacja uwierzytelniania zetonowego, 90  
 indexOf, 259  
 INITIATE\_SHUTDOWN, 192  
 InputStream, 211  
 InputStringItem, 163, 164  
 insertWidget(), 138  
 integer, 318  
 inteligentna reprezentacja danych, 79  
 interfejs, 39
 

- CallbackHandler, 145
- Container, 137
- Controller, 54

Deserializer, 117  
 Manager, 53  
 Model, 71  
 PersistenceProvider, 114  
 Provider, 51  
 RecordReader, 116, 121  
 RecordWriter, 116, 121  
 Serializer, 117  
 Timeline, 78  
 TwitterServer, 75  
 UITheme, 143  
 View, 54  
 International Mobile Equipment Identity, 25  
 IOException, 183  
 iOS, 21  
 isFocusable(), 149

## J

J2ME Army Knife, 329  
 J2ME Polish, 242, 243  
 J2SE, 259  
 JAR, 26  
 Java ME, 21
 

- budowanie aplikacji, 24
- lokalizacja, 32
- wady, 22

 Java Specification Request (JSR), 22, 112  
 javac, 245  
 Jazelle, 30  
 JUnit, 300  
 JPEG, 312  
 JSON, 32, 108, 219  
 JSR, 22  
 JSR 75, 22, 112, 225  
 JSR 184, 22  
 JSR 238, 176  
 JSR 256, 22  
 jumpTo(), 51  
 JUnit, 300  
 Just-In-Time, 245

## K

kinetic scrolling, 222  
 klasa
 

- Application, 59
- BaseWidget, 146
- Bootstrap, 64

## klasa

- Defaults, 195
- EventController, 55
- EventManagerThreads, 61
- EVT, 65, 208
- FlowController, 191
- GameCanvasView, 165
- HorizontalManager, 158
- InputStringItem, 163
- Locale, 181
- MainForm, 200
- MainScreenController, 202
- Player, 67
- ServerImplementation, 85, 86
- SettingsForm, 204
- SettingsScreenController, 206
- SimpleTextButton, 158
- StringItem, 160
- TestModel, 66
- TimelineHome, 93
- TimelineUserTweets, 99
- Tweet, 73
- TweetsController, 193
- TwitterUser, 74
- UserCredentials, 76
- WelcomeForm, 196
- WelcomeScreenController, 197

## klasy, 59

- fikcyjne, 301
- konkretne widżetów, 154

## klawisz zwolniony, 148

## Knuth Donald E., 264

## kod osadzony, 254

## kompilator

- javac, 245
- JIT, 245

## kompozycje, 143

## kontenery, 137

## kontroler, 53, 54

- ekranu głównego, 200
- ekranu powitalnego, 196
- ekranu ustawień konfiguracyjnych, 204
- przepływu sterowania, 190
- wiadomości tweet, 193
- zdarzeń, 50

**L**

## LIFO (last in, first out), 284

## Lightweight UI Toolkit (LWUIT), 242

- loadFromDataInputStream(), 182

- locale, 175

- localization, 175

- login(), 88

- LOGIN\_FAILED, 88

- LOGIN\_SUCCEEDED, 88

- loginUsingTokens(), 89

- loginUsingUnPw(), 89

- logowanie, 87, 91

- lokalizacja, 175

- loop peeling, 257

- loop unswitching, 256

**M**

- MainForm, 200

- MainScreenController, 200, 202

- manager, 52

- Manager, 49, 53

- mapowanie odwrócone, 323

- marginesy

- wewnętrzne, 136

- zewnętrzne, 136

- margins, 136

- maska obrazu, 314

- maskowanie obrazu, 314

- menedżer, 52

- kontrolerów zdarzeń, 57

- łączenia, 285

- kontrolerów zdarzeń, 50

- mergesort, 268

- metoda login(), 88

- microedition.platform, 225

- MicroEmulator, 302, 303

- MIDlet, 59, 231

- MIDlety, 353

- MIDP 1.0, 213, 310, 357

- MIDP 2.0, 310, 353

- MIDP 3.0, 353, 354

- MMAPI, 224

- Mobile Internationalization API, 176

- MockHttpConnection, 301

- Model, 49, 71

- modele, 53

- models, 53

- moduł

- interfejsu użytkownika, 133

- pamięci trwałej, 113, 130

- UI, 134

morfing obrazu, 329  
 motywy, 143  
 MP3, 26  
 MVC, 48

## N

NetBeans, 300  
 next(), 51  
 niemodyfikowalne typy danych, 72  
 niezgodność API  
   architektury lub niezgodność globalna, 229  
   globalna, 231  
   lokalna, 228, 229  
   swoboda w interpretacji, 229  
 NullPointerException, 195

## O

obiekt nadrzędny, 137  
 obiekt z puli, 261  
 obiekty, 59  
   deserializujące, 114, 117  
   serializujące, 114, 117  
   wysokiego poziomu, 84  
 obracanie obrazów, 321  
 obramowania, 136  
 obserwatory zdarzeń, 50  
 obsługa  
   historii, 283  
   interakcji z użytkownikiem, 144  
   motywów, 293  
   tekstu adaptacyjnego, 281  
 odbiorcy, 52  
 odświeżanie pamięci, 31  
 onPressed(), 56, 57  
 onFocus(), 149  
 onLostFocus(), 149  
 open source, 242  
 optymalizacja algorytmu, 247  
 optymalizacja kodu, 245, 246, 248  
   eliminowanie iteracji specjalnych z pętli, 256  
   kod osadzony, 254  
   optymalizacja dostępu do pamięci, 263  
   optymalizacja operacji matematycznych, 250  
   optymalizacja pętli wykonujących operacje matematyczne, 254  
   porównywanie algorytmów, 264  
   pozostawianie przy podstawach, 259

przełączanie ścieżki wykonywania kodu, 248  
 rezygnacja z użycia pętli, 253  
 rozbieżność pętli, 257  
 techniki optymalizacji algorytmu, 263  
 unikanie powielania kodu, 249  
 unikanie tworzenia niepotrzebnych obiektów, 259  
 unikanie wysokiego poziomu funkcji języka, 258  
 usprawnianie algorytmów, 266  
 usunięcie warunków z pętli, 256  
 wykorzystanie zalet płynących z lokalizacji, 249  
 OutOfMemory, 298  
 OutputStream, 260

## P

padding, 136  
 paint(), 149, 158  
 panes, 170  
 pauseApp(), 233  
 PDFParser, 35  
 persistence providers, 114  
 PersistenceProvider, 114, 122  
 Person, 260  
 Photoshop, 328  
 PictureButton, 138  
 PIMContactProvider, 51  
 PNG, 26, 312  
 pojemniki, 137  
 polimorfizm, 39  
 położenie  
   bezwzględne, 135  
   względne, 135  
 porównywanie algorytmów, 264  
 prawie dopasowane, 283  
 PREFETCHED, 232  
 preprocessor, 236  
 processor, 215  
 processEvent(), 56  
 processing of events, 55  
 processNextEvent(), 56, 57  
 programowanie defensywne, 36, 39  
 prostokąt obcinania, 138  
 Provider, 48, 51  
 providers, 51  
 przeciąganie wskaźnika, 148

**Q**

queueEvent(), 56  
quicksort, 266

**R**

RAM, 218  
ramka ograniczająca, 322  
readUTF(), 222  
REALIZED, 232  
RECEIVED\_TWEET, 195  
Record Management System, 111  
RecordReader, 114, 116  
RecordWriter, 114, 116  
rekurencja, 270  
REQUEST\_MAIN\_TWEETS\_BATCH, 195  
REQUEST\_PAINT, 195  
retriveCurrent(), 51  
rgbData, 28  
RMS, 219  
RMSPersistenceProvider, 122  
rozmywanie obrazu, 329

**S**

screen, 196  
scrollToWidget(), 150  
serializacja, 334  
Serializer, 117  
serializeUserCredentials(), 127  
ServerImplementation, 85, 86, 87  
setContentWidth(), 136  
setter, 260  
SettingsForm, 204  
SettingsScreenController, 204, 206  
SeverImplementation, 103  
showNotify(), 233  
showTextBox(), 164  
SimpleTextButton, 158, 164  
sleep mode, 106  
softkeys, 169, 240, 307  
sortowanie, 266  
    przez kopcowanie, 268  
    przez scalanie, 268  
    szybkie, 266  
sprite, 311  
stan  
    jest aktywny, 135  
    ma fokus, 135

    nie jest aktywny, 135  
    nie ma fokusu, 135  
standardowe obiekty, 55  
STARTED, 232  
STOCK\_HIT\_ROCK\_BOTTOM, 86  
store(), 52  
StringItem, 160  
suma kontrolna, 54  
super.handlePointerDragged(), 158  
Symbian, 21  
synchronizacja danych pomiędzy urządzeniami,  
    286  
synchronize, 261  
system  
    informacji zwrotnych, 280  
    zarządzania zapisami, 111  
    zdarzeń, 46  
System.arraycopy(), 298  
System.getProperty(), 226

**T**

TabbedContainer, 138  
TabContainer, 138  
Table Of Contents, 125  
TestModel, 66  
testowanie kodu, 124  
testowanie uprawnień, 307  
theme, 143  
throttling, 107  
ThumbEE, 30  
timeline, 78  
Timeline, 78  
TimelineHome, 93, 98, 99  
TimelineUserTweets, 99  
timestamp, 105  
toByteArray(), 120  
TokenCredentials, 87  
trwały zapis danych, 111  
tryb  
    offline, 109  
    uśpienia, 106  
Tweet, 72, 73  
TweetFilter, 77  
TweetsController, 193  
Twitter, 24  
TwitterServer, 75, 194  
TwitterUser, 74

**U**

UITextHelper.strReplace(), 187  
 UITheme, 143  
 unassign(), 57  
 UNREALIZED, 232  
 UserCredentials, 76, 87  
 UsernamePasswordCredentials, 87  
 usprawnianie
 

- algorytmów, 266
- interakcji z użytkownikiem, 287

 ustawienia regionalne, 175  
 uwierzytelnianie żetonowe, 90

**V**

Vector, 259  
 verifyCredentials(), 90  
 VerticalContainer, 137, 154, 165  
 view, 54  
 View, 49, 54, 142

**W**

Watercolor, 329  
 WAV, 26  
 WelcomeForm, 196  
 WelcomeScreenController, 192, 196, 197  
 wiadomości
 

- pobieranie, 92
- wysyłanie, 91

 widget, 135  
 Widget, 142  
 widok, 54, 142  
 widżet, 135  
 wizualny proces usuwania błędów, 303

wskaźnik
 

- wciśnięty, 148
- zwolniony, 148

 WTK, 26  
 WTK 2.5.2, 111  
 WTK 3.0, 111  
 wykrywanie intencji, 284  
 wyostrzanie obrazu, 329  
 wyszukiwanie, 269  
 wzorzec Factory (Fabryka), 39

**X**

xAuth, 76  
 XML, 32, 108

**Y**

YAML, 108

**Z**

zastępowanie zasobów, 40  
 zestaw brudnych sztuk, 307  
 znaczniki czasu, 105

**Ź**

źródła błędów, 210
 

- alokowanie pamięci dla dużych bloków danych, 210
- błędne lub uszkodzone dane wejściowe, 211
- operacje zewnętrzne, 210
- problemy z pracą współbieżną, 211





# 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**

**P**onad 80% dostępnych obecnie telefonów komórkowych współpracuje z platformą Java ME. Java ME jest okrojona (Micro Edition) wersją popularnego języka Java. Jeżeli chcesz przygotować uniwersalną aplikację, działającą na większości telefonów, ta platforma powinna być Twoim naturalnym wyborem. Korzystając z wygodnego i powszechnie znanego języka oraz licznych narzędzi wspomagających pracę, błyskawicznie osiągniesz swój cel. Jednak zanim przystąpisz do działania, musisz poznać możliwości i ograniczenia Javy ME oraz nauczyć się tworzyć wysokiej jakości kod.

Z tą książką to nic trudnego. Omawia ona wszystkie zagadnienia związane z profesjonalnym wytwarzaniem aplikacji dla platformy Java ME. W trakcie lektury dowiesz się, jak podzielić aplikację na moduły nadające się do ponownego użycia, jak utrzymywać dane oraz jak testować kod. Ponadto opanujesz techniki optymalizacji kodu, tworzenia atrakcyjnego interfejsu użytkownika oraz rysowania zaawansowanych komponentów graficznych. Na koniec będziesz mógł zapoznać się z wizją przyszłości dla platformy Java ME, która mimo inwazji platform Android, iOS i Windows Phone wcale nie jest taka ponura! Książka ta dostarczy mnóstwo przydatnej wiedzy średnio zaawansowanym i zaawansowanym programistom Java ME.

#### Dowiedz się:

- co odróżnia platformę przenośną od platformy biurkowej
- w jaki sposób projektować i implementować aplikację Javy ME
- jak rozwiązywać problemy pojawiające się przy tworzeniu oprogramowania na platformie Java ME
- jakie są poprawne techniki tworzenia aplikacji Javy ME
- jak przeprowadzać optymalizację kodu oraz opracowywać wysokiej jakości aplikacje

**Poznaj nowoczesne podejście do programowania na bazie najnowszych wersji najnowsze wersje platformy Java ME i zbuduj od podstaw w pełni funkcjonalną aplikację!**

**helion.pl**  
księgarnia  
internetowa

Nr katalogowy: 7849



Księgarnia internetowa  
<http://helion.pl>



Zamówienia telefoniczne:  
**0 801 339900**



**0 601 339900**



**Helion**

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

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

sięgnij po **WIĘCEJ!**



KOD KORZYŚCI

ISBN 978-83-246-3589-4



Cena: 59,00 zł

Informatyka w najlepszym wydaniu