Wejdź do świata twórców oprogramowania dla urządzeń mobilnych!

Android Flash

Zaawansowane programowanie aplikacji mobilnych

Stephen Chin • Dean Iverson • Oswald Campesato • Paul Trani

Apress®

Helion

Tytuł oryginału: Pro Android Flash

Tłumaczenie: Łukasz Schmidt

ISBN: 978-83-246-3920-5

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

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

Printed in Poland.

Spis treści

ROZDZIAŁ 4

Grafika i animacja

Grafiki są atrakcyjne dla osób w każdym wieku. Jeśli lubisz grafiki tak bardzo jak my, to ucieszysz się z tego, że grafiki oparte na Fleksie, które działają w przeglądarce komputera biurkowego, będą także poprawnie wyświetlane na urządzeniu mobilnym. Poza tym przy tworzeniu aplikacji związanych z grafiką dla urządzeń mobilnych można również wykorzystać zdarzenia dotykowe i gesty (które opisaliśmy w rozdziale 2.).

W pierwszej części niniejszego rozdziału pokażemy, w jaki sposób renderować różne dwuwymiarowe kształty, takie jak prostokąty, elipsy, krzywe Béziera i ścieżki. Druga część rozdziału zawiera przykłady kodu renderującego obiekty geometryczne wypełnione gradientami liniowymi i radialnymi. W trzeciej części rozdziału przedstawimy przykłady wykorzystywania filtrów, takich jak `Blur`, `DropShadow` i `Glow`.

Zobaczysz także wycinki kodu ilustrujące, jak wykonywać przekształcenia (przesunięcia, zmiana rozmiaru, obracanie i przekształcenia poprzeczne) obiektów graficznych omówionych w pierwszej części rozdziału. Następnie dowiesz się, jak renderować wykresy i grafy za pomocą komponentów MX. Ostatnim przykładem w tym rozdziale będzie aplikacja do szkicowania, w której wykorzystamy omówione wcześniej elementy. Program ten będzie obsługiwał także zdarzenia dotykowe, zapewni również możliwość rysowania na obrazie w formacie JPG i opcję zapisu rysunków w postaci plików JPG na urządzeniu mobilnym.

Dzięki lekturze tego rozdziału dobrze zrozumiesz możliwości, jakie są związane z wykorzystaniem grafiki w aplikacjach dla urządzeń mobilnych. Być może niektóre przykłady w tym rozdziale zainspirują Cię do napisania własnych aplikacji z efektownymi grafikami!

Wykorzystywanie prostych obiektów Spark do rysowania grafiki 2D

W tym podrozdziale pokażemy, jak renderować różne obiekty 2D, takie jak prostokąty, elipsy, krzywe Béziera, wielokąty i ścieżki. Dodatkowo w niektórych przykładowych aplikacjach będziesz mógł obejrzeć utworzone przez nas obrazy — w ten sposób porównasz wyniki działania prezentowanego kodu.

Rysowanie prostokątów i elips

Zacznijmy od narysowania dwóch prostokątów i elipsy — dwóch znanych każdemu obiektów geometrycznych. Utwórz nowy projekt mobilny Flex, nadaj mu nazwę `RectEllipse1` — skorzystaj z szablonu aplikacji mobilnej i dodaj kod pokazany w listingu 4.1.

Listing 4.1. *Renderowanie dwóch prostokątów oraz elipsy*

```
<?xml version="1.0" encoding="utf-8"?>
<s:View xmlns:fx="http://ns.adobe.com/mxml/2009"
         xmlns:s="library://ns.adobe.com/flex/spark" title="HomeView">
   <s:Rect id="rect1" x="10" y="10" width="250" height="200">
      <s:fill>
         <s:SolidColor color="0xFF0000"/>
      </s:fill>
      <s:stroke>
         <s:SolidColorStroke color="0xFFFF00" weight="4"/>
      </s:stroke>
   </s:Rect>

   <s:Ellipse id="ellipse1" x="10" y="220" width="250" height="200">
      <s:fill>
         <s:SolidColor color="0x0000FF"/>
      </s:fill>
      <s:stroke>
         <s:SolidColorStroke color="0xFF0000" weight="4"/>
      </s:stroke>
   </s:Ellipse>

   <s:Rect id="rect2" x="10" y="460" width="250" height="100">
      <s:fill>
         <s:SolidColor color="0xFFFF00"/>
      </s:fill>
      <s:stroke>
         <s:SolidColorStroke color="0x0000FF" weight="8"/>
      </s:stroke>
   </s:Rect>

   <fx:Declarations>
      <!-- Tu umieść elementy niewizualne -->
   </fx:Declarations>
</s:View>
```

Kod w listingu 4.1 zaczyna się od elementu XML `Rect`, który określa wartości atrybutów `id`, `x`, `y`, `width` i `height`. Warto zauważyć, że element XML `Rect` zawiera elementy XML `fill` i `stroke` zamiast atrybutów `fill` i `stroke`, co odróżnia go od SVG, gdzie wartości `fill` i `stroke` są ustalane za pomocą atrybutów. Jednak element XML `stroke` zawiera element potomny XML `SolidColorStroke`, który wskazuje wartości `color` i `weight` jako atrybuty, a nie wartości elementów XML. Zauważ też, że w SVG istnieją atrybuty `stroke` i `stroke-width` zamiast atrybutów `color` i `weight`.

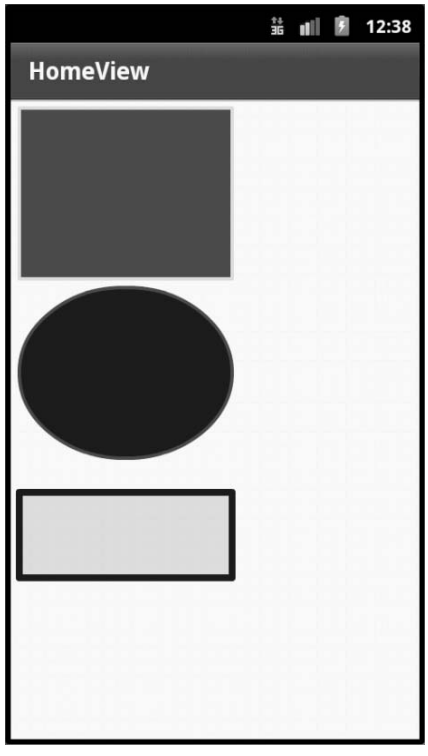
Kod w listingu 4.1 zawiera także element XML `Ellipse` definiujący elipsę z niemal identycznymi atrybutami i wartościami jak element XML `Rect`, ale generujący elipsę zamiast prostokąta.

Drugi element XML `Rect` jest podobny do pierwszego elementu `Rect`, ale ma inne kolory i położenie na ekranie.

Na rysunku 4.1 pokazano dwa prostokąty i elipsę, narysowane w wyniku działania kodu z listingu 4.1.

Stosowanie gradientów liniowych i radialnych

Aplikacje mobilne Flex obsługują gradienty liniowe i radialne. Jak sugeruje nazwa, gradient liniowy powstaje przez obliczenie barw pośrednich i nałożenie ich w sposób liniowy pomiędzy kolorem początkowym a końcowym. Na przykład: jeśli gradient zmienia się od koloru czarnego do czerwonego, początkowym kolorem jest czarny, a końcowym czerwony z liniowym przechodzeniem odcieni kolorów pomiędzy czarnym a czerwonym.



Rysunek 4.1. *Dwa prostokąty i elipsa*

Gradient radialny różni się od liniowego tym, że przechodzenie odcieni następuje w sposób radialny. Wyobraź sobie kroplę upadającą na powierzchnię wody: powstają kolejne okręgi o coraz większej średnicy — w podobny sposób są renderowane gradienty radialne.

Jako ilustrację powyższego napisaliśmy kod, który renderuje prostokąt wypełniony gradientem liniowym i elipsę wypełnioną gradientem radialnym. Utwórz zatem nowy projekt mobilny Flex, nadaj mu nazwę `LinearRadial1` — skorzystaj z szablonu aplikacji mobilnej i dodaj kod pokazany w listingu 4.2.

Listing 4.2. *Stosowanie gradientów liniowych i radialnych*

```
<?xml version="1.0" encoding="utf-8"?>
<s:View xmlns:fx="http://ns.adobe.com/mxml/2009"
      xmlns:s="library://ns.adobe.com/flex/spark"
      title="HomeView">

   <s:Panel width="100%" height="100%"
           title="Gradienty liniowy i radialny">
      <s:Group>
         <s:Rect id="rect1" x="10" y="10"
                 height="250" width="300">
         <s:fill>
            <s:LinearGradient>
               <s:GradientEntry color="0xFF0000"
                        ratio="0"   alpha=".5"/>
               <s:GradientEntry color="0xFFFF00"
                        ratio=".33" alpha=".5"/>
               <s:GradientEntry color="0x0000FF"
                        ratio=".66" alpha=".5"/>
```

```
            </s:LinearGradient>
         </s:fill>

         <s:stroke>
            <s:SolidColorStroke color="0x000000" weight="2"/>
         </s:stroke>
      </s:Rect>

      <s:Ellipse id="ellipse1" x="10" y="270"
                 height="250" width="300">
        <s:fill>
          <s:RadialGradient>
            <s:GradientEntry color="0xFF0000"
                     ratio="0"  alpha="1"/>
            <s:GradientEntry color="0xFFFF00"
                     ratio=".9" alpha="1"/>
          </s:RadialGradient>
        </s:fill>

            <s:stroke>
               <s:SolidColorStroke color="0x000000" weight="2"/>
            </s:stroke>
         </s:Ellipse>
      </s:Group>
   </s:Panel>

   <fx:Declarations>
      <!-- Tu umieść elementy niewizualne -->
   </fx:Declarations>
</s:View>
```

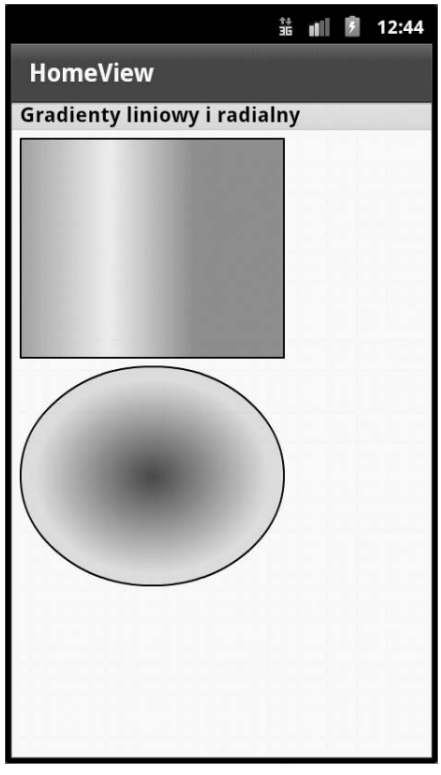
Kod w listingu 4.2 zawiera jeden element XML `Panel`. W tym elemencie znalazł się jeden element XML `Group`, którego atrybuty określają układ panelu. Element XML `Group` zawiera dwa elementy potomne: `Rect` i `Ellipse`. Element XML `Rect` definiuje prostokąt wypełniony gradientem liniowym:

```
<s:Rect id="rect1" x="10" y="10" height="250" width="300">
   <s:fill>
      <s:LinearGradient>
         <s:GradientEntry color="0xFF0000" ratio="0" alpha=".5"/>
         <s:GradientEntry color="0xFFFF00" ratio=".33" alpha=".5"/>
         <s:GradientEntry color="0x0000FF" ratio=".66" alpha=".5"/>
      </s:LinearGradient>
   </s:fill>

   <s:stroke>
      <s:SolidColorStroke color="0x000000" weight="2"/>
   </s:stroke>
</s:Rect>
```

Powyższy element XML `Rect` określa wartości atrybutów `id`, `x`, `y`, `width` i `height`. Element ten zawiera również element XML `Fill` (jak widziałeś już w poprzednim przykładzie), w którym z kolei znajduje się element XML `LinearGradient` określający trzy elementy XML `GradientEntry`. Każdy z nich wskazuje wartość dziesiętną (pomiędzy 0 i 1) dla atrybutów `ratio` i `alpha`. Ostatnia część elementu XML `Rect` zawiera element XML `Stroke`, w którym z kolei znajduje się element XML `SolidStrokeElement` określający wartości atrybutów `color` i `weight`.

W kodzie z listingu 4.2 znajduje się także element XML `Ellipse`, który definiuje elipsę z gradientem radialnym. Ten kod zawiera niemal te same atrybuty i wartości, co element XML `Rect`, ale reprezentuje elipsę zamiast prostokąta.

Na rysunku 4.2 pokazano prostokąt wypełniony gradientem liniowym i elipsę wypełnioną gradientem radialnym.



Rysunek 4.2. *Prostokąt wypełniony gradientem liniowym i elipsa wypełniona gradientem radialnym*

Renderowanie sześciennych krzywych Béziera

Flex obsługuje sześcienne krzywe Béziera (krzywe trzeciego stopnia o dwóch punktach końcowych i dwóch punktach kontrolnych) i kwadratowe krzywe Béziera (krzywe drugiego stopnia o dwóch punktach końcowych i jednym punkcie kontrolnym). Sześcienną krzywą Béziera łatwo odróżnić, ponieważ jej definicja zaczyna się od litery *C* (lub *c*), podczas gdy definicja kwadratowej krzywej Béziera zaczyna się od litery *Q* (lub *q*). Wielkie litery *C* i *Q* wskazują bezwzględne położenie, a małe litery *c* i *q* położenie względem poprzedzającego punktu w elemencie XML `Path`.

Pierwszy z punktów wskazanych dla sześciennej lub kwadratowej krzywej Béziera jest pierwszym punktem kontrolnym, za nim następuje drugi punkt kontrolny w przypadku sześciennej krzywej Béziera, a potem drugi punkt końcowy. Pierwszy punkt końcowy w kwadratowej i sześciennej krzywej Béziera jest poprzedzającym punktem wskazanym w elemencie XML `Path`. Jeśli punkt taki nie zostanie wskazany, początek układu współrzędnych (0, 0) służy za pierwszy punkt końcowy.

Sekwencję dla krzywych Béziera można także wskazać za pomocą litery *S* (dla krzywej sześciennej) lub *T* (dla krzywej kwadratowej).

Teraz utwórz nowy projekt mobilny Flex, nadaj mu nazwę `BezierCurves1` — skorzystaj z szablonu aplikacji mobilnej i dodaj kod pokazany w listingu 4.3. Kod ten tworzy cztery krzywe Béziera: sześcienną, kwadratową, dwie połączone sześcienne oraz sześcienną połączoną z kwadratową.

Listing 4.3. *Renderowanie sześciennych i kwadratowych krzywych Béziera*

```
<?xml version="1.0" encoding="utf-8"?>
<s:View xmlns:fx="http://ns.adobe.com/mxml/2009"
        xmlns:s="library://ns.adobe.com/flex/spark"
        title="HomeView">

   <s:Panel width="100%" height="100%"
            title="Sześcienne/kwadratowe krzywe Béziera">
      <!-- sześcienna krzywa Béziera -->
      <s:Path data="C 100 150 200 20 300 100">
         <s:fill>
            <s:LinearGradient rotation="90">
               <s:GradientEntry color="#FFFFFF" alpha="0.5"/>
               <s:GradientEntry color="#FF0000" alpha="0.5"/>
            </s:LinearGradient>
         </s:fill>
            <s:stroke>
               <s:SolidColorStroke color="0x0000FF" weight="4"/>
            </s:stroke>
      </s:Path>

      <!-- kwadratowa krzywa Béziera -->
      <s:Path data="Q 250 200 100 300">
         <s:fill>
            <s:RadialGradient rotation="90">
               <s:GradientEntry color="#000000" alpha="0.8"/>
               <s:GradientEntry color="#0000FF" alpha="0.8"/>
            </s:RadialGradient>
         </s:fill>

         <s:stroke>
            <s:SolidColorStroke color="0xFF0000" weight="8"/>
         </s:stroke>
      </s:Path>

      <!-- dwie połączone sześcienne krzywe Béziera -->
      <s:Path data="C 100 300 200 20 300 100 S 250 200 300 250">
         <s:fill>
            <s:LinearGradient rotation="90">
               <s:GradientEntry color="#FF0000" alpha="0.5"/>
               <s:GradientEntry color="#FFFF00" alpha="0.5"/>
            </s:LinearGradient>
         </s:fill>

         <s:stroke>
            <s:SolidColorStroke color="0x00FF00" weight="2"/>
         </s:stroke>
      </s:Path>

      <!-- dwie połączone krzywe Béziera: sześcienna i kwadratowa -->
      <s:Path data="C 250 400 200 150 350 100 T 250 250 400 280">
         <s:fill>
            <s:LinearGradient rotation="90">
               <s:GradientEntry color="#FFFF00" alpha="0.5"/>
               <s:GradientEntry color="#FF0000" alpha="0.5"/>
            </s:LinearGradient>
         </s:fill>
```

```
        <s:stroke>
          <s:SolidColorStroke color="0x000000" weight="4"/>
        </s:stroke>
      </s:Path>
   </s:Panel>
</s:View>
```

Kod w listingu 4.3 zawiera element XML `Panel`, w którym z kolei znajdują się cztery elementy XML `Path` tworzące krzywe Béziera z różnymi rodzajami cieniowania. Pierwszy element XML `Path` definiuje sześcienną krzywą Béziera w następujący sposób:

```
<s:Path data="C 100 300 200 20 300 100 S 250 200 300 250">
   [inne elementy pominięte]
</s:Path>
```

Pierwszym punktem końcowym tej krzywej jest (0, 0), ponieważ nie zostały wskazane inne współrzędne; punkty kontrolne to (100, 300) i (200, 20), docelowym punktem końcowym jest (300, 100).

Element XML `Path` zawiera element XML `LinearGradient` definiujący przejście barw od białej do czerwonej, z poziomem krycia `0.5` oraz z niebieskim obrysem o szerokości linii równej `4`, tak jak poniżej:

```
<s:LinearGradient rotation="90">
   <s:GradientEntry color="#FFFFFF" alpha="0.5"/>
   <s:GradientEntry color="#FF0000" alpha="0.5"/>
</s:LinearGradient>
</s:fill>
<s:stroke>
   <s:SolidColorStroke color="0x0000FF" weight="4"/>
</s:stroke>
```

Drugi element XML `Path` definiuje kwadratową krzywą Béziera, której pierwszym punktem końcowym jest (0, 0), ponieważ nie zostały wskazane inne współrzędne; punkt kontrolny to (250, 200), a docelowym punktem końcowym jest (100, 300). Element XML `Path` zawiera element XML `LinearGradient` z przejściem barw od czarnej do niebieskiej i poziomem krycia `0.8`.

Trzeci element XML `Path` definiuje sześcienną krzywą Béziera, która jest „powiązana” z drugą sześcienną krzywą Béziera, tak jak poniżej:

```
<s:Path data="C 100 300 200 20 300 100 S 250 200 300 250">
   [inne elementy pominięte]
</s:Path>
```

Dwoma punktami kontrolnymi dla tej sześciennej krzywej Béziera są (100, 300) i (20, 300), punktem docelowym jest (300, 100). Druga część elementu XML `Path` definiuje kwadratową krzywą Béziera z punktem kontrolnym (250, 200) i z docelowym punktem końcowym (300, 250).

Element XML `Path` zawiera element XML `LinearGradient` tworzący gradient liniowy z przejściem od barwy żółtej do czerwonej, a następnie element XML `Stroke` z kolorem czarnym i o szerokości linii równej 4 jednostkom.

Ostatni element XML `Path` definiuje sześcienną krzywą Béziera, a następnie drugą sześcienną krzywą Béziera tak jak poniżej:

```
<s:Path data="C 250 300 200 150 350 100 T 250 250 400 280">
   [inne elementy pominięte]
</s:Path>
```

Dwa punkty kontrolne dla tej sześciennej krzywej Béziera to (250, 300) i (200, 150), punktem docelowym jest (350, 100). Druga część elementu XML `Path` definiuje kwadratową krzywą Béziera, której punktem kontrolnym jest (250, 250), a docelowym punktem końcowym (400, 280).

Element XML `Path` zawiera element XML `LinearGradient` tworzący gradient liniowy — przejście od barwy żółtej do czerwonej, z poziomem krycia `0.5`, a następnie element XML `Stroke` z kolorem czarnym i o szerokości linii równej 4 jednostkom.

Na rysunku 4.3 pokazano sześcienne, kwadratowe oraz połączone krzywe Béziera.

Rysunek 4.3. *Sześcienne, kwadratowe oraz połączone krzywe Béziera*

Inny przykład elementu Path

W poprzednim przykładowym fragmencie kodu pokazaliśmy, jak używać elementu `Path` do renderowania zestawu krzywych Béziera. Element `Path` umożliwia także łączenie innych obiektów 2D, takich jak segmenty linii i krzywe Béziera, z gradientami liniowymi i radialnymi. Utwórz teraz nowy projekt mobilny Flex, nadaj mu nazwę `Path1` — skorzystaj z szablonu aplikacji mobilnej i dodaj kod pokazany w listingu 4.4.

Listing 4.4. *Łączenie segmentów linii i krzywych Béziera*

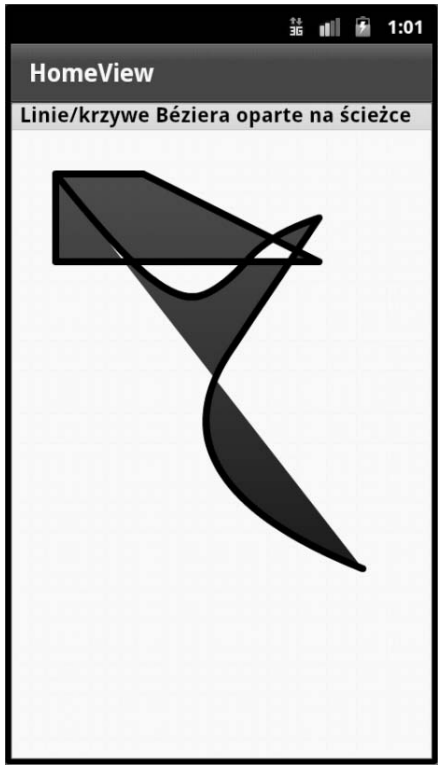
```
<?xml version="1.0" encoding="utf-8"?>
<s:View xmlns:fx="http://ns.adobe.com/mxml/2009"
         xmlns:s="library://ns.adobe.com/flex/spark"
         title="HomeView">

   <s:Panel width="100%" height="100%"
             title="Linie/krzywe Béziera oparte na ścieżce">
      <s:Path data="M 50 50 L150 50 350 150 50 150z
                  C 250 300 200 150 350 100 T 250 250 400 500">
         <s:fill>
            <s:LinearGradient rotation="90">
               <s:GradientEntry color="#FF0000" alpha="1"/>
               <s:GradientEntry color="#0000FF" alpha="1"/>
            </s:LinearGradient>
         </s:fill>

         <s:stroke>
            <s:SolidColorStroke color="0x000000" weight="8"/>
         </s:stroke>
      </s:Path>
   </s:Panel>
</s:View>
```

Element XML `Panel` z kodu w listingu 4.4 zawiera jeden element XML `Path`, w którym fragmenty linii posłużyły do wyrenderowania trapezoidu, a następnie pary sześciennych krzywych Béziera. Atrybut `data` elementu XML `Path` jest następujący:

```
<s:Path data="M 50 50 L150 50 350 150 50 150z
        C 250 300 200 150 350 100 T 250 250 400 280">
```

Część pierwsza atrybutu `data` (zaczynająca się literą `M`) definiuje trapezoid, część druga (zaczynająca się literą `C`) renderuje sześcienną krzywą Béziera, część trzecia (zaczynająca się literą `T`) definiuje kolejną taką krzywą.

Na rysunku 4.4 pokazano narysowane elementy: trapezoid i dwie sześcienne krzywe Béziera.



Rysunek 4.4. *Elementy oparte na ścieżkach: trapezoid i krzywe Béziera*

Stosowanie filtrów Spark

Aby znacząco poprawić wygląd aplikacji pisanych we Fleksie, warto wykorzystać filtry Flex — są bardzo przydatne przy tworzeniu zaawansowanych efektów wizualnych. Proste obiekty Spark (z pakietu `primitives`) obsługują różnorodne filtry, w tym `Blur`, `DropShadow` i `Glow`. Wszystkie filtry należą do pakietu `spark.filters`.

A zatem utwórz nowy projekt mobilny Flex, nadaj mu nazwę `RectLGradFilters3` — skorzystaj z szablonu aplikacji mobilnej i dodaj kod pokazany w listingu 4.5.

Listing 4.5. *Rysowanie prostokątów z wykorzystaniem filtrów Spark*

```
<?xml version="1.0" encoding="utf-8"?>
<s:View xmlns:fx="http://ns.adobe.com/mxml/2009"
        xmlns:s="library://ns.adobe.com/flex/spark"
        title="Prostokąt: gradient i filtry">

   <s:Rect id="rect1" x="50" y="50" height="300" width="250">
```

```
      <s:fill>
         <s:LinearGradient>
            <s:GradientEntry color="0xFF0000"
                             ratio="0" alpha=".5"/>
            <s:GradientEntry color="0xFFFF00"
                             ratio=".33" alpha=".5"/>
            <s:GradientEntry color="0x0000FF"
                             ratio=".66" alpha=".5"/>
         </s:LinearGradient>
      </s:fill>
      <s:stroke>
         <s:SolidColorStroke color="0xFF0000" weight="2"/>
      </s:stroke>
      <s:filters>
         <s:DropShadowFilter distance="80" color="#0000FF"/>
         <s:BlurFilter/>
         <s:GlowFilter/>
      </s:filters>
   </s:Rect>
</s:View>
```

W kodzie z listingu 4.5 znalazł się element XML `Rect` definiujący prostokąt wyrenderowany z gradientem liniowym. Atrybut `ratio` to liczba dziesiętna z przedziału od 0 do 1, określająca ułamek odległości pomiędzy punktem startowym a docelowym, w której zostanie utworzone przejście koloru. W listingu 4.5 element `GradientEntry` posiada atrybut `ratio` o wartości `0`, co oznacza, że prostokąt jest renderowany w kolorze 0×FF0000 (szesnastkowa wartość koloru czerwonego). Drugi element `GradientEntry` posiada atrybut `ratio` o wartości `0.33`, co oznacza, że prostokąt jest renderowany w kolorze 0×FFFF00 (szesnastkowa wartość oznaczająca żółtą barwę) od punktu, który znajduje się w 33% odległości od punktu początkowego do docelowego. Trzeci element `GradientEntry` posiada atrybut `ratio` o wartości `0.33`, co oznacza, że prostokąt jest renderowany w kolorze `0x0000FF` (szesnastkowa wartość oznaczająca barwę niebieską) od punktu znajdującego się w 66% odległości od punktu początkowego do docelowego.

Atrybut `alpha` oznacza wartość krycia. Jest to wartość dziesiętna z przedziału od 0 (element niewidoczny) do 1 (element całkowicie widoczny). Trzy elementy `GradientEntry` w listingu 4.5 mają wartość atrybutu `alpha` równą `0.5`, a więc prostokąt jest częściowo widoczny. Wypróbuj różne wartości atrybutów `ratio` i `alpha` — sprawdź, które z nich dają atrakcyjny wizualnie efekt.

Ostatnia część elementu XML `Rect` zawiera element XML `stroke`, który definiuje kolor jako czerwony i szerokość obrysu `2`. Za nim zdefiniowano trzy następujące filtry Spark:

```
<s:filters>
   <s:DropShadowFilter distance="80" color="#0000FF"/>
   <s:BlurFilter/>
   <s:GlowFilter/>
</s:filters>
```

Trzy filtry Spark wykorzystane w tym przykładzie noszą nazwy wskazujące na uzyskiwany za ich pomocą efekt. Pierwszy filtr, `DropShadowFilter`, dodaje efekt cienia do prostokąta określonego w elemencie XML `Rect`. Drugi filtr, `BlurFilter`, dodaje efekt rozmycia. Ostatni z filtrów, `GlowFilter`, tworzy efekt poświaty.

Na rysunku 4.5 pokazano prostokąt wypełniony gradientem liniowym, dla którego zastosowano trzy filtry Spark.

Stosowanie przekształceń obiektów geometrycznych

W tym podrozdziale pokażemy, jak stosować przekształcenia obiektów geometrycznych, w tym tych, o których powiedzieliśmy we wcześniejszej części tego rozdziału. Proste obiekty geometryczne Spark obsługują następujące efekty i przekształcenia:

Rysunek 4.5. *Prostokąt z gradientem liniowym, dla którego zastosowano trzy filtry Spark*

- `Animate`
- `AnimateColor`
- `AnimateFilter`
- `AnimateShaderTransition`
- `AnimateTransform`
- `Fade`
- `Move`
- `Resize`
- `Rotate`
- `Scale`
- `Wipe`
- `CrossFade`

Proste obiekty geometryczne Spark są dostępne w pakiecie `spark.effects` i mogą być zastosowane do komponentów Spark oraz komponentów MX. Pakiet `mx.effects` (dołączony do Flex 4 SDK) udostępnia efekty odpowiadające funkcjonalnie komponentom Spark, które można zastosować dla komponentów MX.

W dalszej części rozdziału zamieściliśmy przykłady kodu Flex ilustrujące, jak animować zmiany rozmiaru obiektu.

Dodawanie efektu zmiany rozmiaru

Efekty zmian rozmiaru (czyli powiększanie lub zmniejszanie odpowiednich wymiarów obiektu) mogą być przydatne w grach i są bardzo łatwe w stosowaniu w aplikacjach Flex. Utwórz więc nowy projekt mobilny Flex, nadaj mu nazwę `ScaleEffect1` — skorzystaj z szablonu aplikacji mobilnej i dodaj kod pokazany w listingu 4.6.

Listing 4.6. *Dodawanie efektów zmian rozmiaru za pomocą gradientów liniowych*

```
<?xml version="1.0" encoding="utf-8"?>
<s:View xmlns:fx="http://ns.adobe.com/mxml/2009"
        xmlns:s="library://ns.adobe.com/flex/spark"
```

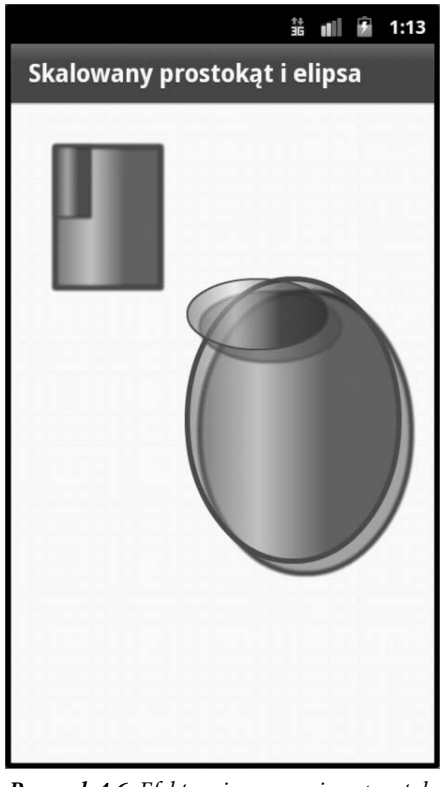
```
        title="Skalowany prostokąt i elipsa">

   <fx:Library>
      <fx:Definition name="MyRect1">
         <s:Rect x="50" y="50" height="40" width="20">
            <s:fill>
               <s:LinearGradient>
                  <s:GradientEntry color="0xFF0000"
                                   ratio="0"   alpha=".5"/>
                  <s:GradientEntry color="0xFFFF00"
                                   ratio=".33" alpha=".5"/>
                  <s:GradientEntry color="0x0000FF"
                                   ratio=".66" alpha=".5"/>
               </s:LinearGradient>
           </s:fill>
           <s:stroke>
              <s:SolidColorStroke color="0xFF0000" weight="1"/>
           </s:stroke>
           <s:filters>
              <s:BlurFilter/>
              <s:GlowFilter/>
           </s:filters>
        </s:Rect>
      </fx:Definition>

      <fx:Definition name="MyEllipse1">
         <s:Ellipse x="200" y="200" height="40" width="80">
            <s:fill>
               <s:LinearGradient>
                  <s:GradientEntry color="0xFF0000"
                             ratio="0"   alpha=".5"/>
                  <s:GradientEntry color="0xFFFF00"
                             ratio=".33" alpha=".5"/>
                  <s:GradientEntry color="0x0000FF"
                             ratio=".66" alpha=".5"/>
               </s:LinearGradient>
            </s:fill>
            <s:stroke>
               <s:SolidColorStroke color="0xFF0000" weight="1"/>
            </s:stroke>
            <s:filters>
               <s:DropShadowFilter distance="20" color="#FF0000"/>
            </s:filters>
         </s:Ellipse>
      </fx:Definition>
   </fx:Library>

   <s:Group>
      <fx:MyRect1    scaleX="6" scaleY="4"/>
      <fx:MyEllipse1 scaleX="3" scaleY="8"/>
      <fx:MyRect1    scaleX="2" scaleY="2"/>
      <fx:MyEllipse1 scaleX="2" scaleY="2"/>
   </s:Group>
</s:View>
```

Listing 4.6 zawiera dwa elementy XML `Definition`. Jeden z nich określa element XML `Rect` zawierający definicję prostokąta, a drugi — element XML `Ellipse` z definicją elipsy. Element XML `Group` zawiera dwie referencje do prostokąta i dwie referencje do elipsy, jak pokazano poniżej:

```
<s:Group>
   <fx:MyRect1 scaleX="6" scaleY="4"/>
   <fx:MyEllipse1 scaleX="3" scaleY="8"/>
   <fx:MyRect1 scaleX="2" scaleY="2"/>
   <fx:MyEllipse1 scaleX="2" scaleY="2"/>
</s:Group>
```

Pierwszy element XML zapewnia przeskalowanie zdefiniowanego wcześniej prostokąta `MyRect1` poprzez ustawienie wartości `6` i `4` dla atrybutów `scaleX` i `scaleY`. Drugi element XML zapewnia przeskalowanie zdefiniowanej wcześniej elipsy `MyEllipse1` poprzez ustawienie wartości `3` i `8` dla atrybutów `scaleX` i `scaleY`.

Na rysunku 4.6 pokazano efekt zmiany rozmiaru prostokąta i elipsy.



Rysunek 4.6. *Efekt zmiany rozmiaru prostokąta i elipsy*

Dodawanie animacji za pomocą biblioteki Spark

W tym podrozdziale przedstawimy aplikację, na przykładzie której pokażemy, jak animować obiekty geometryczne, o których mówiliśmy we wcześniejszej części rozdziału. Efektami animacji z biblioteki Spark `primitives` są:

- `Animate`
- `AnimateColor`
- `AnimateFilter`
- `AnimateShaderTransition`
- `AnimateTransform`

- `CrossFade`
- `Fade`
- `Move`
- `Resize`
- `Rotate`
- `Scale`
- `Wipe`

W kolejnych punktach, wykorzystując sekwencje kodu, pokażemy, w jaki sposób używać elementu XML `Animate` i jak definiować animacje odtwarzane równolegle i sekwencyjnie.

Stosowanie elementu Animate

Animacje są oczywiście bardzo popularne w grach, ale mogą być efektywnie wykorzystywane także w innych rodzajach aplikacji. Jednocześnie powinieneś pamiętać, że w aplikacjach biznesowych prawdopodobnie lepiej będzie nie nadużywać elementów animowanych.

Utwórz nowy projekt mobilny Flex, nadaj mu nazwę `AnimPropertyWidth` — skorzystaj z szablonu aplikacji mobilnej i dodaj kod pokazany w listingu 4.7.

Listing 4.7. *Animowanie szerokości prostokąta*

```
<?xml version="1.0" encoding="utf-8"?>
<s:View xmlns:fx="http://ns.adobe.com/mxml/2009"
        xmlns:s="library://ns.adobe.com/flex/spark"
        title="Animowanie szerokości prostokąta">

   <fx:Declarations>
      <s:Animate id="MyAnimate1">
         <s:motionPaths>
            <s:MotionPath property="width">
               <s:keyframes>
                  <s:Keyframe time="0"    value="200"/>
                  <s:Keyframe time="2000" value="400"/>
               </s:keyframes>
            </s:MotionPath>
         </s:motionPaths>
      </s:Animate>
   </fx:Declarations>

   <s:VGroup>
      <s:Rect id="rect1" height="300" width="200">
         <s:fill>
            <s:LinearGradient>
               <s:GradientEntry color="0xFF0000"
                                ratio="0"   alpha=".5"/>
               <s:GradientEntry color="0xFFFF00"
                                ratio=".33" alpha=".5"/>
               <s:GradientEntry color="0x0000FF"
                                ratio=".66" alpha=".5"/>
            </s:LinearGradient>
         </s:fill>
         <s:stroke>
            <s:SolidColorStroke color="0xFF0000" weight="2"/>
         </s:stroke>
      </s:Rect>
```

```
      <s:Button id="MyButton1" label="Animuj szerokość"
                click="MyAnimate1.play([rect1])"
                bottom="150" right="50">
      </s:Button>
   </s:VGroup>
</s:View>
```

Kod w listingu 4.7 zawiera element XML `Declarations`, w którym z kolei znajduje się element XML `Animate` definiujący szczegóły animacji. Element XML `Animate` posiada atrybut `id` o wartości `MyAnimate1`, do którego tworzymy referencję w procedurze obsługi zdarzenia `click`. Procedurę tę opiszemy w dalszej części tego podrozdziału.

W listingu 4.7 znajduje się także element XML `VGroup`, w którym z kolei znajduje się element XML `Rect`. Jego zawartość przypomina przykładowe elementy XML, jakie już widziałeś w tym rozdziale. W tym samym listingu znalazł się również element XML `Button` uruchamiający efekt animacji. Kiedy użytkownik kliknie lub stuknie ten przycisk, zostanie wykonana procedura obsługi zdarzenia o wartości atrybutu `id` `MyAnimate1`, która została zdefiniowana we wcześniejszej części kodu. Efekt animacji jest prosty: szerokość prostokąta zwiększa się z 200 do 400 jednostek w czasie dwóch sekund (2000 milisekund).

Na rysunkach 4.7 i 4.8 pokazano dwa ujęcia prostokąta, który przesuwa się poziomo po ekranie, kiedy użytkownik kliknie przycisk.

Rysunek 4.7. *Animowany prostokąt (położenie początkowe)*

Rysunek 4.8. *Animowany prostokąt (położenie końcowe)*

Animacje odtwarzane równolegle i sekwencyjnie

Flex obsługuje dwa sposoby odtwarzania animacji. Równoległe odtwarzanie animacji polega na wykonywaniu dwóch lub większej liczby animacji w tym samym czasie. Z kolei odtwarzanie animacji sekwencyjnie polega na wykonywaniu dwóch lub większej liczby animacji jedna po drugiej, co oznacza, że w danej chwili wykonywana jest tylko jedna z nich. Mając to na uwadze, utwórz nowy projekt mobilny Flex, nadaj mu nazwę `SequentialAnimation1` — skorzystaj z szablonu aplikacji mobilnej i dodaj kod pokazany w listingu 4.8.

Listing 4.8. *Tworzenie animacji odtwarzanej sekwencyjnie*

```
<?xml version="1.0" encoding="utf-8"?>
<s:View xmlns:fx="http://ns.adobe.com/mxml/2009"
        xmlns:s="library://ns.adobe.com/flex/spark"
        title="Animacja sekwencyjna">

   <fx:Declarations>
      <s:Sequence id="transformer1" target="{button1}">
         <s:Move xFrom="50" xTo="150"
                 autoCenterTransform="true"/>
         <s:Rotate angleFrom="0" angleTo="360"
                   autoCenterTransform="true"/>
         <s:Scale scaleXFrom="1" scaleXTo="2"
                  autoCenterTransform="true"/>
      </s:Sequence>

      <s:Sequence id="transformer2" target="{button2}">
         <s:Move xFrom="50" xTo="150"
                 autoCenterTransform="true"/>
         <s:Scale scaleXFrom="1" scaleXTo="2"
                  autoCenterTransform="true"/>
         <s:Rotate angleFrom="0" angleTo="720"
                  autoCenterTransform="true"/>
      </s:Sequence>
  </fx:Declarations>

   <s:Rect id="rect1" x="10" y="10" width="400" height="400">
      <s:fill>
         <s:SolidColor color="0xFF0000"/>
      </s:fill>
      <s:stroke>
         <s:SolidColorStroke color="0x0000FF" weight="4"/>
      </s:stroke>
   </s:Rect>

   <s:Button id="button1" x="50" y="100" label="Przekształć mnie"
             click="transformer1.play()"/>

   <s:Button id="button2" x="50" y="200" label="Przekształć także mnie"
             click="transformer2.play()"/>
</s:View>
```

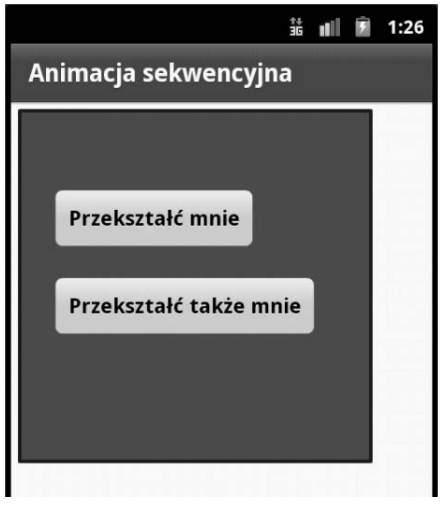
Listing 4.8 zawiera element XML `Declarations`, a ten z kolei trzy elementy XML `Seguence` definiujące trzy efekty przekształcenia. Wykonywanie animacji rozpoczyna się od elementu XML `Move` (efekt przesunięcia), następnym jest element XML `Rotate` (efekt obrotu) i wreszcie element XML `Scale` (efekt zmiany rozmiaru). Kiedy użytkownik stuknie pierwszy element XML `Button`, zostaną wywołane animacje zdefiniowane w elemencie XML `Sequence` o wartości atrybutu `id` równej `transformer1`.

Podobny opis mógłby dotyczyć drugiego elementu XML `Sequence` i drugiego przycisku, poza różnicą polegającą na tym, że animacja obejmuje dwa obroty zamiast jednego.

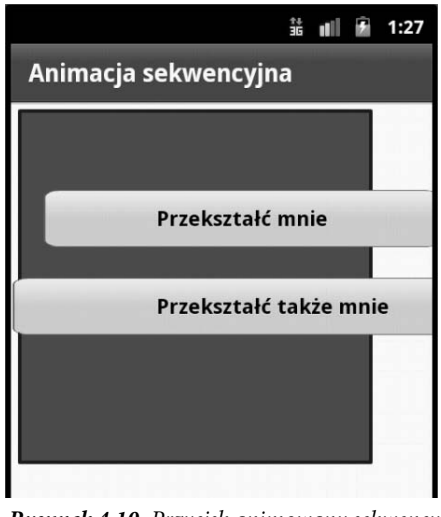
Warto zauważyć, że można łatwo zmienić efekt animacji z sekwencyjnego na równoległy poprzez zamianę elementu XML `Sequence` na element XML `Parallel`, tak jak poniżej:

```
<s:Parallel id="transformer" target="{button}">
   <s:Move xFrom="50" xTo="150"
           autoCenterTransform="true"/>
   <s:Rotate angleFrom="0" angleTo="360"
           autoCenterTransform="true"/>
   <s:Scale scaleXFrom="1" scaleXTo="2"
           autoCenterTransform="true"/>
</s:Parallel>
```

Na zrzutach ekranu przedstawionych na rysunkach 4.9 i 4.10 pokazano dwa przyciski animowane w sposób sekwencyjny. Na zrzutach ekranu pokazano tylko początkowy i końcowy stan animacji, a więc uruchom tę aplikację mobilną i sam zobacz, jak wyglądają animacje przesunięcia i obrotu.



Rysunek 4.9. *Przycisk animowany sekwencyjnie (stan początkowy)*



Rysunek 4.10. *Przycisk animowany sekwencyjnie (stan końcowy)*

Tworzenie animacji 3D

Flex obsługuje kilka rodzajów animacji 3D, w tym przesuwanie, obracanie i zmiany rozmiaru obrazu (pliku JPG). Efekt „ruchu" 3D polega na przesunięciu obrazu JPG oraz zmianie jego rozmiaru, a efekt zmiany rozmiaru 3D obejmuje zwiększenie (lub zmniejszenie) szerokości i wysokości obrazu JPG od wartości początkowej (zwykle 1) do ostatecznej (która może być większa lub mniejsza od 1). Efekt „obracania" 3D polega na przekształcaniu obrazu w taki sposób, że wydaje się on obracać w trzech wymiarach.

Przykładowy kod w listingu 4.9 pokazuje, jak dodawać efekty animacji 3D przesuwania, obracania i zmian rozmiaru pliku JPG w aplikacji mobilnej.

Listing 4.9. *Tworzenie animacji 3D*

```
<?xml version="1.0" encoding="utf-8"?>
<s:View xmlns:fx="http://ns.adobe.com/mxml/2009"
        xmlns:s="library://ns.adobe.com/flex/spark"
        title="Tworzenie efektów 3D">

   <fx:Declarations>
     <s:Move3D id="moveEffect" target="{targetImg}" xBy="100" zBy="100"
                repeatCount="2" repeatBehavior="reverse"
                effectStart="playMoveButton.enabled=false"
                effectEnd="playMoveButton.enabled=true;"/>

      <s:Rotate3D id="rotateEffect" target="{targetImg}"
                angleYFrom="0" angleYTo="360"
                repeatCount="4" repeatBehavior="reverse"
                effectStart="playRotateButton.enabled=false;"
                effectEnd="playRotateButton.enabled=true;"/>

      <s:Scale3D id="atScale" target="{targetImg}"
                scaleXBy="-.45" repeatCount="2"
                repeatBehavior="reverse"
                effectStart="playScaleButton.enabled=false"
                effectEnd="playScaleButton.enabled=true;"/>
  </fx:Declarations>

   <s:VGroup width="100%" height="100%" >
      <s:Image id="targetImg"
           horizontalCenter="0"
           verticalCenter="0"
           source="@Embed(source='images/Cassandra4.jpg')"/>

      <s:HGroup>
         <s:Button id="playMoveButton"
                   left="10" bottom="25"
                   label="Przesuń"
                   click="moveEffect.play();"/>

         <s:Button id="playRotateButton"
                   left="110" bottom="25"
                   label="Obróć"
                   click="rotateEffect.play();"/>

         <s:Button id="playScaleButton"
                   left="222" bottom="25"
                   label="Skaluj" click="atScale.play();"/>
```

```
      </s:HGroup>
   </s:VGroup>

</s:View>
```

Na rysunku 4.11 pokazano obraz JPG *Cassandra4.jpg* — zdjęcie Cassandry Chin (córki Stephena China). Obraz ten wykorzystaliśmy w kodzie, za pomocą którego zademonstrowaliśmy efekty animacji 3D.

Rysunek 4.11. *Obraz JPG do prezentowania efektów animacji 3D*

Kod z listingu 4.9 zawiera element XML `Declarations`, w którym znalazły się trzy elementy określające animacje 3D wraz z trzema elementami XML `Button`. Użytkownik może klikać te przyciski, aby zobaczyć efekt przekształcenia 3D. Elementami tymi są `Move3D`, który wskazuje położenie docelowe za pomocą atrybutów `xBy` i `zBy`, `reapeatCount` o wartości `2` (który powoduje dwukrotne wykonanie efektu animacji) oraz `repeatBehavior` o wartości `reverse` (który za każdym razem przekazuje informację o pierwotnym położeniu). Odpowiadający animacji element XML `Button` zawiera atrybut `label` o wartości `Przesuń` i atrybut `click` o wartości `moveEffect.play()` — ta funkcja wywołuje animację ruchu zdefiniowaną za pomocą elementu XML `MoveEffect`, który znajduje się w elemencie XML `Declarations`.

Efekt obrotu jest obsługiwany za pomocą elementu XML `Rotate3D`, którego atrybuty `angleYFrom` i `angleYTo` określają kąty początkowy i końcowy o wartościach odpowiednio: `0` i `360` (oznacza to, że obiekt wykona pełny obrót). Animacja obrotu jest wykonywana czterokrotnie. Element XML `Button` zawiera atrybut `label` o wartości `Obróć` i atrybut `clik` o wartości `rotateEffect.play()` — ta funkcja wywołuje animację obracania zdefiniowaną za pomocą elementu XML `Rotate3D`, który znajduje się w elemencie XML `Declarations`.

Efekt zmiany rozmiaru (trzeci i jednocześnie ostatni) jest obsługiwany za pomocą elementu XML `Scale3D`, który zawiera kilka atrybutów o wartościach określających szczegóły animowania obrazu JPG. Atrybut `id` ma wartość `atScale` i służy do tworzenia referencji do tego elementu w innych miejscach kodu. Atrybut `target` jest referencją do elementu XML, którego atrybut `id` ma wartość `targetImg`, co wskazuje na obraz JPG. Atrybut `scaleXBy` ma wartość `-0.25`, co powoduje zmniejszenie obrazu JPG o 25%. Atrybut `RepeatCount` ma wartość `4`, a `repeatBahavior` wartość `reverse`, co oznacza, że efekt animacji będzie odtwarzany czterokrotnie, na przemian od lewej do prawej i odwrotnie. Dwa pozostałe atrybuty to `effectStart` i `effectEnd` wskazujące zachowania na początku i na końcu animacji, w tym przypadku polegają one na wyłączeniu i włączeniu przycisku `playButton`.

Zauważ, że element XML `Image` wskazuje położenie obrazu *Cassandra4.jpg* w folderze, który znajduje się w katalogu najwyższego poziomu projektu. Aby mógł powstać układ, element XML `Image` znalazł się wewnątrz elementu XML `VGroup`, który zawiera element XML `HGroup`, gdzie z kolei znalazły się trzy elementy XML `Button`.

Na rysunku 4.12 pokazano obraz JPG po wykonaniu animacji 3D.

Rysunek 4.12. *Obraz JPG po wykonaniu animacji 3D*

Na rysunku 4.13 pokazano obraz JPG po wykonaniu animacji obrotu 3D.

Rysunek 4.13. *Obraz JPG po wykonaniu animacji obrotu 3D*

Na rysunku 4.14 pokazano obraz JPG po wykonaniu animacji zmiany rozmiaru 3D.

Rysunek 4.14. *Obraz JPG po wykonaniu animacji zmiany rozmiaru 3D*

Tworzenie skórek Spark

Własne skórki będą przydatne za każdym razem, kiedy zechcesz zapewnić użytkownikowi ciekawsze efekty wizualne dla określonych części aplikacji mobilnych. Na przykład możesz utworzyć kilka własnych skórek, za pomocą których wzbogacisz przycisk o efekty graficzne (w tym te, które poznałeś wcześniej w tym rozdziale). Poniżej zamieściliśmy przykładowy fragment kodu, na podstawie którego omówimy proces tworzenia własnych skórek Spark.

W listingach od 4.10 do 4.12 pokazaliśmy zawartość plików: *CustomSkinHomeView.mxml*, *ButtonSkin1.mxml* oraz *ButtonSkin2.mxml*.

Przed omówieniem plików MXML pokazanych w tym podrozdziale przyjrzyjmy się czynnościom, jakie należy wykonać, aby dodać do projektu plik *ButtonSkin1.mxml* (w pakiecie `skins`).

1. Dodaj folder `skins` do projektu.
2. Kliknij nazwę projektu prawym przyciskiem myszy i wybierz *New/MXML Skin* (nowy/skórka MXML).
3. Wskaż `skins` jako nazwę pakietu nowej skórki.
4. Wpisz `ButtonSkin1` jako nazwę skórki.
5. Wskaż `spark.components.Button` jako nazwę komponentu.
6. Usuń zaznaczenie obok pola *Create as a copy of:* (utwórz jako kopię).

Powtórz powyższe czynności dla skórki *ButtonSkin2.mxml*, a także dla wszelkich innych własnych skórek, które będziesz chciał dodać do tego projektu. Przyjrzyjmy się teraz zawartości pliku *CustomSkin.mxml* (listing 4.10).

Listing 4.10. *Tworzenie własnych skórek Spark*

```
<?xml version="1.0" encoding="utf-8"?>
<s:View xmlns:fx="http://ns.adobe.com/mxml/2009"
        xmlns:s="library://ns.adobe.com/flex/spark"
        title="Skórki własne">

   <s:VGroup>
      <s:Label text="To jest zwykły przycisk:" x="10" y="0"/>
```

```
      <s:Button label="Button1" x="10" y="25"/>

      <s:Label text="Pierwszy przycisk ze skórką:"   x="10" y="60"/>
      <s:Button skinClass="skins.ButtonSkin1" x="10" y="85"/>

      <s:Label text="Drugi przycisk ze skórką:"  x="10" y="100"/>
      <s:Button skinClass="skins.ButtonSkin2" x="10" y="125"/>

      <s:Label text="Trzeci przycisk ze skórką:"   x="10" y="140"/>
      <s:Button skinClass="skins.ButtonSkin1" x="10" y="165"/>

      <s:Label text="Czwarty przycisk ze skórką:"  x="10" y="180"/>
      <s:Button skinClass="skins.ButtonSkin2" x="10" y="205"/>
   </s:VGroup>
</s:View>
```

Kod w listingu 4.10 zawiera element XML `VGroup` z dziesięcioma ułożonymi w pary elementami XML, które renderują standardowe elementy XML `Label` i `Button`. Pierwszy przycisk jest zwykłym przyciskiem, definiowanym przez poniższy wycinek kodu:

```
<s:Label text="To jest zwykły przycisk:" x="10" y="0"/>
<s:Button label="Button1" x="10" y="25"/>
```

Działanie powyższych elementów XML jest proste: pierwszy to etykieta (*To jest zwykły przycisk*), a drugi renderuje przycisk.

Omówimy teraz pary elementów XML związane z przyciskami, na które nałożyliśmy skórki. Pierwszy element z pary wyświetla etykietę z tekstem *Pierwszy przycisk ze skórką:*, podczas gdy drugi renderuje element XML `Button` na podstawie zawartości skórki `ButtonSkin1` w pakiecie `skins`. Podobnie działa druga para elementów XML: pierwszy element wyświetla etykietę z tekstem *Drugi przycisk ze skórką:*, a drugi renderuje element XML `Button` na podstawie zawartości skórki `ButtonSkin2` w pakiecie `skins`. Podobna sytuacja zachodzi w przypadku dwóch pozostałych przycisków.

Przyjrzymy się teraz zawartości pliku *ButtonSkin1.mxml*, w którym znalazły się dane służące do renderowania drugiego przycisku (pierwszego ze skórką). Kod pokazano w listingu 4.11.

Listing 4.11. *Tworzenie skórki przycisku z grafiką*

```
<?xml version="1.0" encoding="utf-8"?>
<s:Skin xmlns:fx="http://ns.adobe.com/mxml/2009"
        xmlns:s="library://ns.adobe.com/flex/spark">

   <fx:Metadata>
      [HostComponent("spark.components.Button")]
   </fx:Metadata>

   <s:states>
      <s:State name="disabled" />
      <s:State name="down" />
      <s:State name="over" />
      <s:State name="up" />
   </s:states>

   <s:Rect id="rect1" x="0" y="0" height="40" width="100">
      <s:fill>
         <s:LinearGradient>
            <s:GradientEntry color="0xFF0000"
                             ratio="0"   alpha=".5"/>
            <s:GradientEntry color="0xFFFF00"
                             ratio=".33" alpha=".5"/>
```

```
            <s:GradientEntry color="0x0000FF"
                             ratio=".66" alpha=".5"/>
         </s:LinearGradient>
      </s:fill>

      <s:stroke>
         <s:SolidColorStroke color="0x000000" weight="2"/>
      </s:stroke>
   </s:Rect>
</s:Skin>
```

Listing 4.11 zawiera węzeł podstawowy XML `Skin` z trzema elementami potomnymi XML definiującymi zachowanie skórki. Pierwszym z tych elementów jest element XML `Metadata`:

```
<fx:Metadata>
   [HostComponent("spark.components.Button")]
</fx:Metadata>
```

Powyższy element XML wskazuje nazwę pakietu klasy `Button`. Nazwę tę podałeś podczas dodawania pliku `ButtonSkin1.mxml` do projektu.

Drugi element potomny XML to element `states`:

```
<s:states>
   <s:State name="disabled" />
   <s:State name="down" />
   <s:State name="over" />
   <s:State name="up" />
</s:states>
```

Element XML `states` zawiera cztery elementy potomne odpowiadające stanom przycisku i trzem zdarzeniom związanym z zachowaniem myszy. Jeśli chcesz zapewnić obsługę tych stanów, możesz dodać odpowiedni kod. Trzecim elementem potomnym jest element XML `Rect` definiujący gradient liniowy, co zapewni efekt cieniowania oraz czarne obramowanie. Listing 4.12 zawiera kod dla drugiej skórki przycisku.

Listing 4.12. *Tworzenie drugiej skórki przycisku*

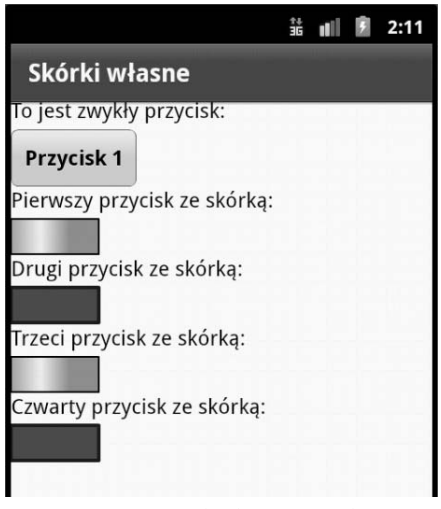
```
<s:Skin xmlns:fx="http://ns.adobe.com/mxml/2009"
        xmlns:s="library://ns.adobe.com/flex/spark" >

   <fx:Metadata>
      [HostComponent("spark.components.Button")]
   </fx:Metadata>
   <s:states>
      <s:State name="disabled" />
      <s:State name="down" />
      <s:State name="over" />
      <s:State name="up" />
   </s:states>

   <s:Path data="M 0 0 L 100 0 L 100 40 L 0 40 Z ">
      <s:fill>
         <s:SolidColor color="#FF0000" alpha="1"/>
      </s:fill>
      <s:stroke>
         <s:SolidColorStroke color="#0000FF" weight="4"/>
      </s:stroke>
   </s:Path>
</s:Skin>
```

Zapewne zauważyłeś, że jedyną różnicą pomiędzy listingami 4.12 i 4.11 jest użycie elementu XML `Path` zamiast `Rect`.

Element XML `Path` jest prosty: zawiera atrybut `data`, którego wartością jest zbiór segmentów linii definiujących prostokąt w kolorze `#FF0000` (czyli czerwonym), z obramowaniem w kolorze `#0000FF` (w niebieskim) i o szerokości linii `4`.

Jak możesz zobaczyć, Flex znacznie ułatwia definiowanie własnych skórek. Jednak w przypadku bardziej złożonych (i bardziej interesujących) skórek często określa się zachowanie po wywołaniu zdarzeń myszy (kliknięcia, zwolnienia przycisku itp.) oraz odpowiadających im zdarzeń dotykowych jako zmiany stanów. Te zmiany stanów możesz powiązać z funkcjami ActionScript (które możesz napisać samodzielnie), tak aby były wykonywane podczas tych zdarzeń, co zwiększy wizualną atrakcyjność aplikacji.

Na rysunku 4.15 pokazano standardowy przycisk Flex i cztery przyciski z własnymi skórkami.



Rysunek 4.15. *Standardowy przycisk i cztery przyciski z własnymi skórkami Spark*

Generowanie wykresów 2D w Spark

Flex 4 umożliwia eleganckie tworzenie następujących rodzajów wykresów 2D:

- wykresy warstwowe,
- wykresy kolumnowe,
- wykresy słupkowe,
- wykresy liniowe,
- wykresy kołowe,
- wykresy punktowe.

Za pomocą przedstawionych niżej przykładów pokażemy Ci, jak pisać kod aplikacji mobilnych renderujący wykresy słupkowe 2D i wykresy kołowe 2D. Dowiesz się także, jak można dodawać do wykresów animacje oraz obsługę zdarzeń myszy i zdarzeń dotykowych. Zwróć uwagę, że we Fleksie używamy pojęcia „wykres słupkowy" na określenie wykresu słupkowego narysowanego poziomo (każdy słupek jest renderowany poziomo od lewej do prawej) i pojęcia „wykres kolumnowy" na określenie wykresu, gdzie poszczególne „słupki" są narysowane pionowo — jak kolumny.

Tworzenie wykresów słupkowych 2D

Wykresy słupkowe są bardzo popularne, szczególnie w aplikacjach biznesowych. Ułatwiają dostrzeganie trendów zmian, które trudno byłoby zauważyć, analizując dane przedstawione w zwykłej tabeli. W następnym przykładzie pokażemy Ci, jak utworzyć aplikację mobilną, która odczyta dane zapisane w formacie XML z dokumentu XML, a następnie wyrenderuje te dane w postaci wykresu słupkowego 2D. Te dane służą tylko celom demonstracyjnym i na pewno zechcesz wykorzystać własne. Pamiętaj tylko, że pełny kod źródłowy przykładów prezentowanych na kartach tej książki możesz pobrać z jej strony internetowej, *www.helion.pl/ksiazki/andfzp.htm*.

Utwórz nowy projekt mobilny Flex, nadaj mu nazwę `BarChart1` — skorzystaj z szablonu aplikacji mobilnej. Do projektu dodaj nowy folder najwyższego poziomu o nazwie `chartdata`, a następnie umieść w nim nowy dokument XML o nazwie `ChartData.xml`, zawierający dane pokazane w listingu 4.13.

Listing 4.13. *Definiowanie danych XML dla wykresu*

```
<?xml version="1.0"?>
<chartdata>
   <data>
      <month>Styczeń</month>
      <revenue>1500</revenue>
   </data>
   <data>
      <month>Luty</month>
      <revenue>1400</revenue>
   </data>
[część danych pominięto]
   <data>
      <month>Listopad</month>
      <revenue>1900</revenue>
   </data>
   <data>
      <month>Grudzień</month>
      <revenue>1800</revenue>
   </data>
</chartdata>
```

Listing 4.13 zawiera element XML `chartdata` z dwunastoma elementami XML `data`; każdy z nich przechowuje dane dotyczące jednego miesiąca. Każdy element XML `data` w listingu 4.13 zawiera elementy XML `month` oraz `revenue`. Przykładowo: pierwszy element XML `data` zawiera element `revenue` o wartości `1500` oraz element `month` o wartości `Styczeń` (nie wskazujemy jednostki waluty).

Przyjrzyj się teraz listingowi 4.14, który zawiera kod renderujący wykres słupkowy z wykorzystaniem danych z dokumentu XML pokazanego w listingu 4.13.

Listing 4.14. *Tworzenie wykresu słupkowego*

```
<?xml version="1.0" encoding="utf-8"?>
<s:View xmlns:fx="http://ns.adobe.com/mxml/2009"
        xmlns:mx="library://ns.adobe.com/flex/mx"
        xmlns:s="library://ns.adobe.com/flex/spark"
        title="Wykres słupkowy">

   <!-- dane XML dla wykresu -->
   <fx:Declarations>
      <fx:Model id="chartModel" source="chartdata/ChartData.xml"/>
      <s:ArrayCollection id="chartData" source="{chartModel.data}"/>
      <mx:NumberFormatter id="nf" precision="1" rounding="nearest"/>
   </fx:Declarations>
```

```
   <fx:Style>
      @namespace s  "library://ns.adobe.com/flex/spark";
      @namespace mx "library://ns.adobe.com/flex/mx";
      mx|ColumnChart
      {
         fontSize:12;
         fontWeight:bold;
      }
   </fx:Style>

   <!-- wskazanie słupka wykresu z odpowiednimi atrybutami -->
   <mx:ColumnChart dataProvider="{chartData}"
                   height="70%" width="100%">
      <mx:horizontalAxis>
         <mx:CategoryAxis dataProvider="{chartData}"
                          categoryField="month"/>
      </mx:horizontalAxis>
      <mx:series>
         <mx:ColumnSeries xField="month" yField="revenue"/>
      </mx:series>
   </mx:ColumnChart>
</s:View>
```

Kod w listingu 4.14 definiuje położenie dokumentu XML `ChartData.xml` za pomocą elementu XML `Model`, wraz z obiektem `ArrayCollection` zawierającym dane z XML. Tworzy także prosty formater danych. Ten sam listing zawiera także element XML `Style` definiujący wartości dwóch atrybutów CSS: `fontSize` oraz `fontWeight` — odpowiednio `12` i `bold`. Służą one do formatowania tekstu w wykresie słupkowym.

Element XML `ColumnChart` definiuje wykres słupkowy wraz z potrzebnymi wartościami atrybutów `dataProvider`, `height` i `weight` — odpowiednio `chartData`, `75%` i `80%`. Wartość `chartData` jest zmienną typu `ArrayCollection`, która została zdefiniowana w elemencie XML `Declarations`. Zmienna `chartData` jest wypełniana wartościami danych wskazanych w dokumencie XML `ChartData.xml`.

Wartości atrybutów `height` i `weight` zostały określone jako wartości procentowe wymiarów ekranu, na którym jest renderowany wykres słupkowy. Możesz je dopasować odpowiednio do obszaru, jaki ma zajmować wykres (50% dla połowy szerokości lub wysokości, 25% dla jednej czwartej itp.).

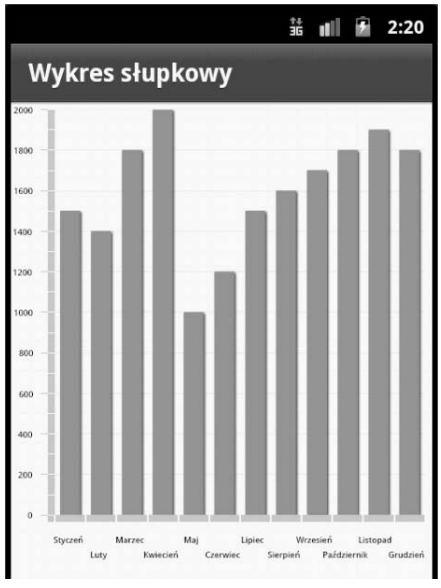
Element XML `ColumnChart` zawiera dwa istotne elementy. Pierwszy z nich to element XML `horizontalAxis` określający wartości zmiennych `month` (wskazane w pliku `CharData.xml`) dla osi poziomej. Drugi to element XML `series`, który odnosi się do wartości zmiennych `month` dla osi poziomej i jednocześnie zmiennych `revenue` dla osi pionowej wykresu słupkowego.

Na rysunku 4.16 pokazano wykres słupkowy, który powstał na podstawie danych XML z pliku *CharData.xml* z listingu 4.13.

Warto pamiętać, że na zrzucie ekranu z rysunku 4.16 brakuje pewnych istotnych informacji, takich jak waluta, w której wyrażone są wielkości dochodu, bieżący rok, nazwa i adres firmy, region (lub państwo). Jeśli będziesz chciał dodać informacje tego rodzaju, wprowadź potrzebne zmiany w kodzie z listingu 4.14. W ten sposób upewnisz się, że kod wskazuje właściwą ścieżkę dostępu do danych.

Tworzenie wykresów kołowych 2D

Wykresy kołowe są bardzo popularnym sposobem przedstawiania danych — na takim wykresie łatwo dostrzec relacje pomiędzy danymi. Utworzymy więc wykres kołowy, korzystając z danych XML z dokumentu *CharData.xml*, który pokazaliśmy w listingu 4.13. Są to więc te same dane, którymi się posłużyliśmy, rysując wykres słupkowy opisany w poprzednim punkcie. Utwórz nowy projekt mobilny Flex, nadaj mu nazwę `PieChart1` — skorzystaj z szablonu aplikacji mobilnej. Do projektu dodaj kod pokazany w listingu 4.15.



Rysunek 4.16. *Wykres słupkowy 2D*

Listing 4.15. *Tworzenie wykresu kołowego*

```
<?xml version="1.0" encoding="utf-8"?>
<s:View xmlns:fx="http://ns.adobe.com/mxml/2009"
        xmlns:mx="library://ns.adobe.com/flex/mx"
        xmlns:s="library://ns.adobe.com/flex/spark"
        title="Wykres kołowy">

   <!-- dane XML dla wykresu -->
   <fx:Declarations>
      <fx:Model id="chartModel" source="chartdata/ChartData.xml"/>
      <s:ArrayCollection id="chartData" source="{chartModel.data}"/>
      <mx:NumberFormatter id="nf" precision="1" rounding="nearest"/>
   </fx:Declarations>

   <fx:Style>
      @namespace s  "library://ns.adobe.com/flex/spark";
      @namespace mx "library://ns.adobe.com/flex/mx";
      mx|PieChart
      {
         fontSize:12;
         fontWeight:bold;
      }
   </fx:Style>

   <!-- dane dla poszczególnych części - pary nazwa:wartość -->
   <fx:Script>
      <![CDATA[
         private function getWedgeLabel (item:Object,
                        field:String,
                        index:Number,
                        percentValue:Number):String {
            return item.month+": "+item.revenue;
         }
```

```
      ]]>
   </fx:Script>

   <!-- definiowanie wykresu kołowego z odpowiednimi atrybutami -->
   <mx:PieChart dataProvider="{chartData}"
                height="100%" width="100%"
                horizontalCenter="0" verticalCenter="-150">
      <mx:series>
         <mx:PieSeries field="revenue"
                  labelFunction="getWedgeLabel"
                  labelPosition="callout"
                  explodeRadius="0.05"/>
      </mx:series>
   </mx:PieChart>
</s:View>
```

W kodzie z listingu 4.15 znalazł się element XML `Declarations` oraz element XML `Style` — takie same jak w listingu 4.14. Element XML `Script` definiuje funkcję prywatną `getWedgeLabel()`. Funkcja ta przekazuje ciąg znaków zawierający parę *nazwa*:*wartość* dla każdej części wykresu:

```
<fx:Script>
   <![CDATA[
      private function getWedgeLabel (item:Object,
                     field:String,
                     index:Number,
                     percentValue:Number):String {
         return item.month+": "+item.revenue;
      }
   ]]>
</fx:Script>
```

Element XML `PierChart` definiuje wykres kołowy wraz z atrybutami, których wartości wskazują, w jaki sposób ten wykres będzie renderowany. Na przykład: oba atrybuty `height` i `width` mają wartość `80%`, co oznacza, że wykres będzie renderowany z wysokością i szerokością równą 80% wymiarów ekranu. Możesz je dopasować odpowiednio do obszaru, jaki ma zajmować wykres (tak jak dla wykresu słupkowego).

Element XML `PieChart` zawiera także element XML `PieSeries`, w którym z kolei znajdują się cztery atrybuty umożliwiające określenie, w jaki sposób będą renderowane dane wykresu i jego poszczególne części. Atrybut `field` ma wartość `revenue`, co oznacza, że wartości danych elementu XML `revenue` będą renderowane na wykresie kołowym.

Atrybut `labelFunction` ma wartość `getWedgeLabel()`, która jest nazwą funkcji ActionScript (zdefiniowanej wcześniej w elemencie `fx:Script`). Funkcja ta tworzy etykiety dla każdego elementu wykresu.

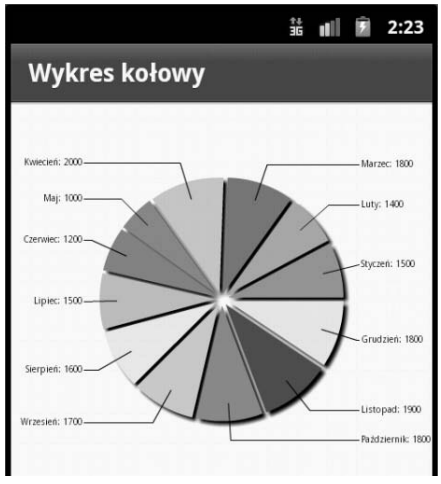
Atrybut `labelPosition` ma wartość `callout`, co oznacza, że etykieta dla każdego elementu wykresu jest renderowana na zewnątrz elementu z łamaną linią łączącą element z etykietą. Warto zauważyć, że atrybut `labelPosition` może mieć trzy inne wartości: `inside`, `outside` i `insideWithCallout`. Przetestuj je i zobacz, jaki mają wpływ na wygląd wykresu.

Na zakończenie: atrybut `explodeRadius` ma wartość `0.5`, dzięki czemu poszczególne elementy wykresu rozdzielone są odstępami, co daje efekt „eksplozji”.

Na rysunku 4.17 pokazano wykres kołowy 2D.

Wykorzystywanie FXG i Spark

W rozdziale 3. zawarliśmy bardzo krótkie wprowadzenie do FXG. W tym podrozdziale zademonstrujemy fragment kodu, dzięki któremu dowiesz się, jak przekształcić kod z listingu 4.1 (renderujący prostokąt i elipsę) do projektu Flex wykorzystującego FXG.



Rysunek 4.17. *Wykres kołowy 2D*

Utwórz nowy projekt mobilny Flex, nadaj mu nazwę `FXG1` — skorzystaj z szablonu aplikacji mobilnej, utwórz folder na najwyższym poziomie projektu i nadaj mu nazwę `components`, następnie utwórz wewnątrz tego folderu plik `RectEllipse1.fxg` z zawartością pokazaną w listingu 4.16.

Listing 4.16. *Wykorzystanie FXG do zdefiniowania elementów graficznych*

```
<?xml version="1.0" encoding="utf-8"?>
<Graphic xmlns="http://ns.adobe.com/fxg/2008" version="2">
   <Rect id="rect1" x="10" y="10" width="250" height="200">
      <fill>
         <SolidColor color="#FF0000"/>
      </fill>
      <stroke>
         <SolidColorStroke color="#FFFF00" weight="4"/>
      </stroke>
   </Rect>

   <Ellipse id="ellipse1" x="10" y="220" width="250" height="200">
      <fill>
         <SolidColor color="#0000FF"/>
      </fill>
      <stroke>
         <SolidColorStroke color="#FF0000" weight="4"/>
      </stroke>
   </Ellipse>

   <Rect id="rect2" x="10" y="460" width="250" height="100">
      <fill>
         <SolidColor color="#FFFF00"/>
      </fill>
      <stroke>
         <SolidColorStroke color="#0000FF" weight="8"/>
      </stroke>
  </Rect>

</Graphic>
```

Element XML `Graphic` zawiera dwa elementy XML, których wartości danych są takie same jak wartości elementu XML `Rect` i elementu XML `Ellipse` z listingu 4.1. Różnice między tymi elementami są następujące:

- Nazwy elementów nie posiadają przedrostków określających przestrzeń nazw.
- Nazwy elementów należą do domyślnej przestrzeni nazw.
- W atrybucie `color` znalazł się symbol `#` zamiast przedrostka `0x`.

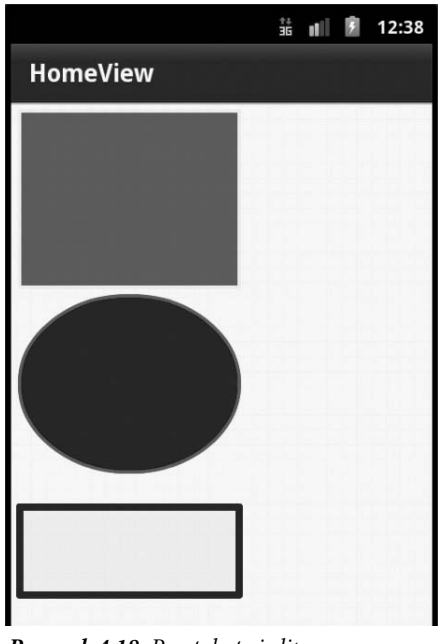
Kod w listingu 4.17 stanowi przykład, w jaki sposób można tworzyć referencje do elementu zdefiniowanego w listingu 4.16.

Listing 4.17. *Tworzenie referencji do komponentów FXG*

```
<?xml version="1.0" encoding="utf-8"?>
<s:View xmlns:fx="http://ns.adobe.com/mxml/2009"
        xmlns:mx="library://ns.adobe.com/flex/mx"
        xmlns:s="library://ns.adobe.com/flex/spark"
        xmlns:comps="components.*">

   <s:VGroup>
      <comps:RectEllipse1 id="rect1"/>
   </s:VGroup>
</s:View>
```

Kod z listingu 4.17 określa przestrzeń nazw, za pomocą której można tworzyć referencje do pliku FXG *RectEllipse1.fxg* znajdującego się w podkatalogu `components`. Element XML `VGroup` zawiera element `RectEllipse1` w przestrzeni nazw `comp`. Element ten tworzy referencję do elementu XML, którego atrybut `id` ma wartość `rect1`, zdefiniowanego w pliku *FXG RectEllipse1.fxg* (listing 4.16).

Na rysunku 4.18 pokazano efekt wykonania kodu z listingu 4.16. Wyrenderowana elipsa i dwa prostokąty są takie same jak te na rysunku 4.1.



Rysunek 4.18. *Prostokąty i elipsa*

Jak mogłeś przekonać się na podstawie tego przykładu, FXG pozwala na tworzenie modularnego kodu w projektach Flex. Ponadto warto wiedzieć, że za pomocą następujących produktów Adobe możesz eksportować projekty jako pliki FXG, które następnie możesz zaimportować do projektów Flex:

- Adobe Photoshop
- Adobe Illustrator
- Adobe Fireworks

Bardziej zaawansowane przykłady zastosowania plików FXG znajdziesz w rozdziale 9.

Program do szkicowania

W tym podrozdziale pokażemy Ci, jak utworzyć kod aplikacji mobilnej — programu do szkicowania. Wykorzystamy w nim łącznie różne techniki związane z grafiką, które opisaliśmy we wcześniejszej części tego rozdziału: obsługę zdarzeń dotykowych, rysowanie na pliku JPG i zapewnienie możliwości zapisywania szkicu na urządzeniu mobilnym jako pliku JPG.

Utwórz zatem nowy projekt mobilny Flex, nadaj mu nazwę `Sketch1` — skorzystaj z szablonu aplikacji mobilnej i dodaj do niego kod pokazany w listingu 4.18. Aby ułatwić omawianie aplikacji, zaprezentujemy kod w postaci mniejszych bloków. Pełny kod można pobrać z witryny tej książki.

Listing 4.18. *Renderowanie i zapisywanie szkiców*

```
<?xml version="1.0" encoding="utf-8"?>
<s:View xmlns:fx="http://ns.adobe.com/mxml/2009"
        xmlns:s="library://ns.adobe.com/flex/spark" title="HomeView">
   <fx:Script>
      <![CDATA[
         import flash.ui.Multitouch;
         import flash.ui.MultitouchInputMode;
         import flash.events.TouchEvent;

         import mx.graphics.ImageSnapshot;
         import mx.graphics.SolidColor;
         import mx.graphics.codec.JPEGEncoder;

         private var colors:Array = [0xFF0000, 0x00FF00, 0xFFFF00, 0x0000FF];
         private var singleTapCount:int = 0;
         private var touchMoveCount:int = 0;
         private var widthFactor:int = 0;
         private var heightFactor:int = 0;
         private var currentColor:int = 0;
         private var rectWidth:int = 20;
         private var rectHeight:int = 20;

         Multitouch.inputMode = MultitouchInputMode.TOUCH_POINT;

         function touchMove(event:TouchEvent):void {
            ++touchMoveCount;

            if (event.isPrimaryTouchPoint) {
               currentColor = colors[touchMoveCount%colors.length];
            } else {
               currentColor = colors[(touchMoveCount+2)%colors.length];
            }
```

```
            var myRect:Rect = new Rect();
            myRect.x = event.localX;
            myRect.y = event.localY;
            myRect.width = rectWidth;
            myRect.height = rectHeight;
            myRect.fill = new SolidColor(currentColor);

            var myGroup1:Group = event.target as Group;
            myGroup1.addElement(myRect);
        }
```

Kod z listingu 4.18 rozpoczyna się od elementu XML `Script`, który zawiera instrukcje importu i definicje odpowiednio nazwanych zmiennych (np. służących do śledzenia zdarzeń dotykowych) wykorzystywanych w metodach ActionScript 3.

Wartość `MultiTouch.inputMode` ustawiana jest odpowiednio dla trybu wielodotykowego, dzięki czemu renderowany jest więcej niż jeden prostokąt, kiedy użytkownik przeciągnie po ekranie więcej niż jednym palcem. Jeśli musisz odświeżyć swoje informacje na temat wielodotykowości, zajrzyj do odpowiedniego podrozdziału w rozdziale 2.

Funkcja `touchMove()` zawiera kod obsługujący zdarzenia ruchu. Funkcja na początku inkrementuje zmienną `touchMoveCount`, a następnie wykorzystuje ją jako indeks w tablicy `colors`. Dzięki temu renderowany jest ciąg prostokątów, których kolory zmieniają się odpowiednio do iterowanych kolorów w tablicy. Pozostała część kodu funkcji tworzy mały prostokąt w tym położeniu, gdzie zostało zarejestrowane zdarzenie dotykowe. Tę funkcję właściwie można nazwać sercem kodu renderującego grafikę, podczas gdy pozostałe funkcje obsługują inne zdarzenia.

Kolejnym blokiem kodu jest definicja funkcji `touchEnd()`, która jest co prawda opcjonalna, jednak stanowi dobry przykład procedury obsługi zdarzenia.

```
function touchEnd(event:TouchEvent):void {
   ++touchMoveCount;

   if (event.isPrimaryTouchPoint) {
      currentColor = colors[touchMoveCount%colors.length];
   } else {
      currentColor = colors[0];
   }

   widthFactor = (touchMoveCount%3)+1;
   heightFactor = (touchMoveCount%3)+2;

   var myRect:Rect = new Rect();
   myRect.x = event.localX;
   myRect.y = event.localY;
   myRect.width = rectWidth*widthFactor;
   myRect.height = rectHeight*heightFactor;
   myRect.fill = new SolidColor(currentColor);

   var myGroup1:Group = event.target as Group;
   myGroup1.addElement(myRect);
}
```

Kod obsługujący zdarzenia „zwolnienia” palca z wyświetlacza w funkcji `touchEnd()` inkrementuje zmienną `touchMoveCount` i służy ona jako indeks w tablicy `colors`, ale tym razem wykonywana jest prosta operacja arytmetyczna mająca na celu wyrenderowanie prostokąta innej wielkości.

```
function touchSingleTap(event:TouchEvent):void {
   var myRect:Rect = new Rect();
   myRect.x = event.localX;
   myRect.y = event.localY;
```

```
   ++singleTapCount;
   if (event.isPrimaryTouchPoint) {
      currentColor = colors[singleTapCount%colors.length];
      myRect.width = rectWidth*3;
      myRect.height = rectHeight*2;
   } else {
      currentColor = colors[(singleTapCount+1)%colors.length];
      myRect.width = rectWidth*2;
      myRect.height = rectHeight*3;
   }

   myRect.fill = new SolidColor(currentColor);

   var myGroup1:Group = event.target as Group;
   myGroup1.addElement(myRect);
}
```

Logika obsługi zdarzeń pojedynczego stuknięcia znajduje się w funkcji `touchSingleTap()`. Ta funkcja inkrementuje zmienną `touchSingleTapCount`, a następnie stosuje prostą logikę w celu ustalenia wymiarów prostokąta, renderowanego w miejscu, gdzie wystąpiło zdarzenie pojedynczego stuknięcia.

```
   function touchMoveHandlerImage(event:TouchEvent):void {
      touchMove(event);
   }

   function touchTapHandlerImage(event:TouchEvent):void {
      touchSingleTap(event);
   }

   private function saveImageToFileSystem():void {
      var jPEGEncoder:JPEGEncoder = new JPEGEncoder(500);
      var imageSnapshot:ImageSnapshot = ImageSnapshot.captureImage(imgPanel, 0, jPEGEncoder);
      var fileReference:FileReference = new FileReference();
      fileReference.save(imageSnapshot.data, "fingersketch.jpg");
      }
   ]]>
</fx:Script>
```

Funkcje `touchMoveHandlerImage()` i `touchTapHandlerImage()` (jak sugerują ich nazwy) obsługują zdarzenia ruchu i pojedynczego stuknięcia dla pliku JPG, który jest przechowywany w podkatalogu `images` aplikacji Flex. Definicje tych dwóch funkcji zawierają pojedynczą linię kodu wywołującą odpowiednio funkcje: `touchMove()` i `touchTapHandler()`, omówione we wcześniejszej części tego podrozdziału.

Funkcja `saveImageToFileSystem()` jest wywoływana za każdym razem, kiedy użytkownik kliknie przycisk *Zapisz szkic*, i zawiera kod zapisujący edytowany szkic w systemie plików urządzenia mobilnego. Zostanie także wyświetlone wyskakujące okno z domyślnym położeniem i nazwą pliku JPG. Oba te elementy użytkownik może zmienić przed zapisaniem pliku.

```
<s:Panel id="imgPanel" title="Szkicowanie palcem dla zabawy!" width="100%" height="100%" >
   <s:Button id="saveImage"
             left="150" bottom="5"
             label="Save Sketch"
             click="saveImageToFileSystem();"/>

   <s:Group name="myGroup1" width="500" height="500"
            touchMove="touchMove(event)"
            touchEnd="touchEnd(event)"
            touchTap="touchSingleTap(event)">
   <s:Ellipse id="ellipse1" x="10" y="10" width="100" height="50">
```

```
      <s:fill> <s:SolidColor color="0xFFFF00"/> </s:fill>
      <s:stroke> <s:SolidColorStroke color="red" weight="5"/> </s:stroke>
      </s:Ellipse>
      <s:Rect id="rect1" x="110" y="10" width="100" height="50">
         <s:fill> <s:SolidColor color="0xFF0000"/> </s:fill>
            <s:stroke> <s:SolidColorStroke color="blue" weight="5"/> </s:stroke>
      </s:Rect>
      <s:Ellipse id="ellipse2" x="210" y="10" width="100" height="50">
         <s:fill> <s:SolidColor color="0xFFFF00"/> </s:fill>
           <s:stroke> <s:SolidColorStroke color="red" weight="5"/> </s:stroke>
      </s:Ellipse>
      <s:Rect id="rect2" x="310" y="10" width="100" height="50">
         <s:fill> <s:SolidColor color="0xFF0000"/> </s:fill>
            <s:stroke> <s:SolidColorStroke color="blue" weight="5"/> </s:stroke>
      </s:Rect>

      <s:Path data="C100 300 200 20 300 100 S 250 200 300 250">
         <s:fill>
           <s:LinearGradient rotation="90">
              <s:GradientEntry color="#FF0000" alpha="0.8"/>
              <s:GradientEntry color="#0000FF" alpha="0.8"/>
           </s:LinearGradient>
        </s:fill>

        <s:stroke>
          <s:SolidColorStroke color="0x00FF00" weight="2"/>
        </s:stroke>
     </s:Path>

      <s:Path data="C 350 300 200 150 350 100 T 250 250 400 280">
         <s:fill>
            <s:LinearGradient rotation="90">
               <s:GradientEntry color="#FFFF00" alpha="0.5"/>
               <s:GradientEntry color="#FF0000" alpha="0.5"/>
            </s:LinearGradient>
         </s:fill>

         <s:stroke>
            <s:SolidColorStroke color="0x000000" weight="4"/>
         </s:stroke>
      </s:Path>
   </s:Group>

   <s:Image id="img" width="480" height="320" source="images/fingersketch.jpg"
            touchMove="touchMoveHandlerImage(event)"
            touchTap="touchTapHandlerImage(event)"
            horizontalCenter="-10" verticalCenter="60"/>
   </s:Panel>
</s:View>
```

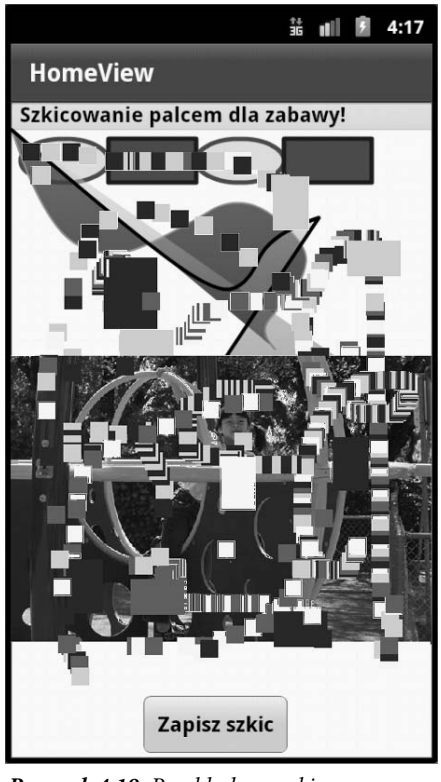
Kolejny duży blok kodu składa się z elementu XML `Panel` zawierającego element XML `Button`, który służy do zapisywania bieżącego szkicu, a następnie elementu XML `Group` definiującego procedury obsługi zdarzeń dotykowych `touchMove`, `touchEnd` i`touchTap`. Zdarzenia te omówiliśmy już wcześniej.

Element XML `Group` zawiera także definicje elementów różnych obiektów graficznych, w tym elips, prostokątów i krzywych Béziera — opisywaliśmy je we wcześniejszej części tego rozdziału.

Obiekty graficzne są oczywiście opcjonalne i mają tylko na celu pokazanie Ci, jak utworzyć program do szkicowania — program, który będzie atrakcyjny i spodoba się użytkownikom.

Element XML `Image` wskazuje plik JPG *fingersketch.jpg* w podkatalogu *images* aplikacji Flex. Element XML `Image` wskazuje funkcję `touchMoveHandlerImage()` dla dotykowych zdarzeń ruchu i funkcję `touchTapHandlerImage()` dla zdarzeń stuknięć. Wypróbuj różne wartości atrybutów `horizontalCenter` i `verticalCenter`, które zmieniają poziome i pionowe położenie układu obrazu JPG.

Na rysunku 4.19 pokazano przykładowy szkic po uruchomieniu programu na urządzeniu mobilnym.



Rysunek 4.19. *Przykładowy szkic*

Podsumowanie

Podczas lektury tego rozdziału dowiedziałeś się, jak używać komponentów Spark do renderowania różnych obiektów graficznych 2D na potrzeby aplikacji mobilnych zorientowanych graficznie. Jeśli już znasz możliwości renderowania grafiki we Fleksie, możesz łatwo poszerzyć swoją wiedzę, aby tworzyć aplikacje mobilne wykorzystujące grafikę.

Obrazy i efekty graficzne, które możesz wykorzystać, zależą od wymagań aplikacji. Oto niektóre z efektów, jakie możesz zastosować w aplikacjach mobilnych:

- Renderowanie podstawowych obiektów, takich jak prostokąty, elipsy i segmenty linii.
- Renderowanie kwadratowych i sześciennych krzywych Béziera dla aplikacji mobilnych wymagających bardziej wymyślnych, nieliniowych efektów wizualnych.
- Stosowanie gradientów liniowych, radialnych i efektów filtrów w celu uzyskania bardziej złożonych i atrakcyjnych efektów.
- Stosowanie przekształceń (przesuwanie, zmiany rozmiarów, obracanie i zniekształcanie).

- Tworzenie własnych skórek w celu zmiany standardowego wyglądu przycisków.
- Wizualizowanie danych za pomocą wykresów słupkowych i kołowych.
- Używanie animacji równoległych i sekwencyjnych w połączeniu ze zdarzeniami dotykowymi.

Skorowidz

B

C

D

E

F

G

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

W

Z

Platforma Android z każdym dniem zdobywa tysiące kolejnych użytkowników. Andy Rubin — odpowiedzialny za nią w Google — pochwalił się aktywacją 700 tysięcy nowych urządzeń z tym systemem każdego dnia. Robi wrażenie? Pomyśl, jak wykorzystać ten rynek i stworzyć aplikację, która podbije serca użytkowników. Dzięki możliwości wykorzystania technologii Flash teraz jest to jeszcze łatwiejsze!

Android Flash. Zaawansowane programowanie aplikacji mobilnych to kompletny przewodnik po budowaniu skomplikowanych, zajmujących aplikacji dla Androida. Dzięki tej książce nauczysz się programować smartfony oraz tablety z wykorzystaniem najpopularniejszych frameworków służących do tworzenia bogatych aplikacji internetowych (RIA) dla Androida — Flash i Flex. Gdy wraz z autorami przejrzysz już zestaw narzędzi programistycznych Flasha, dowiesz się, jak dodawać multimedia, animacje i efekty specjalne do aplikacji. Zrozumiesz, jak działa optymalizowanie dla różnorodnych wyświetlaczy oraz jak wykorzystać dane wejściowe pochodzące z aparatu, GPS, mikrofonu i akcelerometru. Poznasz metody tworzenia certyfikatów podpisywania kodu oraz uzyskiwania jak najlepszej wydajności aplikacji. Zapoznasz się z dużą ilością praktycznego, dobrze skomentowanego kodu i zbudujesz trzy kompletne projekty: program do rysowania, odtwarzacz muzyczny Flash oraz aplikację AIR przechowującą informacje o stanach witryn w bazie danych SQLite.

Sprawdź możliwości technologii Flash dla platformy Android.

- **Korzystaj z usług geolokalizacyjnych**
- **Sprawdź położenie telefonu za pomocą wbudowanych sensorów**
- **Wykorzystaj narzędzia przeznaczone dla technologii Flash**
- **Opublikuj aplikację w Android Market**

Twórz atrakcyjne aplikacje mobilne za pomocą Flasha, Fleksa i Androida!

ISBN 978-83-246-3920-5

Cena: 69,00 zł

9 788324 639205

helion.pl
księgarnia internetowa

Nr katalogowy: 8628

Księgarnia internetowa
http://helion.pl

Zamówienia telefoniczne:
0 801 339900
0 601 339900

Helion

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

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

Informatyka w najlepszym wydaniu