

IDŹ DO

PRZYKŁADOWY ROZDZIAŁ



SPIS TREŚCI

KATALOG KSIĄŻEK

KATALOG ONLINE

ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK

DODAJ DO KOSZYKA

CENNIK I INFORMACJE

ZAMÓW INFORMACJE
O NOWOŚCIACH

ZAMÓW CENNIK

CZYTELNIA

FRAGMENTY KSIĄŻEK ONLINE

Efekty graficzne i animowane dla aplikacji Desktop Java. Tworzenie atrakcyjnych programów

Autor: Chet Haase, Romain Guy

Tłumaczenie: Paweł Gonera

ISBN: 978-83-246-1462-2

Tytuł oryginału: [Filthy Rich Clients: Developing Animated and Graphical Effects for Desktop Java\(TM\) Applications \(The Java Series\)](#)

Format: 180x235, stron: 584



Poznaj tajniki tworzenia efektywnych, a jednocześnie funkcjonalnych aplikacji, które przyciągają rzesze klientów

- Jak tworzyć obrazy i wykorzystywać je w aplikacjach?
- Jak sterować działaniem animacji?
- Jak korzystać z grafiki pulpitu dla języka Java?

Informatyka jest dziedziną, która rozwija się w naprawdę szalonym tempie, a użytkownicy komputerów i konsumenci stawiają coraz większe wymagania wszelkim jej produktom. Oczywiście, to, co atrakcyjne, przyciąga, ale równie istotna jest łatwość korzystania z produktu czy intuicyjność jego użytkowania. Współczesny klient oczekuje takiego właśnie idealnego połączenia. Jak tworzyć funkcjonalne, a jednocześnie efektowne aplikacje? Co powoduje, że klienci są zachwyceni i bawią się, używając aplikacji? O tym właśnie jest książka, którą trzymasz w rękach.

W książce „Efekty graficzne i animowane dla aplikacji Desktop Java. Tworzenie atrakcyjnych programów” autorzy w przystępny, a czasami zabawny sposób opisują dostępne technologie do tworzenia bogatych aplikacji. Czytając ją, nauczysz się, jak wykorzystywać grafikę i animację oraz w jakie efekty wyposażać interfejs użytkownika, aby był naprawdę atrakcyjny. Z tego podręcznika dowiesz się wszystkiego na temat podstawowych mechanizmów języka Java, Swing, Java 2D czy graficznych interfejsów użytkownika (GUI). Poznasz techniki tworzenia aplikacji z bogatym interfejsem użytkownika, którą będziesz mógł wykonać sam, poczynając od szkiców projektu, poprzez implementację różnych elementów, aż do porywającego końcowego efektu!

- Podstawy grafiki i interfejsów GUI
- Biblioteki grafiki pulpitu dla języka Java
- Podstawy renderingu w Swing
- Java 2D
- Typy obrazów i ich przetwarzanie
- Gradienty
- Animacja
- Efekty statyczne i dynamiczne
- Tworzenie własnych efektów
- Biblioteki - Animated Transitions i Timing Framework

Oto Twoje atrakcyjne aplikacje – wyjątkowe połączenie estetyki i funkcjonalności oraz milionów zadowolonych klientów!



Spis treści

Słowo wstępne	13
Przedmowa	15
Podziękowania	21
O autorach	23
Wstęp	25
CZĘŚĆ I PODSTAWY GRAFIKI I INTERFEJSÓW GUI	33
Rozdział 1. Biblioteki grafiki pulpitu dla języka Java	
— Swing, AWT oraz Java 2D	35
Abstract Window Toolkit (AWT)	36
Java 2D	37
Swing	37
Rozdział 2. Podstawy renderingu w Swing	39
Zdarzenia	40
Rysowanie w Swing	41
Asynchroniczne żądania odrysowania	42
Synchroniczne żądania odrysowania	43
Renderowanie w Swing	44
paintComponent()	46
paint()	48
setOpaque()	51
Podwójne buforowanie	52

Wątki	55
Model wątków	57
Stopery a wątek rozsyłania zdarzeń	62
Bezbolesna obsługa wątków poprzez SwingWorker	63
Podsumowanie wątków	66
Rozdział 3. Podstawy grafiki	67
Java 2D	67
Rendering	69
Pobieranie obiektu Graphics	70
Stan grafiki	72
Elementarne operacje graficzne	97
Rozdział 4. Obrazy	115
Typy obrazów	116
BufferedImage	119
Skalowanie obrazów	122
Jakość kontra wydajność	125
Metoda getFasterScaledInstance() — narzędzie do szybszego i lepszego skalowania obrazów	134
Rozdział 5. Wydajność	137
Zastosowanie obcinania	137
Obrazy zapewniające zgodność	143
Dlaczego powinniśmy się nimi zajmować?	143
Co można powiedzieć o obrazach zarządzanych?	145
Zapewnianie zgodności	146
Obrazy zarządzane	148
Odczytywanie danych z DataBuffer	152
Częste renderowanie obrazu	154
Obrazy pośrednie	156
Zasada ogólna	157
Jak to jest realizowane?	157
Uwagi	163
Podsumowanie	165
Optymalne rysowanie figur	165
Testowanie wydajności	167
Opcje wiersza poleceń	167
Renderowanie	169
Debugowanie wydajności	171

CZĘŚĆ II	RENDEROWANIE ZAAWANSOWANEJ GRAFIKI	173
Rozdział 6.	Przezroczystość	175
	Obiekt AlphaComposite	175
	Obiekt AlphaComposite. 12 reguł	177
	Clear	179
	Dst	179
	DstAtop	179
	DstIn	180
	DstOut	180
	DstOver	181
	Src	182
	SrcAtop	183
	SrcIn	183
	SrcOut	183
	SrcOver	184
	Xor	185
	Tworzenie i konfigurowanie obiektu AlphaComposite	186
	Typowe zastosowania AlphaComposite	187
	Zastosowanie reguły Clear	187
	Zastosowanie reguły SrcOver	188
	Zastosowanie reguły SrcIn	189
	Problemy z użyciem AlphaComposite	191
	Tworzenie własnej klasy Composite	193
	Tryb mieszania Add	193
	Implementowanie CompositeContext	197
	Łączenie pikseli	198
	Podsumowanie	200
Rozdział 7.	Gradients	201
	Dwuelementowy gradient liniowy	202
	Efekty specjalne korzystające ze zwykłych gradientów	204
	Wieloelementowy gradient liniowy	208
	Gradient kołowy	211
	Optymalizacja gradientów	214
	Buforowanie gradientu	215
	Sprytniejsze buforowanie	216
	Optymalizacja z użyciem gradientów cyklicznych	217
Rozdział 8.	Przetwarzanie obrazu	219
	Filtry obrazów	219
	Przetwarzanie obrazu za pomocą BufferedImageOp	221
	AffineTransformOp	223
	ColorConvertOp	224

ConvolveOp	226
Tworzenie jądra	228
Praca na krawędzi	229
LookupOp	231
RescaleOp	232
Własna klasa BufferedImageOp	234
Klasa bazowa filtra	234
Filtr barwiący	236
Uwaga na temat wydajności	241
Podsumowanie	242
Rozdział 9. Szklany panel	243
Rysowanie na szklanym panelu	246
Optymalizacja rysowania na szklanym panelu	247
Blokowanie zdarzeń wejściowych	250
Problemy ze zdarzeniami myszy	251
Rozdział 10. Panele warstwowe	255
Wykorzystanie paneli warstwowych	256
Porządkowanie komponentów wewnątrz jednej warstwy	260
Panele warstwowe i warstwy	261
Alternatywa dla JLayeredPane w postaci obiektów Layout	262
Rozdział 11. Zarządca rysowania	267
Gdy Swing staje się zbyt sprytny	267
Poznajemy RepaintManager	269
Zarządzanie obiektami RepaintManager	270
Odbicia i RepaintManager	271
Zapewnienie miejsca na odbicie	272
Rysowanie odbicia	275
Prostszy, przez co sprytniejszy RepaintManager	277
Podsumowanie	280
CZĘŚĆ III ANIMACJA	281
Rozdział 12. Podstawy animacji	283
Wszystko o czasie	283
Podstawowe koncepcje	284
Animacja bazująca na ramkach	284
Częstotliwość klatek	286
Ruch bazujący na czasie	286

Mierzenie czasu (oraz narzędzia pomiaru czasu platformy)	293
„Która godzina?”	293
„Czy mogę zamówić budzenie?”	296
„Zadzwoń do mnie ponownie. I ponownie. I ponownie”	298
Rozdzielczość stopera	305
Rozdzielczość System.currentTimeMillis() oraz System.nanoTime()	308
Rozdzielczość usypiania	310
Rozdzielczość stopera	315
Rozwiązanie problemu rozdzielczości	317
Animacje w aplikacjach Swing	318
Grafika animowana	319
Animowany interfejs użytkownika	321
Podsumowanie	331
Rozdział 13. Płynne przesunięcia	333
Podstawy. Dlaczego moja animacja źle wygląda?	333
Co powoduje „szarpanie” animacji i jak można ją wygładzić?	334
Synchronizacja jest (niemal) wszystkim	335
Kolor — co za różnica?	338
Pionowy powrót płamki — poczucie synchronizacji	348
SmoothMoves — program demonstracyjny	353
Tworzenie obiektów graficznych	353
Uruchamianie stopera	354
Renderowanie	355
Opcje renderowania	356
Podsumowanie	360
Rozdział 14. Biblioteka Timing Framework. Podstawy	361
Wstęp	361
Podstawowe koncepcje	363
Animator	364
Wywołania zwrotne	366
Czas trwania	368
Powtarzanie	369
Rozdzielczość	370
Operacje startowe	370
Interpolacja	377
Przyspieszenia i opóźnienia	378
Interpolator	383
Podsumowanie	395

Rozdział 15. Biblioteka Timing Framework. Funkcje zaawansowane	397
Wyzwalacze	398
Koncepcje i wykorzystanie	398
Klasy bazowe wyzwalaczy	400
Wbudowane wyzwalacze	400
Metody ustawiania właściwości	410
PropertySetter	413
Ewaluator	417
KeyFrames	419
Podsumowanie	437
CZĘŚĆ IV EFEKTY	439
Rozdział 16. Efekty statyczne	441
Rozmycie	441
Motywacja	441
Proste rozmycie	443
Rozmycie Gaussa	446
Sposób na wydajność	451
Odbicia	452
Motywacja	453
Rysowanie odbić	453
Rozmyte odbicia	454
Rzucanie cieni	455
Motywacja	455
Proste rzucanie cieni	457
Realistyczne rzucanie cieni	459
Wyróżnienia	460
Motywacja	461
Rozjaśnienie	462
Oświetlenie punktowe	464
Podświetlanie tekstu dla zapewnienia lepszej czytelności	466
Wyostrzanie	468
Motywacja	468
Proste wyostrzanie	470
Wyostrzanie selektywne	472
Wyostrzanie zmniejszonych obrazów	473
Podsumowanie	476

Rozdział 17. Efekty dynamiczne	477
Ruch	478
Motywacja	478
Ruch	480
Zanikanie	483
Motywacja	483
Strategie zanikania	486
Zanikanie przy użyciu AlphaComposite	486
Zanikanie koloru	488
Wzajemne przenikanie	490
Proste zanikanie	490
Pulsowanie	491
Motywacja	491
Poczuj puls	492
Automatyczne podświetlenie	496
Przyspieszony puls	501
Sprężyny	503
Motywacja	504
Gorączka sprężynowania	505
Morfing	508
Motywacja	508
Morfing przycisków	510
Podsumowanie	514
Rozdział 18. Biblioteka Animated Transitions	515
Płynne animowanie stanu aplikacji	515
Zasada ogólna	516
Płynne przejścia — biblioteka	519
Animacja stanu aplikacji	519
Stany GUI	519
API	520
Efekty	527
Struktura GUI	540
Obrazy i ImageHolder	540
ScreenTransition	542
Płynne przejścia — mechanizmy wewnętrzne, czyli jak zmusić Swing do takich rzeczy	543
Konfigurowanie następnego ekranu — po cichu	544
Animowanie zmian układu	545
Przyspieszanie Swing — wydajność	546
Podsumowanie	546

Rozdział 19. Narodziny bogatego interfejsu użytkownika	547
Aerith	547
Uruchamianie programu Aerith	548
Organizacja kodu	549
Projekt przepływu sterowania na papierze	549
Wizja	551
Projektowanie ekranu na papierze	553
Makiety	553
Od makiety do kodu	555
Zastosowanie warstw	556
Tryby mieszania	557
Stosowanie prowadnic	558
Ale... ja nie jestem artystą!	559
Wybór ładnych kolorów	561
Książki na temat projektowania	563
Podsumowanie	564
Zakończenie	565
Skorowidz	569

Przezroczystość

Efekt przezroczystości jest bardzo ważnym narzędziem dla programisty bogatego interfejsu użytkownika. Przezroczystość należy uznać za zasadę określającą sposób przechowywania i łączenia kolorów rysowanych figur z zastanym tłem. Przezroczystość może być zdefiniowana również w taki sposób, aby na przykład tylko czerwony składnik rysowanego obrazu był kopiowany na obszar grafiki. Przezroczystość jest również znana pod nazwą *trybu łączenia* w aplikacjach służących do edycji grafiki, takich jak Adobe Photoshop lub The GIMP, w których jest on używany do tworzenia złożonych efektów oświetleniowych. W środowisku Java przezroczystość jest reprezentowana przez obiekt implementujący interfejs `java.awt.Composite`, który może być dodany do `Graphics2D` przez wywołanie `setComposite()`.

Obiekt `AlphaComposite`

Platforma Java zawiera tylko jedną implementację interfejsu `Composite`, `java.awt.AlphaComposite`. Obiekt ten implementuje podstawowe mieszanie typu alfa, pozwalające na osiągnięcie efektów prześwitywania. Klasa `AlphaComposite` implementuje zbiór 12 reguł opisanych w artykule *Compositing Digital Images* autorstwa T. Portera oraz T. Duffa¹. Wszystkie te reguły bazują na równaniach matematycznych definiujących wartości koloru i komponentu alfa wynikowego piksela dla danego źródła (rysowanej figury) i obszaru docelowego (obszaru graficznego). Implementacja w środowisku Java zawiera dodatkowy parametr, wartość alfa wykorzystywaną do modyfikowania przezroczystości źródła przed wykonaniem mieszania.

¹ Thomas Porter i Tom Duff, *Compositing Digital Images*, [w:] *Computer Graphics*, 18, s. 253 – 259, lipiec 1984.

**Uwaga****Komponenty i kanały**

Kolory są kodowane za pomocą trzech wartości, nazywanych również komponentami lub kanałami. Najczęściej stosowane kodowanie programowe jest znane pod nazwą RGB, które korzysta czerwonego, zielonego i niebieskiego komponentu. Innym sposobem kodowania jest Yuv, które korzysta z kanału luminancji (Y) oraz dwóch kanałów chrominancji (u oraz v).

Kanał alfa, czyli komponent alfa, jest czwartym komponentem, niezależnym od kodowania kolorów, który definiuje poziom przezroczystości lub nieprzezroczystości koloru. Na przykład kolor z kanałem alfa równym 50% wartości maksymalnej będzie w połowie przezroczysty.

Aby móc zdecydować, której zasady należy użyć, konieczne jest zrozumienie równań Portera-Duffa, przedstawionych w dokumentacji `java.awt.AlphaComposite`. Aby uniknąć znużenia Czytelnika matematycznymi opisami (przeanalizowanie wszystkich 12 równań doświadczyłoby ciężko autorów), skupimy się również na jednej z najużyteczniejszych reguł Source Over, która pozwala na łączenie źródła z powierzchnią docelową tak, jakby źródło było przezroczystym rysunkiem na szkle umieszczonym na powierzchni docelowej. Równanie opisujące tę zasadę jest następujące:

$$A_r = A_s + A_d \times (1 - A_s)$$

$$C_r = C_s + C_d \times (1 - A_s)$$

Składnik A oznacza kanał alfa koloru piksela, natomiast C — każdy z komponentów koloru piksela. Indeksy r , s oraz d oznaczają odpowiednio wynik, źródło oraz cel. Łącząc wszystko razem, A_s oznacza kanał alfa źródła, czyli figury rysowanej na obszarze graficznym, natomiast A_d — kanał alfa piksela znajdującego się na obszarze graficznym. Wartości te są używane do wyliczenia wynikowej wartości kanału alfa, A_r . Wszystkie wartości używane w tych równaniach są liczbami rzeczywistymi z zakresu od 0,0 do 1,0, a wynik jest przycinany do tego zakresu.

W naszym kodzie wartości te są konwertowane do zakresów typów Java. Na przykład, gdy kolory są przechowywane przy użyciu typu całkowitego bez znaku, zamiast korzystać z zakresu od 0,0 do 1,0, każdy komponent ma wartość między 0 a 255.

**Uwaga****Komponenty wstępnie pomnożone**

Należy pamiętać, że równania Portera-Duffa są definiowane przy użyciu komponentu koloru, który jest wstępnie przemnożony przez odpowiedni komponent alfa.

Jak będzie wyglądał wynik, gdy narysujemy półprzezroczysty czerwony prostokąt na niebieskim prostokącie? Zacznijmy od zapisania równań w postaci kodu Java, w których każdy komponent będzie reprezentowany w całości:

```

int srcA = 127;           // półprzezroczyste źródło
int srcR = 255;          // pełny czerwony
int srcG = 0;            // bez zielonego
int srcB = 0;            // bez niebieskiego

int dstA = 255;          // nieprzezroczysta powierzchnia docelowa
int dstR = 0;            // bez czerwonego
int dstG = 0;            // bez zielonego
int dstB = 255;          // pełny niebieski

srcR = (srcR * srcA) / 255; // wstępne mnożenie srcR
srcG = (srcG * srcA) / 255; // wstępne mnożenie srcG
srcB = (srcB * srcA) / 255; // wstępne mnożenie srcB

dstR = (dstR * dstA) / 255; // wstępne mnożenie dstR
dstG = (dstG * dstA) / 255; // wstępne mnożenie dstG
dstB = (dstB * dstA) / 255; // wstępne mnożenie dstB

int resultA = srcA + (dstA * (255 - srcA)) / 255;
int resultR = srcR + (dstR * (255 - srcR)) / 255;
int resultG = srcG + (dstG * (255 - srcR)) / 255;
int resultB = srcB + (dstB * (255 - srcR)) / 255;

System.out.printf("(%d, %d, %d, %d)",
    resultA, resultR, resultG, resultB);

```

Po wykonaniu tego programu otrzymamy następujący wynik:

```
(255, 127, 0, 128)
```

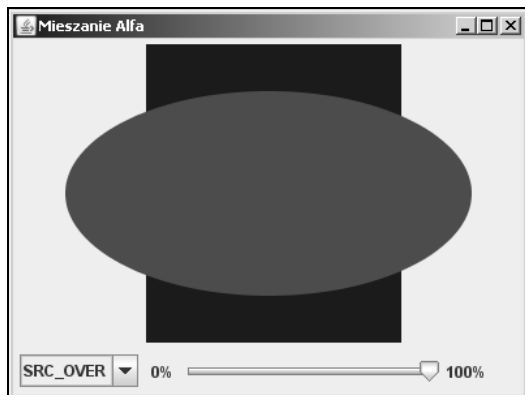
Wynikiem będzie nieprzezroczysty kolor magenta, którego możemy się spodziewać po umieszczeniu przezroczystego czerwonego arkusza na niebieskim tle². Choć zachęcamy do wykonania tych samych operacji dla pozostałych reguł, to jednak nic nie przebiję pokazania przykładów grafiki.

Obiekt AlphaComposite. 12 reguł

Przedstawimy teraz 12 reguł Portera i Duffa z krótkim opisem każdej z nich oraz rysunkiem czerwonego owalu narysowanego na niebieskim prostokącie. Proces rysowania przebiega w następujący sposób — na przezroczystym obrazie jest rysowany nieprzezroczysty niebieski prostokąt, więc mamy docelowy obraz z przezroczystymi pikselami (alfa = 0) poza niebieskim prostokątem oraz nieprzezroczyste piksele (alfa = 1) wewnątrz niebieskiego prostokąta.

² Jednak nie jest jasne, po co to chcielibyśmy zrobić.

Następnie rysowany jest czerwony owal o innej wartości alfa, tak jak jest to pokazane w oknie aplikacji z rysunku 6.1. Na koniec obraz jest kopiowany do komponentu Swing, który jest umieszczany w oknie widocznym na rysunku.



Rysunek 6.1. Demo AlphaComposites z zasadą SRC_OVER i dodatkowym kanałem alfa równym 100%

DEMO W SIECI

Każda zasada została skonfigurowana z dodatkową przezroczystością równą 50%. Można samodzielnie wypróbować różne wartości przezroczystości, korzystając z aplikacji AlphaComposites, dostępnej na witrynie WWW książki. Można również porównać wyniki uzyskane dla każdej reguły z rysunkiem 6.1, który zawiera scenę z domyślną regułą ustawioną w Graphics2D, czyli AlphaComposite.SrcOver z wartością alfa równą 100%.

Kolejne reguły są przedstawiane razem z równaniami wykorzystywanymi do obliczenia wyniku. Podobnie jak w opisie reguły Source Over, składnik A oznacza kanał alfa koloru piksela, natomiast C — każdy z komponentów koloru piksela. Indeksy r , s oraz d oznaczają odpowiednio wynik, źródło oraz cel.



Uwaga Terminologia

Gdy mówimy o pikselu *źródłowym*, mamy na myśli obszary źródła, które nie są przezroczyste. Podobnie piksel *docelowy* określa te obszary powierzchni docelowej, które nie są przezroczyste. Z tego powodu określenie „obszar źródłowy wewnątrz docelowego” oznacza te nieprzezroczyste piksele źródłowe, które są narysowane na nieprzezroczystym obszarze docelowym.

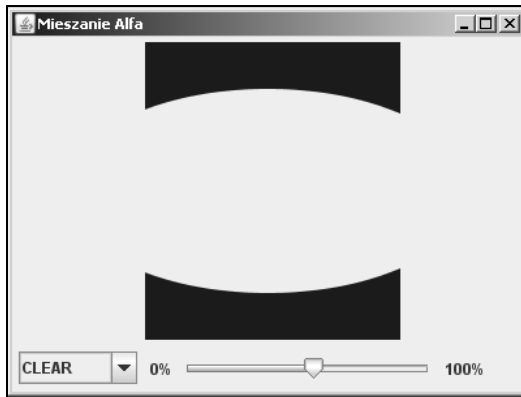
Oprócz przeczytania opisu (i porównania wyników ze zrzutami ekranu) można się również zapoznać z artykułami JavaDoc na temat AlphaComposite, które dokładniej opisują sposób działania tych reguł.

Clear

$$A_r = 0$$

$$C_r = 0$$

Zarówno kolor, jak i kanał alfa są zerowane. Niezależnie od koloru użytego do rysowania każdy piksel docelowy pokryty przez piksel źródłowy znika, tak jak jest to pokazane na rysunku 6.2.



Rysunek 6.2. Demo AlphaComposites z zasadą Clear

Dst

$$A_r = A_d$$

$$C_r = C_d$$

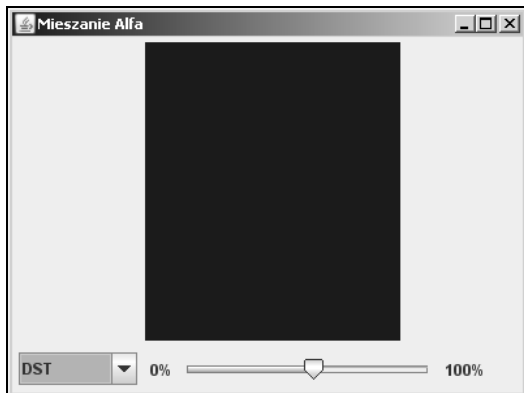
Obszar docelowy pozostaje niezmieniony. Cokolwiek zostanie narysowane na powierzchni docelowej, zostanie anulowane, tak jak jest to pokazane na rysunku 6.3.

DstAtop

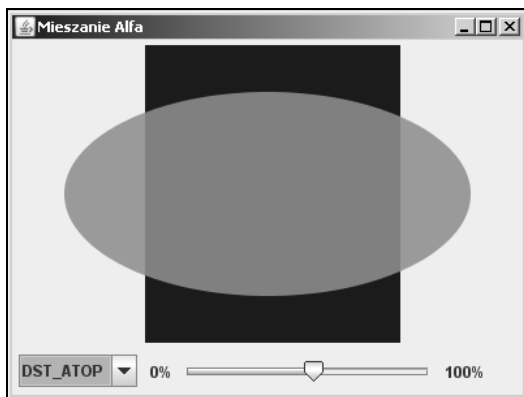
$$A_r = A_s \times (1 - A_d) + A_d \times A_s = A_s$$

$$C_r = C_s \times (1 - A_d) + C_d \times A_s$$

Część obszaru docelowego znajdującego się wewnątrz źródłowego jest łączona ze źródłem i zastępuje obszar docelowy. Daje to efekt rysowania obszaru docelowego na źródłowym (rysunek 6.4), a nie odwrotnie.



Rysunek 6.3. Demo AlphaComposites z zasadą Dst



Rysunek 6.4. Demo AlphaComposites z zasadą DstAtop

DstIn

$$A_r = A_d \times A_s$$

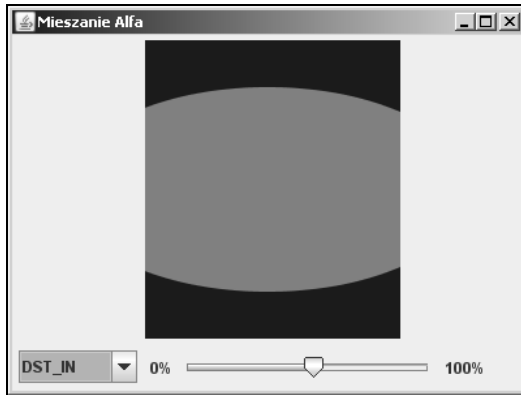
$$C_r = C_d \times A_s$$

Część obszaru docelowego leżąca wewnątrz źródłowego zastępuje obszar docelowy. Jest to odwrotność DstOut, ale przy wartości alfa 50% obie operacje dają te same wyniki (rysunek 6.5).

DstOut

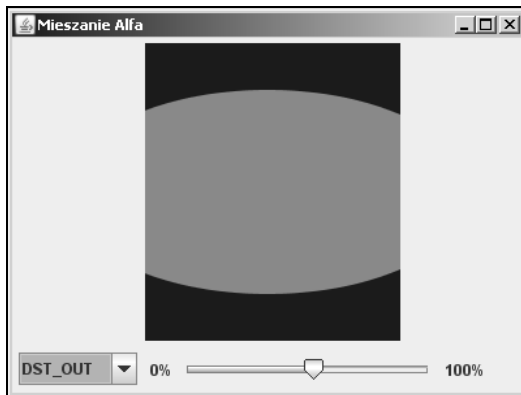
$$A_r = A_d \times (1 - A_s)$$

$$C_r = C_d \times (1 - A_s)$$



Rysunek 6.5. Demo AlphaComposites z zasadą DstIn

Część obszaru docelowego leżąca na zewnątrz źródłowego zastępuje obszar docelowy. Jest to odwrotność DstIn, ale przy wartości alfa 50% obie operacje dają te same wyniki (rysunek 6.6).



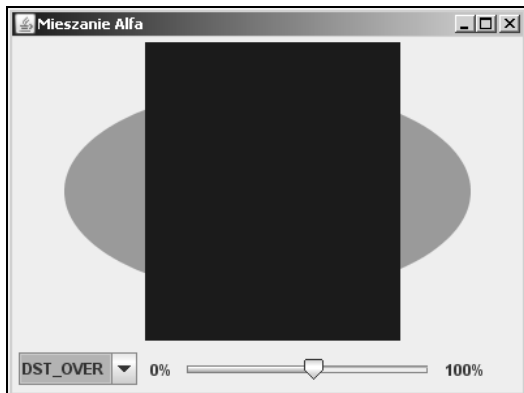
Rysunek 6.6. Demo AlphaComposites z zasadą DstOut

DstOver

$$A_r = A_s \times (1 - A_d) + A_d$$

$$C_r = C_s \times (1 - A_d) + C_d$$

Obszar docelowy jest łączony ze źródłowym, a wynik zastępuje obszar docelowy. Części źródła leżące poza obszarem docelowym są rysowane normalnie z dodatkową przezroczystością, tak jak jest to pokazane na rysunku 6.7.



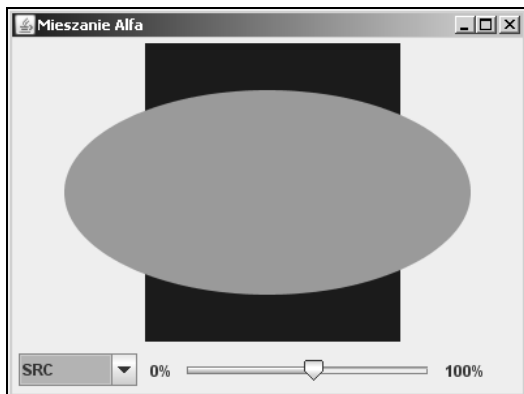
Rysunek 6.7. Demo AlphaComposites z zasadą DstOver

Src

$$A_r = A_s$$

$$C_r = C_s$$

Obszar źródłowy jest kopiowany do docelowego. Jest on zastępowany przez obszar źródłowy. Na rysunku 6.8 niebieski prostokąt (docelowy) nie jest widoczny pod czerwonym owalem, ponieważ owal ten (źródło) zastępuje go.



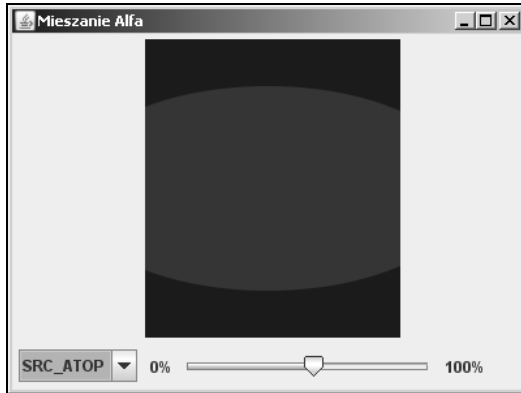
Rysunek 6.8. Demo AlphaComposites z zasadą Src

SrcAtop

$$A_r = A_s \times A_d + A_d \times (1 - A_s) = A_d$$

$$C_r = C_s \times A_d + C_d \times (1 - A_s)$$

Część obszaru źródłowego leżąca wewnątrz docelowego jest łączona z obszarem docelowym. Część obszaru źródłowego leżąca poza docelowym jest usuwana (rysunek 6.9).



Rysunek 6.9. Demo AlphaComposites z zasadą SrcAtop

SrcIn

$$A_r = A_s \times A_d$$

$$C_r = C_s \times A_d$$

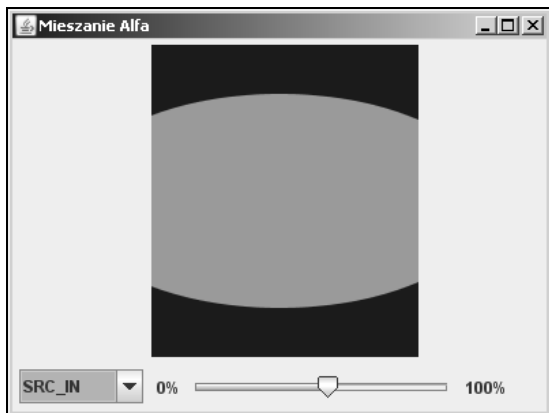
Część obszaru źródłowego leżąca wewnątrz docelowego zastępuje obszar docelowy. Część obszaru źródłowego leżąca poza docelowym jest usuwana (rysunek 6.10).

SrcOut

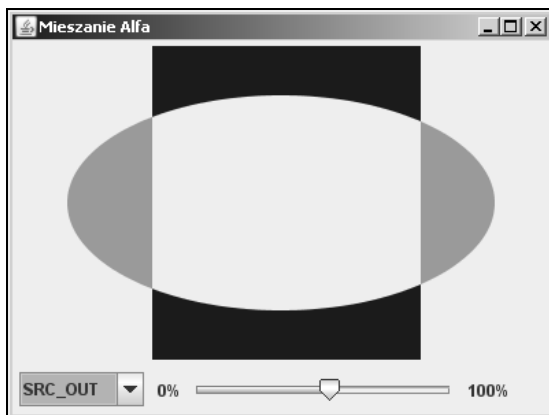
$$A_r = A_s \times (1 - A_d)$$

$$C_r = C_s \times (1 - A_d)$$

Część obszaru źródłowego leżąca na zewnątrz docelowego zastępuje obszar docelowy. Część obszaru źródłowego leżąca wewnątrz docelowego jest usuwana (rysunek 6.11).



Rysunek 6.10. Demo AlphaComposites z zasadą SrcIn



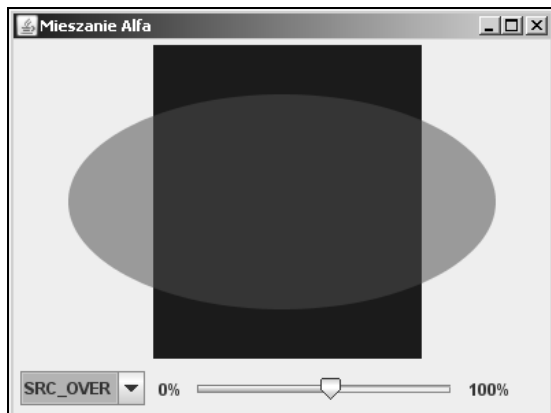
Rysunek 6.11. Demo AlphaComposites z zasadą SrcOut

SrcOver

$$A_r = A_s + A_d \times (1 - A_s)$$

$$C_r = C_s + C_d \times (1 - A_s)$$

Obszar źródłowy jest łączony z obszarem docelowym (rysunek 6.12). SrcOver jest domyślną regułą ustawianą dla powierzchni `Graphics2D`.



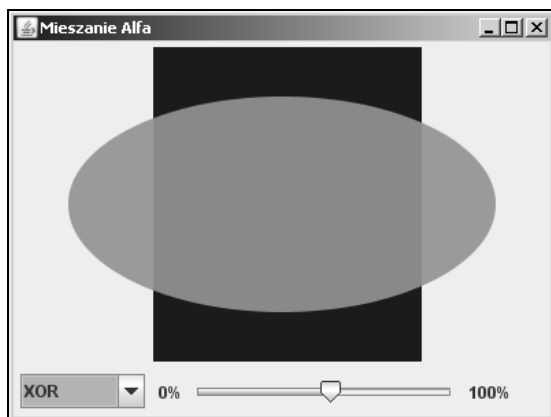
Rysunek 6.12. Demo AlphaComposites z zasadą SrcOver

Xor

$$A_r = A_s \times (1 - A_d) + A_d \times (1 - A_s)$$

$$C_r = C_s \times (1 - A_d) + C_d \times (1 - A_s)$$

Część obszaru źródłowego, który znajduje się poza obszarem docelowym, jest łączona z częścią obszaru docelowego leżącego poza źródłowym (rysunek 6.13).



Rysunek 6.13. Demo AlphaComposites z zasadą Xor

Tworzenie i konfigurowanie obiektu `AlphaComposite`

Obiekt `AlphaComposite` może być ustawiony w `Graphics2D` w dowolnym momencie przez wywołanie metody `setComposite()`. Metoda ta wpływa na wszystkie kolejne operacje graficzne, więc po zakończeniu rysowania należy pamiętać o przywróceniu wcześniejszego obiektu.



Wskazówka

Można również użyć metody `Graphics.create()` do wykonania kopii powierzchni rysowania, którą można usunąć po zakończeniu rysowania.

Mamy dwie możliwości utworzenia obiektu `AlphaComposite`. Pierwsza, prostsza, polega na wykorzystaniu instancji predefiniowanych w klasie `AlphaComposite`. Wszystkie te instancje są udostępniane jako publiczne pola statyczne, których nazwy są zgodne z konwencją nazewnictwa dla klas. Na przykład instancja `Source Over` może być pobrana za pomocą wyrażenia `AlphaComposite.SrcOver`. Poniżej przedstawiony jest przykład wykorzystania tej metody:

```
@Override
protected void paintComponent(Graphics g) {
    Graphics2D g2 = (Graphics2D) g;
    Composite oldComposite = g2.getComposite();

    g2.setComposite(AlphaComposite.SrcOver);
    g2.setColor(Color.RED);
    g2.fillOval(0, 0, 80, 40);
    g2.setComposite(oldComposite);
}
```

Predefiniowane instancje `AlphaComposite` mają ustawioną dodatkową wartość alfa na 100%.

Innym sposobem na utworzenie instancji z wartością alfa równą 100% jest wykorzystanie metody `getInstance(int)`. Poprzedni kod pozostanie bez zmian, poza następującym wierszem:

```
g2.setComposite(AlphaComposite.getInstance(AlphaComposite.SRC_OVER));
```

Gdy chcemy użyć obiektu `AlphaComposite` z wartością alfa mniejszą niż 100%, musimy skorzystać z wywołania `getInstance(int, float)`. Drugi parametr reprezentuje przezroczystość i może przyjmować wartości od `0.0f` do `1.0f`. W poniższym wierszu tworzona jest instancja dla metody `Source Over` z wartością alfa równą 50%:

```
g2.setComposite(AlphaComposite.getInstance(AlphaComposite.SRC_OVER, 0.5f));
```

Łatwiejsze tworzenie obiektu AlphaComposite

Jedną z moich ulubionych funkcji w Java SE 6³ są dwie nowe metody klasy AlphaComposite: `derive(int)` oraz `derive(float)`. Można z nich skorzystać do utworzenia kopii istniejącego obiektu AlphaInstance z nowymi ustawieniami. Poniżej przedstawiony jest sposób konwersji obiektu ze zdefiniowaną zasadą Source In na Source Over:

```
AlphaComposite composite = AlphaComposite.SrcIn;
composite = composite.derive(AlphaComposite.SRC_OVER);
```

Wywołanie `derive()` w celu zmiany zasady pozostawia niezmienną wartość alfa i przypisuje ją do nowej zasady. Można również zmieniać przezroczystość, pozostawiając tę samą zasadę. Zamiast wywołania `getInstance(int, float)` można użyć:

```
g2.setComposite(AlphaComposite.SrcOver.derive(0.5f));
```

Wywołania `derive()` mogą być łączone, aby jednocześnie zmienić wartość alfa oraz regułę:

```
g2.setComposite(composite.derive(0.5f).derive(AlphaComposite.DST_OUT));
```

Kod ten jest czytelniejszy i łatwiejszy do utrzymania niż w przypadku ogólnego wywołania `getInstance()` dostępnego w wersjach wcześniejszych niż Java SE 6.

Typowe zastosowania AlphaComposite

Obiekty AlphaComposite są uniwersalnym i zaawansowanym narzędziem, o ile umie się z nich korzystać. Choć nie jesteśmy w stanie zdefiniować sytuacji, w których należy użyć każdej z 12 zasad, przedstawimy tu cztery najbardziej przydatne: Clear, Src, SrcOver oraz SrcIn.

Zastosowanie reguły Clear

Reguła Clear może być wykorzystywana w przypadkach, gdy chcemy ponownie wykorzystać przezroczysty lub prześwitujący obraz. Można w ten sposób wymazać tło, aby obraz był całkowicie przezroczysty. Jak pamiętamy, równanie dla reguły Clear jest następujące:

$$A_r = 0$$

$$C_r = 0$$

³ Faktycznie metody te są moim numerem jeden. Gdy korzysta się z AlphaComposite każdego dnia, to prędzej lub później konieczność kolejnego wywołania `getInstance(int, float)` powoduje odruch gryzienia biurka.

Jak można zauważyć, wynik nie zależy od koloru źródłowego ani docelowego. Dzięki temu można narysować cokolwiek, aby skasować obraz. Oznacza to również, że przezroczystość obiektu `Composite` nie ma znaczenia. Wynikiem operacji z zastosowaniem tej reguły jest wycięcie dziury o kształcie rysowanej figury. Dzięki temu reguła `Clear` może być traktowana identycznie jak gumka z Adobe Photoshop lub innego podobnego programu. Poniżej zamieszczony jest przykład kasowania zawartości przezroczystego obrazu:

```
// Obraz ma kanał alfa
BufferedImage image = new BufferedImage(200, 200, BufferedImage.TYPE_INT_ARGB);
Graphics2D g2 = image.createGraphics();
// Rysowanie obrazu
// ...
// Usuwanie zawartości obrazu
g2.setComposite(AlphaComposite.Clear);
// Kolor i pędzel nie mają znaczenia
g2.fillRect(0, 0, image.getWidth(), image.getHeight());
```

Reguła `Clear` pozwala kasować obszary o dowolnym kształcie.

Zastosowanie reguły `SrcOver`

`SrcOver` jest domyślną regułą ustawianą dla powierzchni `Graphics2D`. Tego typu obiekt `Composite` zapewnia, że źródło zostanie narysowane w całości, bez żadnych modyfikacji. Można użyć tej reguły, aby upewnić się, że obszar grafiki jest prawidłowo skonfigurowany i renderowanie nie będzie zakłócanie przez modyfikacje wprowadzone w obiekcie `Graphics` przez inny komponent naszej aplikacji.

Można również użyć `SrcOver` do rysowania prześwitujących obiektów bez wpływania na powierzchnię docelową. Spójrzmy na aplikację pokazaną na rysunku 6.14.

W kilku miejscach można zauważyć, że metoda `SrcOver` została wykorzystana do osiągnięcia efektu przezroczystości. Okno dialogowe pośrodku oraz palety z brzegu ekranu są prześwitujące.

Można sterować przezroczystością źródła, zmieniając wartość alfa skojarzoną z obiektem `AlphaComposite`, zgodnie z informacjami z punktu „Tworzenie i konfigurowanie obiektu `AlphaComposite`”.

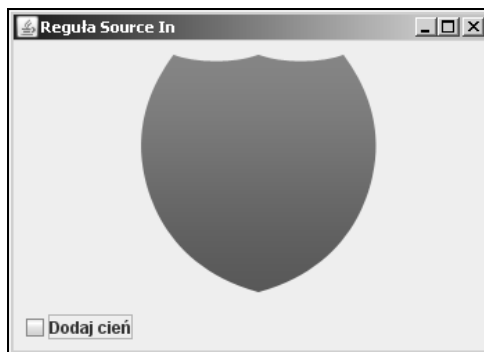
Należy pamiętać, że obiekty `Composite` działają w przypadku wszystkich rysowanych figur, również obrazów. Można również animować wartość alfa dla reguły `SrcOver`, co pozwala na tworzenie interesujących efektów przenikania.



Rysunek 6.14. Wynik zastosowania metody Source Over

Zastosowanie reguły SrcIn

SrcIn jest przydatną, ale zbyt rzadko stosowaną regułą dla Composita. Może być ona wykorzystywana w przypadku, gdy chcemy zastąpić zawartość istniejącego obrazu. Na rysunku 6.15 przedstawiona jest aplikacja, która rysuje na ekranie rysunek tarczy wypełnionej niebieskim gradientem.

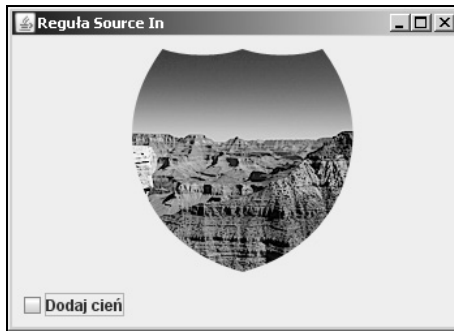


Rysunek 6.15. Proste wypełnienie gradientem

W jaki sposób można narysować podobną tarczę, ale z fotografią zamiast gradientu? Można to łatwo osiągnąć ustawiając obiekt Composita z regułą SrcIn dla obszaru graficznego:

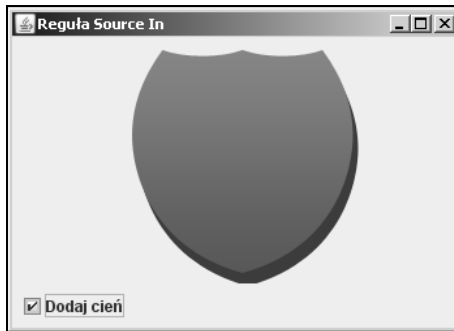
```
// Rysowanie niebieskiej tarczy
g2.drawImage(image, x, y, null);
// Zmiana zawartości tarczy na zdjęcie Wielkiego Kanionu
g2.setComposite(AlphaComposite.SrcIn);
g2.drawImage(landscape, x, y, null);
```

Zgodnie z regułą SrcIn Java 2D zamienia zawartość powierzchni docelowej na zawartość powierzchni źródłowej, która znajduje się wewnątrz docelowej, jak jest to pokazane na rysunku 6.16.



Rysunek 6.16. Przycięcie fotografii do kształtu tarczy

Można skorzystać z tej techniki do tworzenia ramek zdjęć, do przycinania rysunków lub obrazów, a nawet do tworzenia cieni. Jeżeli wypełnimy czarny prostokąt na oryginalnym obrazie, uzyskamy cień. Po ponownym narysowaniu oryginalnego rysunku, ale nieco przesuniętego, uzyskamy oczekiwany efekt, pokazany na rysunku 6.17.



Rysunek 6.17. Prosty efekt cienia

Jeżeli zmienimy przezroczystość obiektu SrcIn, otrzymamy przezroczysty cień.

**DEMO
W SIECI**

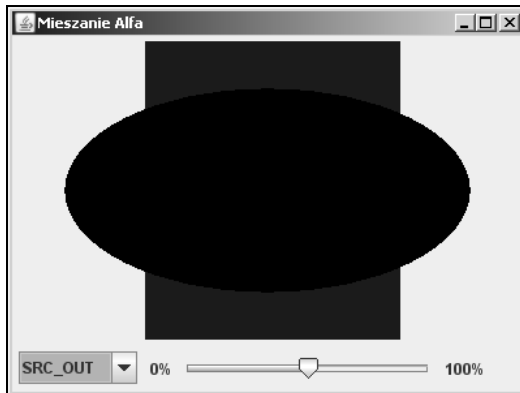
Pełny kod źródłowy tych przykładów można znaleźć w projekcie SourceIn na witrynie WWW tej książki.

**Uwaga****Miękkie przycinanie**

Przykład ten przedstawia zastosowanie reguły SrcIn do wykonania miękkiego przycinania lub przycinania z wygładzaniem z użyciem dowolnych kształtów.

Problemy z użyciem AlphaComposite

Niektóre z reguł AlphaComposite mogą dawać dziwne wyniki w przypadku ich użycia w komponentach Swing. Może się zdarzyć, że zobaczymy dużą czarną dziurę w miejscu, które powinno być puste lub zawierać inny kolor naszego rysunku, tak jak jest to pokazane na rysunku 6.18.



Rysunek 6.18. W tym użyciu SrcOut czarny owal w złożeniach miał być czerwony

Problem ten występuje, jeżeli rysujemy bezpośrednio na obszarze docelowym, który nie posiada wartości alfa, na przykład buforze tylnym Swing lub innym obrazie bez kanału alfa z użyciem reguły wymagającej wartości z tego kanału w swoim równaniu. W tym przypadku reguła SrcOut korzysta z następujących równań:

$$A_r = A_s \times (1 - A_d)$$

$$C_r = C_s \times (1 - A_d)$$

Wartości koloru i kanału alfa dla wyniku są obliczane z wykorzystaniem wartości alfa powierzchni docelowej. Gdy rysujemy komponent Swing, docelowy bufor tylny nie posiada kanału alfa. W takiej sytuacji wszystkie piksele docelowe są traktowane jako nieprzezroczyste i ich wartości alfa (A_d) mają zawsze wartość 1.0. Jest to dosyć nienaturalne, ponieważ jako programiści zawsze uważaliśmy, że interfejsy użytkownika są strukturami warstwowymi.

Gdy spojrzymy na rysunek 6.18, zauważymy jedną warstwę szarego koloru (tło), jedną warstwę niebieskiego (prostokąt) oraz jedną warstwę czarnego (owal). Można więc uważać, że oczywiste jest, że niebieski prostokąt jest otoczony przezroczystymi pikselami. W rzeczywistości okno Swing jest płaskie, a nie warstwowe. Za każdym razem, gdy rysujemy komponent Swing, bufor tła reprezentuje obraz nieprzezroczysty.

Dlaczego więc owal jest nadal czarny?

Jeżeli rozwiążemy poprzednie równania i w miejsce A_d umieścimy jej wartość 1.0, otrzymamy następujące wyniki:

$$A_r = A_s \times (1 - 1) = 0$$

$$C_r = C_s \times (1 - 1) = 0$$

Wartości wszystkich komponentów mają wartość 0, co reprezentuje kolor czarny. Nawet jeżeli wynikowy kanał alfa ma wartość 0, czyli jest całkowicie przezroczysty, nie ma to znaczenia, ponieważ bufor tła Swing nie uwzględnia wartości alfa. Za każdym razem, gdy rysujemy komponent Swing lub na innej nieprzezroczystej powierzchni przy użyciu reguły odczytującej wartość alfa powierzchni docelowej, otrzymamy nieprawidłowe wyniki.

Na szczęście rozwiązanie tego problemu jest dosyć łatwe do zaimplementowania. Zamiast rysować bezpośrednio w komponencie Swing, należy wcześniej narysować potrzebne elementy na obrazie z kanałem alfa, a następnie skopiować wynik na ekran.

```
@Override
protected void paintComponent(Graphics g) {
    // Tworzenie obrazu z kanałem alfa
    // Obraz ten może być buforowany dla uzyskania lepszej wydajności
    BufferedImage image = new BufferedImage(getWidth(),
        getHeight(), BufferedImage.TYPE_INT_ARGB);

    Graphics2D g2 = image.createGraphics();
    g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
        RenderingHints.VALUE_ANTIALIAS_ON);

    g2.setColor(Color.BLUE);
    g2.fillRect(4 + (getWidth() / 4), 4,
        getWidth() / 2, getHeight() - 8);

    // Ustawianie obiektu Composite
    g2.setComposite(AlphaComposite.SrcOut);
    g2.setColor(Color.RED);
    g2.fillOval(40, 40, getWidth() - 80, getHeight() - 80);
    g2.dispose();

    // Rysowanie obrazu na ekranie
    g.drawImage(image, 0, 0, null);
}
```

Obraz z kanałem alfa jest po utworzeniu całkowicie pusty; każdy piksel jest domyślnie przezroczysty. Dzięki temu równania działają tak, jak tego oczekujemy.



Wskazówka

Tymczasowe obrazy pamięciowe

Tworzenie tymczasowego obrazu pamięciowego jest dodatkową operacją, która może być kosztowna, jeżeli wykonujemy ją przy każdym odrysowaniu komponentu, więc prawdopodobnie warto go ponownie wykorzystywać. Można buforować gotowy rysunek, buforować wyniki rysowania lub po prostu przechowywać obiekt `BufferedImage` i rysować na nim przy każdym wywołaniu metody `paintComponent()`. Jeżeli zdecydujemy się na jego późniejsze wykorzystanie, nie należy zapominać o wcześniejszym jego wyczyszczeniu.

Jeżeli kiedykolwiek zobaczymy nieoczekiwane czarne piksele na ekranie, należy sprawdzić, czy docelowa powierzchnia posiada kanał alfa. Jeżeli nie, należy rozwiązać problem przez użycie obrazu pamięciowego.

Tworzenie własnej klasy Composite

W Java SE 6 jedyną klasą `Composite` dostępną w podstawowej platformie jest `AlphaComposite`. W większości aplikacji jest to wystarczające, ponieważ i tak niektóre reguły są rzadko wykorzystywane. Niemniej przydatne jest posiadanie różnych takich klas, szczególnie gdy trzeba zaimplementować makiety przygotowane przez projektantów grafiki.

Programiści żyją w świecie utworzonym przez środowiska IDE oraz kompilatory, natomiast projektanci mają do dyspozycji tak zaawansowane narzędzia, jak Adobe Photoshop. Narzędzia te pozwalają użytkownikom na stosowanie różnych trybów mieszania warstw składających się na projekt graficzny, a większość z nich nie waha się z nich skorzystać.

Implementowanie projektu graficznego zbudowanego na bazie tych trybów mieszania może stać się trudnym zadaniem, jeżeli będziemy korzystać z podstawowych trybów mieszania dostępnych w JDK. Nie należy jednak się załamywać, możemy napisać własne klasy realizujące mieszanie!

Tryb mieszania Add

Tworzenie klas `Composite` nie wymaga zbyt wiele pracy. Faktycznie najtrudniejsze jest opracowanie interesującej reguły mieszania, a nie jej zakodowanie.

**DEMO
W SIECI**

Na witrynie WWW tej książki dostępny jest projekt o nazwie `BlendComposites`, który zawiera 31 nowych metod mieszania, które zostały zainspirowane funkcjami graficznymi dostępnymi w takich programach, jak Adobe Photoshop. Pokażemy tutaj, jak zaimplementować jedną z nich — `Add`. Sposób realizacji innych metod można znaleźć w tym projekcie dostępnym poprzez WWW.

Tryb mieszania `Add` polega na dodawaniu do siebie wartości źródłowej i docelowej.

$$A_r = A_s + A_d$$

$$C_r = C_s + C_d$$

Rysunki 6.19, 6.20 oraz 6.21 ilustrują efekty osiągnięte za pomocą tej metody.



Rysunek 6.19. Obraz docelowy w przypadku obiektu `Composite`

Pierwszym krokiem jest utworzenie nowej klasy implementującej interfejs `java.awt.`

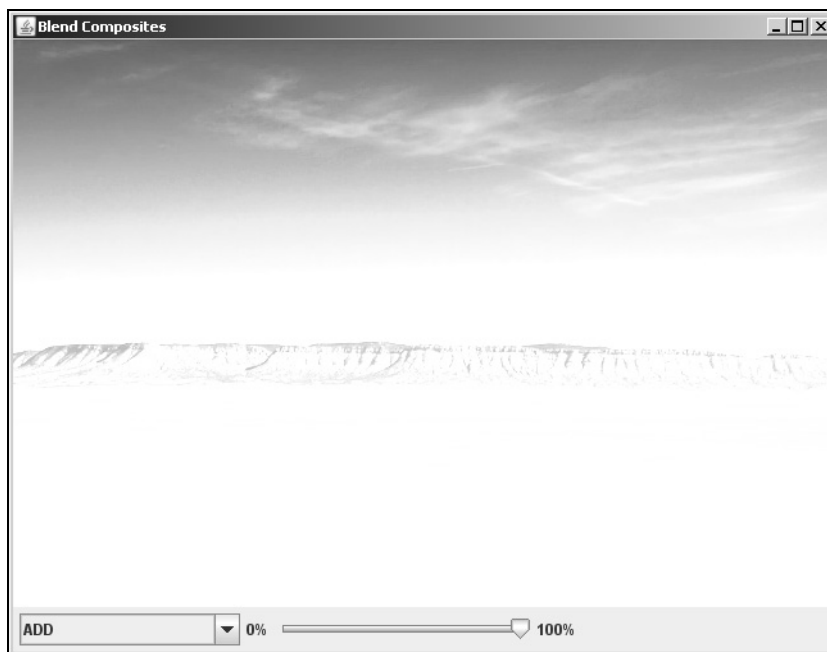
↳ `Composite`:

```
public class AddComposite implements Composite {
    private static boolean checkComponentsOrder(ColorModel cm) {
        if (cm instanceof DirectColorModel &&
            cm.getTransferType() == DataBuffer.TYPE_INT) {
            DirectColorModel directCM = (DirectColorModel) cm;
            return directCM.getRedMask() == 0x00FF0000 &&
                directCM.getGreenMask() == 0x0000FF00 &&

```



Rysunek 6.20. Obraz źródłowy w przypadku obiektu Composite



Rysunek 6.21. Wynik mieszania — ciemne piksele źródła mają mniejszy wpływ na wynik

```

        directCM.getBlueMask() == 0x000000FF &&
        (directCM.getNumComponents() == 3 ||
        directCM.getAlphaMask() == 0xFF000000);
    }

    return false;
}

public CompositeContext createContext(
    ColorModel srcColorModel, ColorModel dstColorModel,
    RenderingHints hints) {
    if (!checkComponentsOrder(srcColorModel) ||
        !checkComponentsOrder(dstColorModel)) {
        throw new RasterFormatException(
            "Niezgodne modele kolorów");
    }

    return new BlendingContext(this);
}
}

```

Jest to bardzo prosta klasa, ponieważ konieczne jest zaimplementowanie w niej tylko jednej metody, `createContext()`. Metoda `checkComponentsOrder()` jest wykorzystywana przez `createContext()` do zagwarantowania prawidłowego formatu źródłowego i docelowego. Kod ten sprawdza model kolorów, by ustalić, czy piksele są zapisywane jako liczby całkowite. Dodatkowo kontrolowane jest to, czy komponenty kolorów są w następującym porządku: alfa (jeżeli istnieje), czerwony, zielony i niebieski.

Oprócz tego dokumentacja zobowiązuje programistów, aby obiekty `Composite` były niezmiennie.



Uwaga

Dlaczego niezmiennosc jest tak ważna?

Gdy mamy instancje niezmienniej klasy `Composite`, to nie ma możliwości zmiany jej wewnętrznych wartości. Wystarczy sobie wyobrazić, co by się stało, gdyby wątek tła zmienił ustawienia obiektu `Composite`, gdyby był on wykorzystywany do rysowania na ekranie. W przypadku niezmiennych obiektów `Composite` zawsze możemy zagwarantować wynik operacji rysowania.

Można sprawdzić w dokumentacji, w jaki sposób osiągnięta jest niezmiennosc `AlphaComposite`. Jedne metody, które pozwalają na zmodyfikowanie wartości obiektu, zwracają nowe instancje — `getInstance()` oraz `derive()`. Nie ma żadnego sposobu, aby pobrać obiekt przypisany do obszaru graficznego i zmienić jego ustawienia.

Przedstawiona poniżej klasa `AddComposite` jest zgodna z tą zasadą, ponieważ nie ma publicznie dostępnych metod modyfikujących stan. Instancja `AddContext` zwracana przez metodę `createContext()` jest częścią implementacji; przedstawiono ją poniżej.

Implementowanie CompositeContext

Wszystkie operacje związane z mieszaniem kolorów są wykonywane w klasie `CompositeContext` zwracanej przez `createContext()`. Jak już wspominaliśmy, dokumentacja ostrzega przed środowiskami wielowątkowymi i wyjaśnia, jak można jednocześnie korzystać z kilku kontekstów. Dlatego właśnie metoda implementowana w `AddComposite` zwraca nowy obiekt `AddContext`. Jeżeli obiekt ma parametry, na przykład wartość alfa, tak jak `AlphaComposite`, można przekazać je do konstruktora kontekstu. W metodzie `createContext()` można zapisywać kontekst potrzebny do wykonania operacji mieszania.

Poniżej zamieszczone są dwie metody, które muszą być zdefiniowane w klasie implementującej interfejs `CompositeContext`:

```
void compose(Raster src, Raster dstIn, WritableRaster dstOut);  
void dispose();
```

Pierwsza metoda, `compose()`, realizuje operację mieszania. Druga, jest wywoływana po zakończeniu mieszania i może być wykorzystywana do czyszczenia zasobów, które mogły być zbuforowane w konstruktorze lub metodzie `compose()`.

Implementowanie metody `compose()` wymaga zrozumienia znaczenia jej trzech parametrów. Obiekt `Raster` jest reprezentacją prostokątnej tablicy pikseli. Dlatego `src Raster` jest tablicą pikseli reprezentujących źródło, które jest obiektem graficznym do połączenia z docelowym obszarem grafiki. Parametr `dstIn Raster` reprezentuje docelową tablicę pikseli lub inaczej piksele znajdujące się już w obszarze grafiki. Ostatni parametr, `dstOut`, jest tablicą pikseli, w której będzie zapisany wynik połączenia. Zarówno `src`, jak i `dstIn` są tylko do odczytu, a nowe dane będą zapisane w `dstOut`. Obiekt `Raster` przechowuje dosyć dużo informacji, razem z rozmiarem tablicy oraz jej typu przechowywania.

Dla uproszczenia `AddContext` działa tylko na obiektach `Raster`, przechowujących reprezentację pikseli w postaci liczb całkowitych. Jeżeli na przykład za pomocą tego obiektu będziemy próbować obraz o typie `BufferedImage.TYPE_3BYTE_BGR`, zostanie zgłoszony wyjątek.

W implementacji klasy `AddComposite` nie ma potrzeby buforowania wartości, więc metoda `dispose()` będzie pusta. Zanim napiszemy kod metody `compose()`, potrzebne będą dwie metody narzędziowe `fromRGBArray()` oraz `toRGBArray()`. Ponieważ ta klasa mieszająca operuje na pikselach przechowywanych jako wartości całkowite, wszystkie cztery komponenty (alfa, czerwony, zielony i niebieski) są reprezentowane jako jedna liczba całkowita. Aby zastosować zdefiniowane wcześniej równanie, konieczne jest rozbicie wartości piksela na cztery liczby, reprezentujące kolejne komponenty. Metody `fromRGBArray()` oraz `toRGBArray()` są prostymi metodami pomocniczymi przekształcającymi piksele na komponenty kolorów oraz komponenty kolorów na piksele. Niekompletna implementacja `AddContext` wygląda następująco:

```
private class AddContext implements CompositeContext {
    public void dispose() {
    }

    public void compose(Raster src, Raster dstIn, WritableRaster dstOut) {
        // Tu znajdzie się dalszy kod
    }

    private static void toRGBArray(int pixel, int[] argb) {
        argb[0] = (pixel >> 24) & 0xFF;
        argb[1] = (pixel >> 16) & 0xFF;
        argb[2] = (pixel >> 8) & 0xFF;
        argb[3] = (pixel ) & 0xFF;
    }

    private static int fromRGBArray(int[] argb) {
        return (argb[0] & 0xFF) << 24 |
            (argb[1] & 0xFF) << 16 |
            (argb[2] & 0xFF) << 8 |
            (argb[3] & 0xFF);
    }
}
```

Łączenie pikseli

Pierwszym krokiem przy implementacji metody `compose()` jest zdefiniowanie obszaru, w którym zostanie wykonane łączenie. Można odczytać wymiary wejściowych obiektów `Raster`, ale niekoniecznie muszą być takie same. Na przykład raster źródłowy może być mniejszy niż docelowy. Aby uniknąć odczytywania lub zapisywania poza granicami obiektu `Raster`, należy znaleźć wymiary wspólne dla obu obiektów:

```
int width = Math.min(src.getWidth(), dstIn.getWidth());
int height = Math.min(src.getHeight(), dstIn.getHeight());
```

Ponieważ równania są zdefiniowane wcześniej, proces łączenia polega na przejściu przez wszystkie piksele źródłowe i docelowe w zdefiniowanym właśnie obszarze i zmieszaniu ich ze sobą. W tym celu napiszemy dwie pętle, jedną dla wierszy i drugą dla kolumn, wewnątrz których będą odczytywane piksele za pomocą `src.getPixel()` oraz `dstIn.getPixel()` i będzie wykonywana operacja łączenia.

Podejście to działa bez zarzutu, ale wymaga wywołania metody `Raster.getPixel()` dwukrotnie dla każdego piksela w obszarze łączenia, co powoduje powstanie kilkuset lub kilku tysięcy wywołań metody. Jeżeli wykorzystana zostanie metoda `Raster.getDataElements()`, można ograniczyć tę liczbę i poprawić wydajność. Metoda ta pozwala na jednoczesne pobranie całego prostokątnego obszaru pikseli.

Wyobraźmy sobie łączenie obszaru o wymiarach 640×400 z obszarem grafiki o wymiarach również 640×400 — konieczne będzie wywołanie metody `getPixel()` 512 000 razy i tylko 800-krotny odczyt obiektów `Raster`, jeżeli będziemy odczytywali wiersze za pomocą `getDataElements()`. Wywoływanie metod w wewnętrznej pętli powinno być w razie możliwości unikane; `getDataElements()` oferuje bardzo efektywny sposób na uniknięcie wielokrotnych wywołań metody. Rozmiar obszaru pobieranego za pomocą `getDataElements()` jest pozostawiony programiście. Im większy obszar, tym mniej będzie potrzebnych wywołań metod, ale każdy przebieg pętli będzie wymagał przydzielenia większego obszaru pamięci.

W naszym przypadku odczytujemy obiekt `Raster` wierszami, zachowując równowagę między szybkością i zużyciem pamięci. Po zdefiniowaniu strategii odczytu obiektów `Raster` można zadeklarować struktury danych, w których będą przechowywane połączone piksele:

```
// Tymczasowa tablica dla operacji łączenia
// Przechowuje komponenty koloru dla jednego piksela źródłowego
int[] srcPixel = new int[4];
// Przechowuje komponenty koloru jednego piksela docelowego
int[] dstPixel = new int[4];
// Przechowuje jeden wiersz rastra źródłowego
int[] srcPixelsArray = new int[width];
// Przechowuje jeden wiersz rastra docelowego
int[] dstPixelsArray = new int[width];
// Przechowuje wyniki łączenia
int[] result = new int[4];
```

Najważniejsza część łączenia czeka jeszcze na napisanie. Zewnętrzna pętla przebiega po kolejnych wierszach obszaru łączenia i zapisuje je w `srcPixels` oraz `dstPixels`. Zadaniem pętli wewnętrznej jest odczyt wszystkich pikseli przechowywanych w wierszu i wykonywanie łączenia. Wyniki są przechowywane w `dstPixelsArray` i zapisywane do parametru `dstOut Raster`:

```
// Dla każdego wiersza w obszarze grafiki
for (int y = 0; y < height; y++) {
    // Odczytanie tablicy pikseli z rastrów wejściowych
    src.getDataElements(0, y, width, 1, srcPixelsArray);
    dstIn.getDataElements(0, y, width, 1, dstPixelsArray);

    // Dla każdego piksela w wierszu
    for (int x = 0; x < width; x++) {
        // Wyodrębnienie komponentów koloru
        toRGBArray(srcPixelsArray[x], srcPixel);
        toRGBArray(dstPixelsArray[x], dstPixel);

        // Wykonanie mieszania
        result[0] = Math.min(255, srcPixel[0] + dstPixel[0]);
        result[1] = Math.min(255, srcPixel[1] + dstPixel[1]);
        result[2] = Math.min(255, srcPixel[2] + dstPixel[2]);
    }
}
```

```
        result[3] = Math.min(255, srcPixel[3] + dstPixel[3]);  
        // Zapisanie wyniku  
        dstPixelsArray[x] = fromRGBArray(result);  
    }  
    // Zapisanie wiersza pikseli do rastra docelowego  
    dstOut.setDataElements(0, y, width, 1, dstPixelsArray);  
}
```

Implementowanie algorytmu łączenia wymaga napisania dużej ilości kodu bazowego. Jeżeli przyjrzymy się dokładniej klasom `AddComposite` oraz `AddContext`, można zauważyć, że tylko cztery wiersze kodu są związane z równaniami definiującymi sposób łączenia. Projekt `Blend-Composities` implementuje 32 metody łączenia w mniej niż 600 wierszach kodu przez utworzenie ogólnych klas `Composite` oraz `Context` i przez zdefiniowanie każdego trybu mieszania za pomocą czterech wierszy kodu realizujących faktycznie to zadanie.

Podsumowanie

Operacje łączenia są początkowo trudne do zrozumienia, ale szybko udowadniają swoją przydatność w wielu sytuacjach. Przez tworzenie własnych klas realizujących łączenie można zrealizować funkcje, o których twórcy JDK nawet nie pomyśleli, i bez trudu powielić najbardziej popularne funkcje aplikacji edycji grafiki, takich jak Adobe Photoshop.