Popularna platforma — profesjonalne aplikacje!

Java ME

Tworzenie zaawansowanych aplikacji na smartfony

Ovidiu Iliescu

Apress®

Helion

Tytuł oryginału: Pro Java ME Apps

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

ISBN: 978-83-246-3589-4

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)

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.

Spis treści

ROZDZIAŁ
13

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 osiągana 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 niezmienione 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.

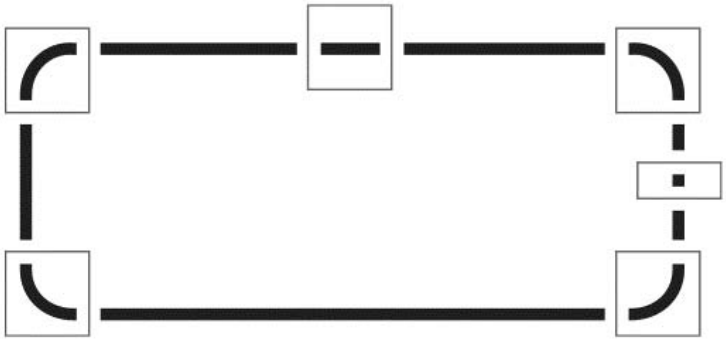
Używanie przygotowanej wcześniej grafiki

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 obrazu żądanego 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 niezmienne 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 o 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.

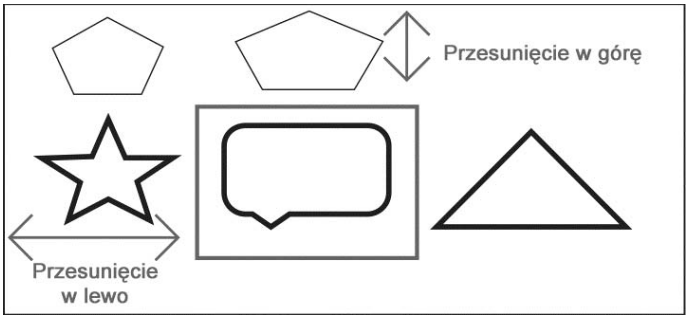
Czasami można nawet przygotować wcześniej całe dane wyjściowe efektu graficznego, zwłaszcza gdy z natury nie są one dynamiczne (na przykład danymi wyjściowymi nie jest klatka animacji). Przypuśćmy, że chcesz wykorzystać obraz maski do utworzenia logo Twojej firmy z teksturą nieba. Taki efekt można wygenerować 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 Dobrą strategią jest mieszanie grafiki dynamicznej i wcześniej przygotowanej. Przykładowo, jeżeli tworzysz grę bazującą na elementach sprite, które muszą obracać się w szesnastu klatkach, 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 żądanego 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 nieprzezroczyste spowodują utworzenie całkowicie nieprzezroczystych pikseli w obrazie docelowym, natomiast przezroczyste i półprzezroczyste piksele maski utworzą przezroczyste i półprzezroczyste 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`.

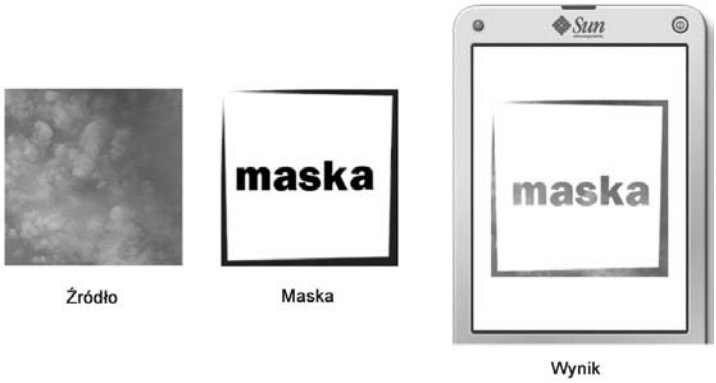
Listing 13.2. Operacja maskowania obrazu w Javie ME

```
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 `ArrayIndexOutOfBounds`. 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 „prześwitywanie" 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 przekształceń 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 wyłączenie 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
```

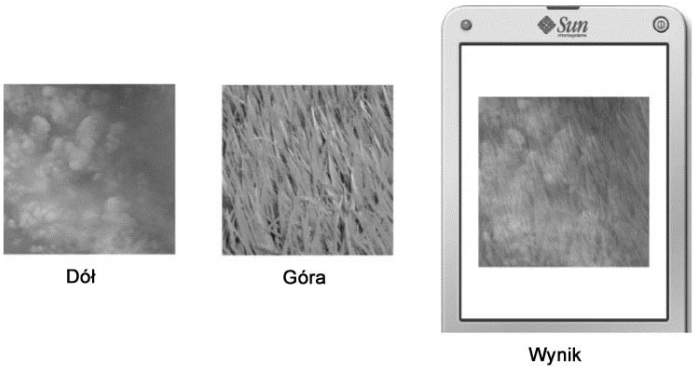
Ponieważ jest to operacja przeprowadzana „na poziomie piksela", powyższy wzór musi być zastosowany względem wszystkich pikseli obrazu wynikowego. Odpowiedzialny za to kod Javy ME został przedstawiony na listingu 13.5, na którym `coeff` oznacza współczynnik przezroczystości w zakresie od 0 do 255.

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 (góra, 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);

        // 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

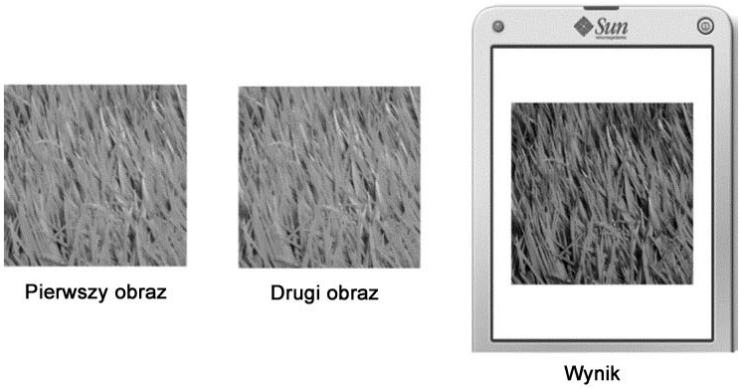
```
wynik = ( kolor1 * kolor2 ) / 255
```

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 (góra, 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 64×64 lub 32×128 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.

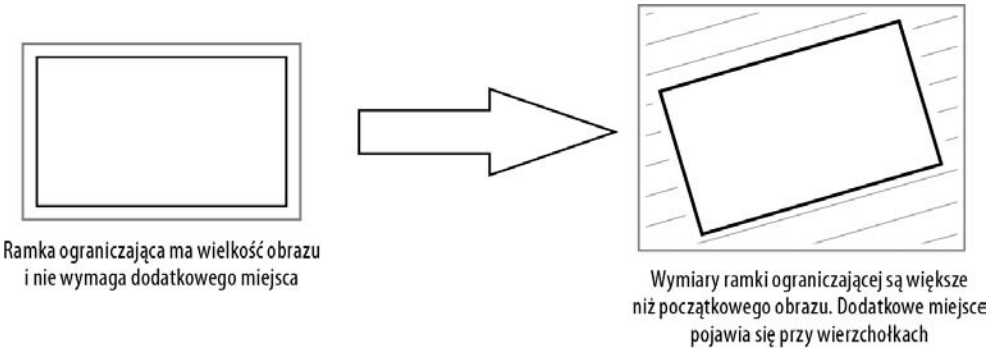
W celu zaimplementowania własnego rozwiązania przede wszystkim trzeba więc zdefiniować wzór obrotu punktu wokół punktu początkowego w systemie współrzędnych kartezjańskich. Zaangażowane w to operacje matematyczne są trudne do wyjaśnienia w kilku akapitach, a ich przedstawienie wykracza poza zakres tematyczny 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 obróceniu 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 obróceniu obrazu o dowolny kąt ramka ograniczająca będzie musiała być większa, co pokazano na rysunku 13.7.



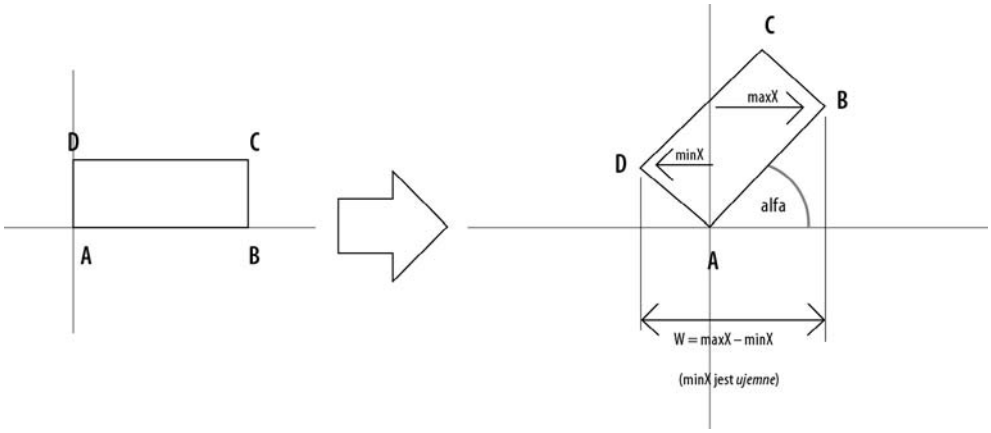
Rysunek 13.7. Po obróceniu obrazu wielkość ramki ograniczającej zwiększa się

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 `minX`, `maxX`, `minY` i `maxY`. 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.

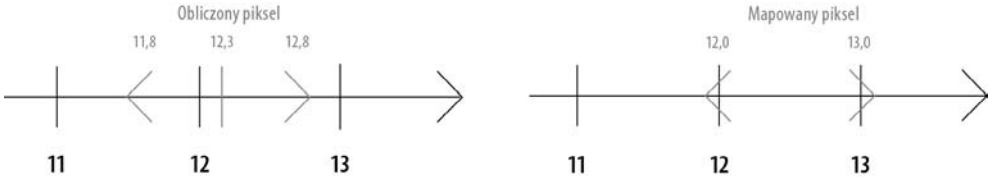
Listing 13.10. Kod odpowiedzialny za obliczenie wielkości obróconej ramki ograniczającej obraz

```
// 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.

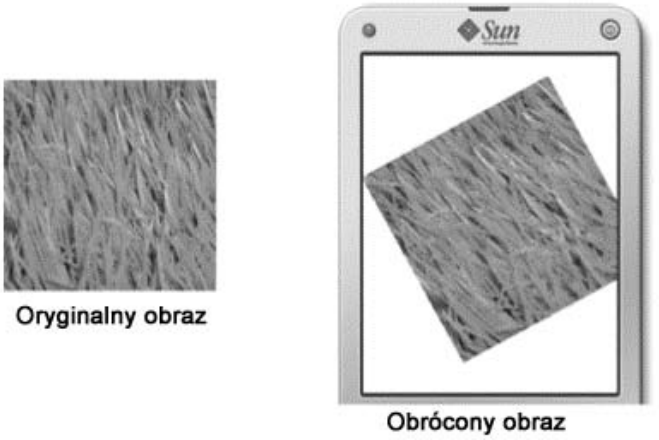
Listing 13.12. Pełna procedura obrotu obrazu przeznaczona dla platformy Java ME

```
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 `ArrayIndexOutOfBounds`.

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.

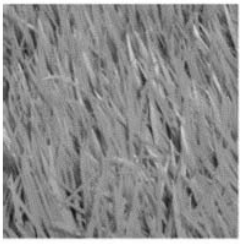
Cały proces przedstawiono na listingu 13.13.

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.getWidth();
   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 sprytne 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 wyostrzanie 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 teksturowanego 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 osiągane 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

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

Ź

Ponad 80% dostępnych obecnie telefonów komórkowych współpracuje z platformą Java ME. Java ME jest okrojoną (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 utrwalać 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

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
http://helion.pl

ISBN 978-83-246-3589-4

Cena: 59,00 zł