Kompendium wiedzy na temat platformy Android!

Android 3
tworzenie aplikacji

Satya Komatineni • Dave MacLean • Sayed Hashimi

Apress®

Helion

Tytuł oryginału: Pro Android 3

Tłumaczenie: Krzysztof Sawka

ISBN: 978-83-246-3586-3

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/and3ta.zip

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/and3ta
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

ROZDZIAŁ
16

Analiza animacji dwuwymiarowej

Animacja jest procesem pozwalającym wyświetlanemu na ekranie obiektowi na zmianę koloru, pozycji, rozmiaru lub orientacji w określonym przedziale czasowym. Animacje, które można wykorzystać w Androidzie, są bardzo praktyczne, zabawne, proste oraz są często wykorzystywane.

W wersji 2.3 i wcześniejszych Androida dostępne są trzy rodzaje animacji: animacja poklatkowa, która polega na rysowaniu serii klatek jedna po drugiej w regularnych odstępach czasu; animacja układu graficznego, w której są przetwarzane widoki osadzone w pojemniku, na przykład w tabeli na liście; a także animacja widoku, polegająca na animowaniu dowolnego widoku ogólnego przeznaczenia. Dwa ostatnie rodzaje należą do kategorii animacji klatek kluczowych (ang. *tweening*), która polega na interpolowaniu przez komputer klatek pośrednich pomiędzy klatkami kluczowymi.

Uwaga! W wersji 3.0 Androida zmodernizowano mechanizm animacji, gdyż wprowadzono możliwość animowania elementów interfejsu użytkownika. Niektóre z nowych koncepcji, zwłaszcza dotyczące fragmentów, zostały omówione w rozdziale 29. Niniejszy rozdział ukończyliśmy przed wydaniem wersji 3.0 Androida, więc z powodu ograniczeń czasowych zajęliśmy się w nim jedynie elementami dostępnymi do wersji 2.3 systemu. W rozdziale 29. opisaliśmy kilka funkcji animacji dostępnych od wersji 3.0 Androida.

Animację klatek kluczowych można wyjaśnić również w taki sposób, że *nie* wymaga ona rysowania klatki po klatce. Jeżeli możemy animować obiekt bez konieczności nakładania i powtarzania kolejnych klatek, to mamy do czynienia z techniką klatek kluczowych. Jeżeli na przykład dany obiekt znajduje się w punkcie A, a za 4 sekundy znajdzie się w punkcie B, możemy zmieniać jego położenie co sekundę i za każdym razem od nowa go rysować. Będziemy odnosić wrażenie, że obiekt ten porusza się z punktu A do punktu B.

Koncepcja jest taka, że znajomość początkowych i końcowych stanów animowanego obiektu pozwala grafikowi na zmianę pewnych aspektów tego obiektu w trakcie procesu animowania. Takim aspektem może być kolor, pozycja, rozmiar albo jakiś inny element. W komputerach jest to osiągane poprzez zmianę średnich wartości w regularnych odstępach czasu oraz ponowne rysowanie powierzchni.

W tym rozdziale zajmiemy się zagadnieniami animacji poklatkowej, układu graficznego oraz widoku — zaprezentujemy je z wykorzystaniem działających przykładów oraz poddamy je dogłębnej analizie.

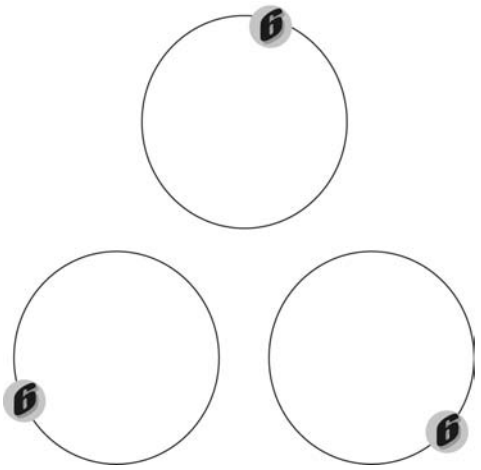
Uwaga! Na końcu rozdziału zamieściliśmy adres URL, z którego możemy pobrać projekty utworzone na potrzeby tego rozdziału i zaimportować je do środowiska Eclipse.

Animacja poklatkowa

Animacja jest prostym procesem polegającym na wyświetlaniu serii obrazów następujących po sobie w krótkich odstępach czasu, w wyniku czego powstaje wrażenie poruszającego lub zmieniającego się obiektu. W taki sposób działają projektory filmowe. Pokażemy przykładowy projekt, w którym zaprojektujemy obraz i zapiszemy go w formie serii oddzielnych klatek, różniących się od siebie w niewielkim stopniu. Następnie umieścimy ten zbiór obrazów w przykładowym kodzie umożliwiającym uruchomienie animacji.

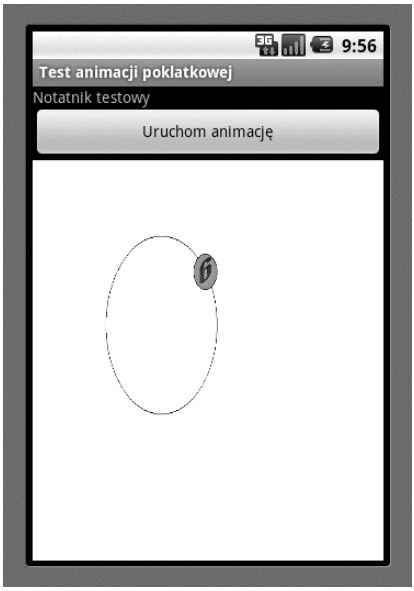
Zaplanowanie animacji poklatkowej

Przed rozpoczęciem pisania kodu należy najpierw zaplanować sekwencję animacji za pomocą rozrysowania jej na papierze. Przykład takiego planowania został zilustrowany na rysunku 16.1, przedstawiającym zbiór równowymiarowych okręgów, na których obwodach zostały umieszczone w różnych pozycjach kolorowe kółka. Można stworzyć zbiór takich rysunków przedstawiających okrąg oraz kółko umieszczone w różnych miejscach na obwodzie tego okręgu. Po zachowaniu siedmiu lub ośmiu takich klatek utworzymy animację symulującą ruch kółka po okręgu.



Rysunek 16.1. Etap projektowania animacji

Określmy sobie podstawowy człon nazwy takiego rysunku, na przykład `colored-ball`, a następnie zachowajmy utworzone rysunki w podkatalogu */res/drawable*, żeby w przyszłości można było uzyskać do nich dostęp za pomocą identyfikatorów zasobów. Nazwa każdego pliku powinna zostać utworzona za pomocą wzoru `colored-ballN`, gdzie `N` jest numerem porządkowym klatki. Po utworzeniu animacji powinna ona wyglądać tak jak na rysunku 16.2.



Rysunek 16.2. Środowisko testowe animacji poklatkowej

Główny obszar aktywności jest wykorzystywany przez widok animacji. Wstawiliśmy przycisk uruchamiania i zatrzymywania animacji w celu obserwacji jej zachowania. W górnej części ekranu umieściliśmy również notatnik testowy, w którym można zapisywać wszelkie ważne zdarzenia podczas eksperymentowania z programem. Zobaczmy, w jaki sposób można utworzyć układ graficzny takiej aktywności.

Utworzenie aktywności

Rozpocznijmy od utworzenia prostego pliku XML układu graficznego w podkatalogu */res/layout* (listing 16.1).

Listing 16.1. Plik XML układu graficznego do przykładu animacji poklatkowej

```
<?xml version="1.0" encoding="utf-8"?>
<!-- nazwa pliku: /res/layout/frame_animations_layout.xml -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:orientation="vertical"
   android:layout_width="fill_parent"
   android:layout_height="fill_parent"
   >
<TextView android:id="@+id/textViewId1"
   android:layout_width="fill_parent"
   android:layout_height="wrap_content"
```

```
   android:text="Notatnik testowy"
/>
<Button
  android:id="@+id/startFAButtonId"
   android:layout_width="fill_parent"
   android:layout_height="wrap_content"
   android:text="Uruchom animację"
/>
<ImageView
      android:id="@+id/animationImage"
      android:layout_width="fill_parent"
     android:layout_height="wrap_content"
      />
</LinearLayout>
```

Pierwszą kontrolką jest kontrolka notatnika testowego stanowiąca prosty widok `TextView`. Następnie dodajemy przycisk uruchamiania i zatrzymywania animacji. Ostatni jest widok `ImageView`, w którym będzie odtwarzana animacja. Po skonstruowaniu układu graficznego należy utworzyć aktywność wczytującą ten widok (listing 16.2).

Listing 16.2. Aktywność wczytująca widok ImageView

```
public class FrameAnimationActivity extends Activity
{
    @Override
   public void onCreate(Bundle savedInstanceState)
   {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.frame_animations_layout);
   }
}
```

Taką aktywność będzie można uruchomić za pomocą dowolnego elementu menu, dostępnego w bieżącej aplikacji, poprzez wykonanie następującego kodu:

```
Intent intent = new Intent(inActivity,FrameAnimationActivity.class);
inActivity.startActivity(intent);
```

W tym momencie aktywność powinna wyglądać tak jak na rysunku 16.3.

Dodawanie animacji do aktywności

Po utworzeniu aktywności oraz układu graficznego pokażemy, w jaki sposób można do nich dodawać animację. W Androidzie animacja poklatkowa jest tworzona poprzez klasę `AnimationDrawable` z pakietu graficznego. Można stwierdzić po nazwie, że będzie się ona zachowywać jak każdy inny obiekt rysowany, mogący stanowić tło dla dowolnego widoku (na przykład mapy bitowe tła są reprezentowane jako elementy `Drawable`). Klasa `AnimationDrawable` poza tym, że należy do kategorii `Drawable`, może pobierać listę innych obiektów tego typu (na przykład obrazów) i wyświetlać je w określonych interwałach czasowych. W rzeczywistości klasa ta jest cienką osłoną wokół obsługi animacji zapewnianej przez bazową klasę `Drawable`.

Rysunek 16.3. Aktywność animacji poklatkowej

Wskazówka Klasa `Drawable` uruchamia animację w ten sposób, że pojemnik lub widok wywołuje klasę `Runnable`, która w istocie przerysowuje obiekt `Drawable` za pomocą innego zestawu parametrów. Zwróćmy uwagę, że nie musimy znać takich szczegółów wewnętrznej implementacji, żeby korzystać z klasy `AnimationDrawable`. Jednak w przypadku bardziej złożonych wymagań można zajrzeć do kodu źródłowego klasy `AnimationDrawable`, aby znaleźć wskazówki do napisania własnych protokołów animacji.

Żeby móc skorzystać z klasy `AnimationDrawable`, należy najpierw umieścić zestaw zasobów typu `Drawable` (na przykład zbiór obrazów) w podkatalogu */res/drawable*. Gwoli ścisłości, umieścimy tam osiem podobnych, lecz nie identycznych obrazów omówionych w punkcie „Zaplanowanie animacji poklatkowej”. Następnie utworzymy plik XML definiujący listę klatek (listing 16.3). Także ten plik musi zostać umieszczony w podkatalogu */res/drawable*.

Listing 16.3. Plik XML definiujący listę animowanych klatek

```
<animation-list xmlns:android="http://schemas.android.com/apk/res/android"
      android:oneshot="false">
   <item android:drawable="@drawable/colored_ball1" android:duration="50" />
   <item android:drawable="@drawable/colored_ball2" android:duration="50" />
   <item android:drawable="@drawable/colored_ball3" android:duration="50" />
   <item android:drawable="@drawable/colored_ball4" android:duration="50" />
   <item android:drawable="@drawable/colored_ball5" android:duration="50" />
   <item android:drawable="@drawable/colored_ball6" android:duration="50" />
   <item android:drawable="@drawable/colored_ball7" android:duration="50" />
   <item android:drawable="@drawable/colored_ball8" android:duration="50" />
</animation-list>
```

Uwaga! Podczas przygotowywania listy obrazów musimy pamiętać o pewnych ograniczeniach klasy `AnimationDrawable`. Przed rozpoczęciem animacji klasa ta wczytuje wszystkie obrazy do pamięci. Podczas testowania przykładowego projektu na emulatorze wyposażonym w wersję systemu 2.3 liczba klatek większa od 6 przekraczała pojemność pamięci przydzielonej dla aplikacji. W zależności od środowiska testowego być może będziemy musieli ograniczyć liczbę klatek. Aby rozwiązać ten problem, musimy bezpośrednio skorzystać z funkcji animacyjnych klasy `Drawable` i wprowadzić własny mechanizm. Niestety, klasa `Drawable` nie została szczegółowo omówiona w tym wydaniu książki. Proponujemy wizytę na stronie *www.androidbook.com*, gdyż planujemy zaktualizować jej zawartość w niedługim czasie.

Każda klatka wskazuje na jeden z rysunków określony przez jego identyfikator zasobu. Znacznik `animation-list` zostaje przekształcony do obiektu `AnimationDrawable`, reprezentującego zbiór obrazów. Musimy teraz umieścić klasę `Drawable` jako zasób tła dla widoku `ImageView`. Zakładając, że nazwaliśmy ten plik *frame_animation.xml* i umieściliśmy go w podkatalogu */res/drawable*, możemy zastosować poniższy kod do ustanowienia klasy `AnimationDrawable` jako tła widoku `ImageView`:

```
view.setBackGroundResource(Resource.drawable.frame_animation);
```

Dzięki tej linii kodu Android rozpoznaje identyfikator zasobu `Resource.drawable.frame_animation` jako zasób XML i zgodnie z nim tworzy odpowiedni obiekt Java `AnimationDrawable`, zanim ustawi go jako tło. Gdy już będziemy mieli tło, możemy uzyskać dostęp do tego obiektu `AnimationDrawable` poprzez wprowadzenie instrukcji `get` do widoku `View` w następujący sposób:

```
Object backgroundObject = view.getBackground();
AnimationDrawable ad = (AnimationDrawable)backgroundObject;
```

Po umieszczeniu obiektu klasy `AnimationDrawable` możemy wprowadzić metody `start()` i `stop()` służące do uruchamiania i zatrzymywania animacji. Poniżej zaprezentowaliśmy dwie inne istotne metody tego obiektu:

```
setOneShot();
addFrame(drawable, duration);
```

Metoda `setOneShot()` odtwarza animację jeden raz i potem ją zatrzymuje. Metoda `addFrame()` dodaje nową klatkę za pomocą obiektu `Drawable` i konfiguruje czas jej wyświetlania. Działanie tej metody przypomina funkcję znacznika XML `android:drawable`.

Teraz musimy złożyć wszystkie fragmenty kodu w całość, aby otrzymać środowisko testowe animacji poklatkowej (listing 16.4).

Listing 16.4. Pełny kod środowiska testowego animacji poklatkowej

```
public class FrameAnimationActivity extends Activity {
   @Override
   public void onCreate(Bundle savedInstanceState)
   {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.frame_animations_layout);
      this.setupButton();
   }
   private void setupButton()
   {
```

```
      Button b = (Button)this.findViewById(R.id.startFAButtonId);
      b.setOnClickListener(
         new Button.OnClickListener(){
            public void onClick(View v)
            {
               parentButtonClicked(v);
            }
         });
   }
   private void parentButtonClicked(View v)
   {
      animate();
   }
   private void animate()
   {
         ImageView imgView = (ImageView)findViewById(R.id.animationImage);
         imgView.setVisibility(ImageView.VISIBLE);
         imgView.setBackgroundResource(R.drawable.frame_animation);

         AnimationDrawable frameAnimation =
            (AnimationDrawable) imgView.getBackground();

         if (frameAnimation.isRunning())
         {
            frameAnimation.stop();
         }
         else
         {
            frameAnimation.stop();
            frameAnimation.start();
         }
   }
}//eof-class
```

Metoda `animate()` lokalizuje widok `ImageView` w bieżącej aktywności i przypisuje mu tło `AnimationDrawable` rozpoznane przez zasób `R.drawable.frame_animation`. Następnie kod odczytuje ten obiekt i przeprowadza proces animowania. Przycisk uruchamiania i zatrzymywania jest skonfigurowany w ten sposób, że jego naciśnięcie w trakcie odtwarzania animacji zatrzyma ją; jeżeli animacja jest zatrzymana, jego naciśnięcie spowoduje jej uruchomienie.

Zauważmy, że jeśli przypiszemy parametrowi listy animacji `OneShot` wartość `true`, animacja wykona tylko jeden cykl. Jednak nie można dokładnie przewidzieć, kiedy się to stanie. Chociaż animacja zostaje zakończona po wyświetleniu ostatniego obrazu, nie otrzymamy żadnego informującego o tym komunikatu. Z tego powodu nie istnieje żaden bezpośredni sposób wywołania kolejnej czynności w odpowiedzi na zakończenie animacji.

Pomimo tej niedogodności można uzyskać wspaniałe efekty wizualne poprzez wyświetlanie po kolei serii obrazów w prostym procesie animacji poklatkowej.

Animacja układu graficznego

Podobnie jak w przypadku animacji poklatkowej, animacja układu graficznego jest bardzo prosta. Jak sama nazwa wskazuje, animacja tego typu jest przeznaczona dla pewnego rodzaju widoków,

ułożonych w określony sposób. Stosowana jest ona na przykład w przypadku widoków `ListView` oraz `GridView`, które są dwiema powszechnie implementowanymi kontrolkami w systemie Android. W szczególności animacja układu graficznego jest używana do dodawania efektów graficznych, zmieniających sposób wyświetlania elementów umieszczonych w wymienionych widokach. Tak naprawdę może być ona stosowana wobec wszystkich kontrolek wywodzących się z klasy `ViewGroup`.

W przeciwieństwie do animacji poklatkowej, animacja układu graficznego nie jest generowana poprzez powtarzanie klatek. Zamiast tego są zmieniane w czasie różne właściwości widoku. Każdy widok w Androidzie zawiera macierz transformacji, która odwzorowuje widok wyświetlony na ekranie. Poprzez zmianę takiej macierzy na różne sposoby można przeprowadzić procesy skalowania, obracania i przemieszczania (translacji) tego widoku. Na przykład poprzez zmianę przezroczystości widoku w skali od 0 do 1 otrzymujemy tak zwaną animację typu `alfa`.

Podstawowe typy animacji klatek kluczowych

Poniżej prezentujemy nieco bardziej szczegółowo podstawowe rodzaje animacji klatek kluczowych (ang. *tweening*):

- **Animacja skali.** Ten typ animacji umożliwia powiększanie lub zmniejszanie widoku w osi *x* oraz w osi *y*. Można także określić punkt zwrotny, wokół którego będzie odtwarzana animacja.
- **Animacja rotacyjna.** Dzięki niej można obracać widok wokół punktu zwrotnego o określony kąt.
- **Animacja translacyjna.** Służy ona do przesuwania widoku wzdłuż osi *x* lub *y*.
- **Animacja typu alfa.** Służy do zmieniania przezroczystości widoku.

Animacje tego typu są definiowane w postaci plików XML umieszczonych w podkatalogu */res/anim*. Listing 16.5 prezentuje krótki przykład, pomagający zrozumieć, w jaki sposób te animacje są definiowane.

Listing 16.5. Animacja skali zdefiniowana w pliku XML, umieszczonym w podkatalogu /res/anim/scale.xml

```
<set xmlns:android="http://schemas.android.com/apk/res/android"
android:interpolator="@android:anim/accelerate_interpolator">
   <scale
      android:fromXScale="1"
      android:toXScale="1"
      android:fromYScale="0.1"
      android:toYScale="1.0"
      android:duration="500"
      android:pivotX="50%"
      android:pivotY="50%"
      android:startOffset="100" />
</set>
```

Wszystkie wartości parametrów w tym pliku animacji zostają określane „od – do”, ponieważ musimy określić wartości początkowe i końcowe animacji.

Każda z animacji dopuszcza również możliwość korzystania z interpolatorów czasu w postaci argumentów. Interpolatory zostaną omówione na końcu podrozdziału związanego z animacją układu graficznego, teraz jednak wystarczy wiedzieć, że są one odpowiedzialne za szybkość zmian argumentów w trakcie przetwarzania animacji.

Po utworzeniu tego pliku deklarowania animacji możemy powiązać animację z układem graficznym, dzięki czemu elementy składowe układu graficznego będą animowane.

Uwaga! W tym miejscu warto wspomnieć, że każda z tych animacji jest reprezentowana jako klasa Java w pakiecie `android.view.animation`. Dokumentacja każdej z tych klas nie tylko opisuje jej metody języka Java, lecz również dopuszczalne argumenty XML dla każdego typu animacji.

Skoro już naszkicowaliśmy zarys rodzajów animacji układu graficznego wystarczający do ich chociażby podstawowego zrozumienia, zajmijmy się projektowaniem przykładu.

Zaplanowanie środowiska testowego animacji układu graficznego

Za pomocą prostego zestawu `ListView` w aktywności można przetestować wszystkie omówione przez nas koncepcje animacji układu graficznego. Po utworzeniu widoku `ListView` można do niego dołączyć animację, co spowoduje jej przetworzenie wobec każdego elementu tego widoku.

Załóżmy, że chcemy utworzyć animację skali, która powiększa widok od zera do oryginalnego rozmiaru w osi *y*. Możemy to sobie wyobrazić wizualnie jako linijkę tekstu, która najpierw przypomina poziomą linię, a następnie zostaje powiększona do właściwego rozmiaru czcionki.

Można tę animację dołączyć do widoku `ListView`. Kiedy to zrobimy, każdy element tej listy będzie wyświetlany za pomocą tej animacji.

Możemy dodać kilka parametrów, które urozmaicą podstawową animację, na przykład animowanie listy od góry do dołu lub odwrotnie. Parametry te są definiowane w klasie pośredniej, zachowującej się jak mediator pomiędzy konkretnym plikiem XML animacji a widokiem listy.

Istnieje możliwość zdefiniowania zarówno animacji, jak i mediatora w pliku XML umieszczonym w podkatalogu */res/anim*. Gdy już utworzymy taki pośredniczący plik XML, możemy go wykorzystać w postaci danych wejściowych dla widoku `ListView` w jego własnym pliku definicji XML. Gdy ta podstawowa konfiguracja będzie już działać, będziemy zmieniać animacje, żeby przekonać się, w jaki sposób wpływają one na wyświetlanie elementów widoku `ListView`.

Zanim rozpoczniemy ćwiczenie, przyjrzyjmy się, jak widok `ListView` będzie wyglądał po zakończeniu animacji (rysunek 16.4).

Utworzenie aktywności oraz widoku ListView

Rozpoczniemy od utworzenia układu graficznego XML dla widoku `ListView` przedstawionego na rysunku 16.4, dzięki czemu możliwe będzie wczytanie tego układu graficznego w prostej aktywności. Na listingu 16.6 został umieszczony taki nieskomplikowany układ graficzny z zaimplementowanym widokiem `ListView`. Taki plik należy umieścić w podkatalogu */res/layout*. Zakładając, że nazwa pliku brzmi *list_layout.xml*, kompletna ścieżka do niego będzie wyglądała następująco: */res/layout/list_layout.xml*.

Rysunek 16.4. Animowana lista ListView

Listing 16.6. Plik XML układu graficznego definiujący widok ListView

```
<?xml version="1.0" encoding="utf-8"?>
<!-- nazwa pliku: /res/layout/list_layout.xml -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:orientation="vertical"
   android:layout_width="fill_parent"
   android:layout_height="fill_parent"
   >

   <ListView
      android:id="@+id/list_view_id"
      android:layout_width="fill_parent"
      android:layout_height="fill_parent"
      />
</LinearLayout>
```

Listing 16.6 przedstawia prosty menedżer `LinearLayout` z umieszczonym wewnątrz niego prostym widokiem `ListView`. Powinniśmy jednak skorzystać z okazji i wyjaśnić pewną rzecz dotyczącą definicji widoku `ListView`, która jest dość marginalnie powiązana z treścią rozdziału. Jeżeli Czytelnik będzie pracował na aplikacji Notepad lub innych przykładowych programach, zauważy zapewne, że identyfikator widoku `ListView` jest przeważnie określany jako `@android:id/list`. Zgodnie z informacjami z rozdziału 3. odniesienie `@android:id/list` wskazuje na identyfikator predefiniowany w przestrzeni nazw `android`. Pytanie brzmi: kiedy należy stosować odniesienie `android:id`, a kiedy nasz własny identyfikator, na przykład `@+id/list_view_id`?

Identyfikatora `@android:id/list` używamy jedynie w przypadku, gdy aktywnością jest `List`
`↪Activity`. W przypadku tej aktywności zakłada się, że widok `ListView`, określony przez ten predefiniowany identyfikator, jest dostępny do wczytania. W tym wypadku używamy raczej

aktywności ogólnego przeznaczenia, a nie `ListActivity`, i musimy własnoręcznie zapełnić w jawny sposób widok `ListView`. W związku z tym nie ma żadnych ograniczeń co do rodzaju identyfikatora, który ma reprezentować tę listę. Jednak można także wykorzystać odniesienie `@android:id/list`, ponieważ nie stwarza to żadnego konfliktu z powodu braku aktywności `ListActivity`.

To taka mała dygresja, warto jednak o niej pamiętać podczas tworzenia własnych widoków `ListView` poza aktywnością `ListActivity`. Gdy już posiadamy układ graficzny wymagany dla aktywności, możemy napisać kod odpowiedzialny za wczytanie tego pliku układu graficznego, dzięki czemu zostanie wygenerowany interfejs użytkownika (listing 16.7).

Listing 16.7. Kod aktywności odpowiedzialnej za animację układu graficznego

```
public class LayoutAnimationActivity extends Activity
{
   @Override
   public void onCreate(Bundle savedInstanceState)
   {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.list_layout);
      setupListView();
   }
   private void setupListView()
   {
         String[] listItems = new String[] {
            " Element 1", " Element 2", " Element 3",
            " Element 4", " Element 5", " Element 6",
         };

         ArrayAdapter listItemAdapter =
            new ArrayAdapter(this
               ,android.R.layout.simple_list_item_1
               ,listItems);
         ListView lv = (ListView)this.findViewById(R.id.list_view_id);
         lv.setAdapter(listItemAdapter);
   }
}
```

Niektóre fragmenty kodu widocznego na listingu 16.7 są oczywiste, ale inne nie. Pierwsza część kodu w zwykły sposób wczytuje widok na podstawie wygenerowanego identyfikatora układu graficznego `R.layout.list_layout`. Naszym zadaniem jest zapełnienie widoku `ListView` z tego układu graficznego sześcioma elementami. Te elementy tekstowe zostały wczytane do tablicy. Musimy ustanowić adapter danych wobec widoku `ListView`, żeby te elementy mogły zostać wyświetlone.

Aby utworzyć wymagany adapter, musimy określić, w jaki sposób każdy z elementów będzie wstawiany podczas wyświetlania listy na ekranie. Układ graficzny określamy za pomocą predefiniowanego układu, znajdującego się w strukturze Androida. W naszym przykładzie układ graficzny wyznaczono następująco:

```
android.R.layout.simple_list_item_1
```

Innymi dostępnymi układami graficznymi widoku dla tych elementów są:

```
simple_list_item_2
simple_list_item_checked
simple_list_item_multiple_choice
simple_list_item_single_choice
```

Można zajrzeć do dokumentacji Androida, aby się dowiedzieć, jak te układy graficzne wyglądają i jak się zachowują. Teraz możemy wywołać tę aktywność za pomocą dowolnego przycisku menu w aplikacji po wstawieniu następującego kodu:

```
Intent intent = new Intent(inActivity,LayoutAnimationActivity.class);
inActivity.startActivity(intent);
```

Jednak — podobnie jak w przypadku wywołań innych aktywności — musimy zarejestrować aktywność `LayoutAnimationActivity` w pliku *AndroidManifest.xml*, jeżeli powyższe wywołanie aktywności ma zadziałać. Poniżej umieściliśmy potrzebny do tego kod:

```
<activity android:name=".LayoutAnimationActivity"
android:label="Testowa aktywność widoku animacji"/>
```

Animowanie widoku ListView

Po przygotowaniu środowiska testowego (listingi 16.6 i 16.7) Czytelnik dowie się, w jaki sposób wstawiać animację skali do widoku `ListView`. Spójrzmy, jak animacja ta zostaje zdefiniowana w pliku XML (listing 16.8).

Listing 16.8. Definiowanie animacji skali w pliku XML

```
<set xmlns:android="http://schemas.android.com/apk/res/android"
android:interpolator="@android:anim/accelerate_interpolator">
   <scale
      android:fromXScale="1"
      android:toXScale="1"
      android:fromYScale="0.1"
      android:toYScale="1.0"
      android:duration="500"
      android:pivotX="50%"
      android:pivotY="50%"
      android:startOffset="100" />
</set>
```

Jak już wcześniej wspomnieliśmy, pliki definiujące animacje są przechowywane w podkatalogu */res/anim*.

Przetłumaczmy te atrybuty XML na język polski.

Wagi `from` i `to` są wskaźnikami początku oraz zakończenia procesu powiększania. W naszym wypadku powiększanie rozpoczyna się od wartości `1` i takie pozostaje dla osi *x*. Oznacza to, że element nie będzie powiększany ani zmniejszany w tej osi.

Jednak w przypadku osi *y* powiększanie rozpoczyna się od wartości `0.1` i dąży do `1.0`. Innymi słowy, na początku animacji rozmiar obiektu stanowi jedną dziesiątą jego naturalnego rozmiaru, do którego dąży w czasie trwania animacji.

Cała operacja skalowania zajmie 500 milisekund.

Środek działania znajduje się w połowie drogi obydwu osi (`50%`).

Wartość `startOffset` odnosi się do czasu (wyrażonego w milisekundach), po którym animacja zostanie uruchomiona.

Węzeł nadrzędny animacji skali wskazuje na zestaw animacji, który dopuszcza wprowadzenie większej liczby animacji. Omówimy również tego rodzaju przykład. Na razie jednak mamy do dyspozycji tylko jedną animację w zestawie.

Nazwijmy ten plik *scale.xml* i umieśćmy go w podkatalogu */res/anim*. Nie jesteśmy na razie gotowi, żeby wstawić ten plik XML animacji jako argument w widoku `ListView`; widok ten wymaga jeszcze jednego pliku XML, który będzie zachowywał się jak pośrednik pomiędzy widokiem a zestawem animacji. Kod pliku XML, w którym zaimplementowane jest takie powiązanie, został pokazany na listingu 16.9.

Listing 16.9. Definicja dla pliku XML stanowiącego kontroler układu graficznego

```
<layoutAnimation xmlns:android="http://schemas.android.com/apk/res/android"
   android:delay="30%"
   android:animationOrder="reverse"
   android:animation="@anim/scale" />
```

Również ten plik należy umieścić w podkatalogu */res/anim*. W naszym przykładzie zakładamy, że plik nosi nazwę `list_layout_controller`. Po przyjrzeniu się definicji pliku pośredniczącego zrozumiemy, dlaczego jest on niezbędny.

W pliku tym zostaje określone, że animacja tej listy powinna przebiegać w odwróconym porządku oraz że animacja każdego elementu będzie opóźniona o 30% względem całkowitego czasu trwania animacji. Znajduje się tu również odniesienie do pliku animacji — *scale.xml*. Zauważmy również, że w kodzie jest użyte odniesienie do tego pliku *@anim/scale* zamiast jego nazwy.

Gdy już posiadamy wymagane pliki XML z danymi wejściowymi, pokażemy, w jaki sposób należy zaktualizować definicję XML widoku `ListView`, żeby obejmowała ona animację XML jako argument. Najpierw przejrzyjmy dotychczas utworzone pliki XML:

```
// pojedyncza animacja skali
/res/anim/scale.xml

// plik pośredniczący
/res/anim/list_layout_controller.xml

// plik układu graficznego widoku aktywności
/res/layout/list_layout.xml
```

Gdy te pliki są gotowe, musimy zmodyfikować plik XML układu graficznego *list_layout.xml* w taki sposób, żeby widok `ListView` wskazywał plik *list_layout_controller.xml* (listing 16.10).

Listing 16.10. Zaktualizowany kod pliku List_Layout.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:orientation="vertical"
   android:layout_width="fill_parent"
   android:layout_height="fill_parent"
   >
   <ListView
```

```
      android:id="@+id/list_view_id"
      android:persistentDrawingCache="animation|scrolling"
      android:layout_width="fill_parent"
      android:layout_height="fill_parent"
      android:layoutAnimation="@anim/list_layout_controller" />
      />
</LinearLayout>
```

Zmienione wiersze zostały wyróżnione pogrubioną czcionką. Kluczowym znacznikiem jest `android:layoutAnimation`, który wskazuje pośredniczący plik XML definiujący kontroler układu graficznego za pomocą znacznika `layoutAnimation` (listing 16.9). Z kolei znacznik `layoutAnimation` odnosi się do animacji, w naszym wypadku animacji skali zdefiniowanej w pliku *scale.xml*.

Android zaleca także wstawienie znacznika `persistentDrawingCache`, który optymalizuje animację i przesuwanie. Więcej informacji na jego temat można znaleźć w dokumentacji środowiska Android SDK.

Po zaktualizowaniu pliku *list_layout.xml* zgodnie z listingiem 16.10 wtyczka ADT środowiska Eclipse automatycznie przekompiluje pakiet, uwzględniając wprowadzone zmiany. Gdybyśmy teraz uruchomili aplikację, zobaczylibyśmy, że animacja skali jest przeprowadzana na każdym elemencie. Zdefiniowaliśmy czas trwania animacji na 500 milisekund, zatem ujrzymy wyraźnie zmianę skali podczas rysowania obiektu.

Możemy już eksperymentować z innymi rodzajami animacji. Sprawdzimy teraz animację typu alfa. W tym celu utworzymy plik */res/anim/alpha.xml* i umieścimy w nim treść listingu 16.11.

Listing 16.11. Plik alpha.xml do testowania animacji typu alfa

```
<alpha xmlns:android="http://schemas.android.com/apk/res/android"
       android:interpolator="@android:anim/accelerate_interpolator"
       android:fromAlpha="0.0" android:toAlpha="1.0" android:duration="1000" />
```

Animacja typu alfa jest odpowiedzialna za kontrolę zmiany nasycenia kolorów. W tym przykładzie w ciągu 1000 milisekund (1 sekundy) kolor z przezroczystego staje się w pełni nasycony. Dobrze jest ustawić czas trwania animacji na co najmniej 1 sekundę, w przeciwnym wypadku zmiana nasycenia będzie trudna do zaobserwowania.

W przypadku zmiany animacji pojedynczego elementu musimy zmienić również treść pliku pośredniczącego (listing 16.9), żeby wskazywała plik z nową animacją. Poniżej pokazaliśmy sposób wskazywania z animacji skali na animację typu alfa:

```
<layoutAnimation xmlns:android="http://schemas.android.com/apk/res/android"
   android:delay="30%"
   android:animationOrder="reverse"
   android:animation="@anim/alpha" />
```

Zmieniony wiersz w tym kodzie wyróżniono pogrubioną czcionką. Spróbujmy teraz stworzyć animację łączącą zmianę położenia ze zmianą gradientu nasycenia koloru. Listing 16.12 przedstawia przykładowy kod takiej animacji.

Listing 16.12. Połączenie animacji translacyjnej z animacją typu alfa w zestawie animacji

```
<set xmlns:android="http://schemas.android.com/apk/res/android"
android:interpolator="@android:anim/accelerate_interpolator">
   <translate android:fromYDelta="-100%" android:toYDelta="0"
android:duration="500" />
   <alpha android:fromAlpha="0.0" android:toAlpha="1.0"
android:duration="500" />
</set>
```

Zwróćmy uwagę, w jaki sposób określiliśmy dwie animacje w zestawie animacji. Animacja translacyjna będzie przesuwała tekst z góry na dół w wydzielonym dla niego obszarze wyświetlania. Animacja typu alfa będzie powodować zmianę gradientu nasycenia koloru od przezroczystego do całkowicie nasyconego podczas przesuwania tekstu w dół. Wartość 500 czasu trwania animacji pozwoli użytkownikowi obserwować w wygodny sposób zmianę. Oczywiście znowu będzie trzeba zmienić plik pośredniczący `layoutAnimation`, tak żeby znalazło się w nim odniesienie do nowego pliku. Zakładając, że nazwą pliku zawierającego połączone animacje jest */res/anim/translate-alpha.xml*, plik *layoutAnimation* będzie wyglądał następująco:

```
<layoutAnimation xmlns:android="http://schemas.android.com/apk/res/android"
   android:delay="30%"
   android:animationOrder="reverse"
   android:animation="@anim/translate-alpha" />
```

Zobaczmy, w jaki sposób można używać animacji rotacyjnej (listing 16.13).

Listing 16.13. Plik XML animacji rotacyjnej

```
<rotate xmlns:android="http://schemas.android.com/apk/res/android"
    android:interpolator="@android:anim/accelerate_interpolator"
    android:fromDegrees="0.0"
   android:toDegrees="360"
   android:pivotX="50%"
   android:pivotY="50%"
   android:duration="500" />
```

Kod z listingu 16.13 spowoduje wykonanie jednego pełnego obrotu przez każdy element tekstowy wokół środka tego elementu. Czas trwania 500 milisekund całkowicie wystarczy, żeby obserwator dostrzegł animację. Podobnie jak w poprzednich przypadkach, tak i teraz muszą zostać zmodyfikowane pliki XML kontrolera animacji oraz układu graficznego `ListView`, a aplikacja musi zostać ponownie uruchomiona, żeby animacja zadziałała.

Omówiliśmy już podstawowe pojęcia dotyczące animacji układu graficznego, począwszy od prostego pliku animacji, a skończywszy na powiązaniu go poprzez plik pośredniczący `layout`
`↪Animation` z widokiem `ListView`. Ta wiedza wystarczy, żeby ujrzeć animowane efekty. Musimy omówić jednak jeszcze jedno pojęcie dotyczące animacji układu graficznego — interpolatory.

Stosowanie interpolatorów

Interpolatory określają, w jaki sposób dana właściwość, na przykład gradient koloru, zmienia się względem czasu. Czy będzie się ona zmieniała w sposób liniowy, czy w sposób wykładniczy? Czy rozpocznie się szybko, lecz będzie zwalniała z biegiem czasu? Zastanówmy się nad przykładem animacji typu alfa z listingu 16.11:

```
<alpha xmlns:android="http://schemas.android.com/apk/res/android"
       android:interpolator="@android:anim/accelerate_interpolator"
       android:fromAlpha="0.0" android:toAlpha="1.0" android:duration="1000" />
```

Animacja rozpoznaje zastosowany interpolator — w tym przypadku `accelerate_interpolator`. Istnieje odpowiedni obiekt Java, służący do definiowania tego interpolatora. Poza tym zwróćmy uwagę, że określiliśmy ten interpolator jako odniesienie do zasobów. Oznacza to, że musi istnieć plik odpowiadający identyfikatorowi *anim/accelerate_interpolator*, w którym opisany jest ten obiekt języka Java oraz jego dodatkowe parametry. Tak jest w istocie. Przyjrzyjmy się definicji pliku XML, do którego odniesieniem jest identyfikator `@android:anim/accelerate_interpolator`:

```
<accelerateInterpolator
xmlns:android="http://schemas.android.com/apk/res/android"
factor="1" />
```

Plik ten można odnaleźć w następującym podkatalogu pakietu Android:

/res/anim/accelerate_interpolator.xml

Znacznik XML `accelerateInterpolator` odpowiada następującemu obiektowi środowiska Java:

```
android.view.animation.AccelerateInterpolator
```

W dokumentacji języka Java dotyczącej tej klasy można zobaczyć, jakie znaczniki XML są dla niej dostępne. Zadaniem tego interpolatora jest zapewnienie współczynnika powielania danego przedziału czasowego w oparciu o krzywą hiperboliczną. Widać to w kodzie źródłowym interpolatora:

```
public float getInterpolation(float input)
{
   if (mFactor == 1.0f)
   {
      return (float)(input * input);
   }
   else
   {
      return (float)Math.pow(input, 2 * mFactor);
   }
}
```

Każdy interpolator w inny sposób implementuje metodę `getInterpolation`. W naszym przypadku, jeśli interpolator zostanie skonfigurowany tak, że współczynnik będzie wynosił `1.0`, zostanie zwrócony kwadrat tego współczynnika. W przeciwnym razie zostanie zwrócona potęga danych wejściowych, które będą nadal skalowane przez ten współczynnik. Jeżeli zatem wartość współczynnika będzie wynosiła `1.5`, zamiast funkcji kwadratowej ujrzymy funkcję sześcienną.

Poniżej wypisaliśmy listę obsługiwanych interpolatorów:

```
AccelerateDecelerateInterpolator
AccelerateInterpolator
CycleInterpolator
DecelerateInterpolator
LinearInterpolator
AnticipateInterpolator
AnticipateOvershootInterpolator
BounceInterpolator
OvershootInterpolator
```

Żeby zaprezentować potencjalną elastyczność interpolatorów, przyjrzyjmy się pokrótce obiektowi `BounceInterpolator`, powodującemu „podskakiwanie" elementu (to znaczy jego naprzemienny ruch w górę i w dół) do samego końca poniższej animacji:

```
public class BounceInterpolator implements Interpolator {
   private static float bounce(float t) {
      return t * t * 8.0f;
   }

   public float getInterpolation(float t) {
      t *= 1.1226f;
      if (t < 0.3535f) return bounce(t);
      else if (t < 0.7408f) return bounce(t - 0.54719f) + 0.7f;
      else if (t < 0.9644f) return bounce(t - 0.8526f) + 0.9f;
      else return bounce(t - 1.0435f) + 0.95f;
   }
}
```

Zachowanie tych interpolatorów zostało omówione pod poniższym adresem:

http://developer.android.com/reference/android/view/animation/package-summary.html

W dokumentacji języka Java wymienione są również znaczniki XML, pozwalające na kontrolowanie każdej z tych klas. Jednak z dokumentacji trudno wywnioskować przeznaczenie każdego typu interpolatora. Najlepiej jest samemu wypróbować wszystkie interpolatory i sprawdzić skutki ich działania. Pod poniższym adresem można również przejrzeć kod źródłowy:

http://android.git.kernel.org/?p=platform%2Fframeworks%2Fbase.git&a=search&h=HEAD&st=grep&s=BounceInterpolator

Na tym zakończymy wywody poświęcone animacji układu graficznego. Przejdziemy teraz do trzeciej części animowania, poświęconej programowaniu animacji widoku.

Animacja widoku

Skoro zapoznaliśmy się już z animacją poklatkową oraz animacją układu graficznego, możemy zająć się animacją widoku — najbardziej skomplikowanym rodzajem animacji. Stosowana jest w niej technika animowania dowolnego widoku poprzez kontrolowanie macierzy transformacji, służącej do wyświetlania widoku.

Animacja widoku

Widok wyświetlany przez Androida przechodzi przez macierz transformacji. W aplikacjach graficznych macierze transformacji służą do przekształcenia w jakiś sposób widoku. Proces ten polega na przetłumaczeniu wejściowego zestawu współrzędnych pikseli i kombinacji kolorów na nowy zestaw. Po przeprowadzeniu transformacji ujrzymy obraz zmieniony pod względem rozmiaru, pozycji, orientacji lub koloru.

Te przekształcenia można przeprowadzić za pomocą aparatu matematycznego, mnożąc w określony sposób wejściowy zestaw współrzędnych przez wartości macierzy transformacji, dzięki czemu powstanie nowy zestaw współrzędnych. Poprzez zmianę macierzy transformacji wpływamy na wygląd widoku.

Macierz, która *nie* zmienia widoku podczas tego mnożenia, nazywana jest macierzą jednostkową. Transformację przeważnie rozpoczynamy od macierzy jednostkowej i kolejno wprowadzamy serie transformacji rozmiaru, pozycji i orientacji. Następnie za pomocą macierzy końcowej rysujemy widok.

Android odsłania taką macierz transformacji widoku poprzez umożliwienie zarejestrowania obiektu animacji wobec tego widoku. Obiekt animacji będzie posiadał procedurę wywołania, dzięki której uzyska dostęp do tej macierzy i w określony sposób zmieni jej wartości, co pociągnie za sobą zmianę wyświetlania widoku. Zajmiemy się teraz tym procesem.

Rozpocznijmy tworzenie przykładowego projektu od zaplanowania animacji widoku. Na początek zapełnimy aktywność kilkoma elementami w widoku `ListView`, podobnie jak miało to miejsce w podrozdziale „Animacja układu graficznego". Następnie w górnej części ekranu umieścimy przycisk powodujący uruchomienie animacji `ListView` (rysunek 16.5). Widoczne są zarówno lista elementów, jak i przycisk, żadna animacja nie została jednak jeszcze uruchomiona. Do tego będzie służył utworzony przycisk.

Rysunek 16.5. Aktywność animacji widoku

Po kliknięciu przycisku *Uruchom animację* powinien się pojawić mały widok pośrodku ekranu, który następnie stopniowo będzie się powiększał aż do wypełnienia zarezerwowanej dla niego przestrzeni. Zaprezentujemy kod, który nam to umożliwi. Na listingu 16.14 został pokazany kod pliku XML układu graficznego, nadający się do zastosowania w aktywności.

Listing 16.14. Plik XML układu graficznego dla aktywności animacji widoku

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Ten plik jest umieszczony w /res/layout/list_layout.xml -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:orientation="vertical"
   android:layout_width="fill_parent"
   android:layout_height="fill_parent"
   >
```

```
<Button
   android:id="@+id/btn_animate"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Uruchom animację"
/>
<ListView
     android:id="@+id/list_view_id"
     android:persistentDrawingCache="animation|scrolling"
     android:layout_width="fill_parent"
     android:layout_height="fill_parent"
 />
</LinearLayout>
```

Pogrubiona czcionka ma zwrócić uwagę Czytelnika na lokalizację oraz nazwę pliku. Ten układ graficzny składa się z dwóch części: pierwsza z nich to przycisk *btn_animate*, służący do uruchomienia animacji widoku; drugą jest widok `ListView`, w naszym przypadku nazwany *list_view_id*.

Skoro mamy już układ graficzny dla aktywności, możemy utworzyć samą aktywność, żeby wyświetlić widok i skonfigurować przycisk *Uruchom animację* (listing 16.15).

Listing 16.15. Kod dla aktywności animacji widoku przed rozpoczęciem animacji

```
public class ViewAnimationActivity extends Activity {

   @Override
   public void onCreate(Bundle savedInstanceState)
   {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.list_layout);
      setupListView();
      this.setupButton();
   }
   private void setupListView()
   {
      String[] listItems = new String[] {
         "Element 1", "Element 2", "Element 3",
         "Element 4", "Element 5", "Element 6",
      };
      ArrayAdapter listItemAdapter =
         new ArrayAdapter(this
               ,android.R.layout.simple_list_item_1
               ,listItems);
      ListView lv = (ListView)this.findViewById(R.id.list_view_id);
      lv.setAdapter(listItemAdapter);
   }
   private void setupButton()
   {
      Button b = (Button)this.findViewById(R.id.btn_animate);
      b.setOnClickListener(
         new Button.OnClickListener(){
            public void onClick(View v)
            {
```

```
                //animateListView();
            }
        });
    }
}
```

Kod przedstawiony dla aktywności animacji widoku z listingu 16.15 bardzo przypomina kod aktywności animacji układu graficznego z listingu 16.7. W podobny sposób wczytaliśmy widok i wstawiliśmy sześć elementów tekstowych do widoku `ListView`. Skonfigurowaliśmy przycisk w taki sposób, żeby wywoływał metodę `animateListView()` po kliknięciu. Na razie jednak oznaczymy ten fragment komunikatem, dopóki nasz przykład nie zadziała.

Aktywność możemy wywołać tuż po jej zarejestrowaniu w pliku *AndroidManifest.xml*:

```
<activity android:name=".ViewAnimationActivity"
          android:label="Aktywność testowa animacji widoku">
```

Po przeprowadzeniu procesu rejestracji możemy wywołać aktywność animacji widoku za pomocą dowolnego przycisku menu w aplikacji, korzystając z poniższego fragmentu kodu:

```
Intent intent = new Intent(this, ViewAnimationActivity.class);
startActivity(intent);
```

Po uruchomieniu programu pojawi się ekran pokazany na rysunku 16.5.

Dodawanie animacji

W tym ćwiczeniu naszym celem jest dodanie animacji do widoku `ListView`, widocznego na rysunku 16.5. W tym celu potrzebujemy klasy wywodzącej się z pakietu `android.view.animation.Animation`. Następnie musimy przesłonić metodę `applyTransformation`, aby można było zmodyfikować macierz transformacji. Nazwijmy tę klasę `ViewAnimation`. Po jej utworzeniu możemy przeprowadzić w klasie `ListView` następującą czynność:

```
ListView lv = (ListView)this.findViewById(R.id.list_view_id);
lv.startAnimation(new ViewAnimation());
```

Pójdźmy dalej. Przyjrzyjmy się kodowi źródłowemu klasy `ViewAnimation` i zastanówmy się, jaki rodzaj animacji chcemy otrzymać (listing 16.16).

Listing 16.16. Kod źródłowy klasy ViewAnimation

```
public class ViewAnimation extends Animation
{

   @Override
   public void initialize(int width, int height, int parentWidth,
                                                 int parentHeight)
   {
      super.initialize(width, height, parentWidth, parentHeight);
      setDuration(2500);
      setFillAfter(true);
      setInterpolator(new LinearInterpolator());
   }
   @Override
   protected void applyTransformation(float interpolatedTime, Transformation t)
```

```
    {
        final Matrix matrix = t.getMatrix();
        matrix.setScale(interpolatedTime, interpolatedTime);
    }
}
```

Metoda zwrotna `initialize` informuje nas o wymiarach widoku. W niej są również inicjalizowane wszelkie parametry animacji. W naszym przykładzie skonfigurowaliśmy czas trwania na `2500` milisekund (2,5 sekundy). Sprawimy także, że wynik końcowy animacji pozostanie niezmieniony po jej zakończeniu, a to za sprawą przypisania parametrowi `FillAfter` wartości `true`. W dodatku określiliśmy, że nasz interpolator jest liniowy, co oznacza, że animacja zmienia się stopniowo od początku do końca. Wszystkie wymienione właściwości pochodzą z bazowej klasy `android.view.animation.Animation`.

Część zasadnicza animacji jest przeprowadzana w metodzie `applyTransformation`. Szkielet Androida będzie ją bez przerwy wywoływał w celu symulowania animacji. Za każdym wywołaniem tej metody zmienia się wartość parametru `interpolatedTime`. Zmienia się ona w zakresie od `0` do `1` w zależności od tego, w jakim momencie się znajdujemy podczas 2,5-sekundowego cyklu animacji, ustawionego na etapie jej inicjalizacji. Kiedy wartość parametru `interpolatedTime` wynosi `1`, znajdujemy się na końcu animacji.

Naszym kolejnym zadaniem jest zmiana macierzy transformacji, dostępnej poprzez obiekt transformacji `t`, umieszczony w metodzie `applyTransformation`. Najpierw należy uzyskać dostęp do macierzy i zmienić jej wartości. Po narysowaniu nowego widoku zadziała również zmodyfikowana macierz. W dokumentacji interfejsów API dotyczącej klasy `android.graphics.Matrix` można znaleźć opis wielu metod dostępnych w obiekcie `Matrix`:

http://developer.android.com/reference/android/graphics/Matrix.html

W kodzie z listingu 16.16 zmianą macierzy transformacji zajmuje się poniższy wiersz:

```
matrix.setScale(interpolatedTime, interpolatedTime);
```

Metoda `setScale` zawiera dwa parametry — są to współczynniki skali w osiach *x* oraz *y*. Ponieważ wartości parametru `interpolatedTime` mieszczą się w zakresie od `0` do `1`, można go zastosować bezpośrednio w postaci współczynnika skali.

Zatem na początku animacji współczynnik ten wynosi `0` w obydwu kierunkach. W połowie przebiegu animacji osie *x* oraz *y* będą miały wartość `0,5`. Po zakończeniu animacji widok będzie miał pełny rozmiar, ponieważ obydwa współczynniki skali będą miały wartości równe `1`. W wyniku tego widok `ListView` jest na początku animacji niewielki i powiększa się do standardowego rozmiaru.

Na listingu 16.17 został zaprezentowany kompletny kod źródłowy aktywności `ViewAnimationActivity` zawierającej animację.

Listing 16.17. Kod źródłowy aktywności animacji widoku wraz z animacją

```
public class ViewAnimationActivity extends Activity {

    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.list_layout);
        setupListView();
```

```
      this.setupButton();
   }
   private void setupListView()
   {
      String[] listItems = new String[] {
            "Element 1", "Element 2", "Element 3",
            "Element 4", "Element 5", "Element 6",
      };

      ArrayAdapter listItemAdapter =
         new ArrayAdapter(this
               ,android.R.layout.simple_list_item_1
               ,listItems);
      ListView lv = (ListView)this.findViewById(R.id.list_view_id);
      lv.setAdapter(listItemAdapter);
   }
   private void setupButton()
   {
      Button b = (Button)this.findViewById(R.id.btn_animate);
      b.setOnClickListener(
         new Button.OnClickListener(){
            public void onClick(View v)
            {
               animateListView();
            }
         });
   }
   private void animateListView()
   {
      ListView lv = (ListView)this.findViewById(R.id.list_view_id);
      lv.startAnimation(new ViewAnimation());
   }
}
```

Po uruchomieniu kodu z listingu 16.17 Czytelnik zobaczy coś dziwnego. Widok `ListView`, zamiast równomiernie powiększać się od środka ekranu, rozrasta się od lewego górnego rogu. Wynika to z faktu, że operacje macierzy transformacji mają swój początek właśnie w lewym górnym rogu ekranu. Chcąc uzyskać zamierzony efekt, musimy najpierw przesunąć cały widok w taki sposób, żeby jego środek pokrywał się ze środkiem animacji (w lewym górnym rogu). Następnie wprowadzamy macierz i z powrotem przenosimy widok na właściwe miejsce.

Na listingu 16.18 wstawiliśmy przerobiony kod z listingu 16.16 i zaznaczyliśmy najistotniejsze elementy.

Listing 16.18. Animacja widoku wykorzystująca metody preTranslate i postTranslate

```
public class ViewAnimation extends Animation {
   float centerX, centerY;
   public ViewAnimation3(){}

   @Override
   public void initialize(int width, int height, int parentWidth, int parentHeight) {
      super.initialize(width, height, parentWidth, parentHeight);
      centerX = width/2.0f;
      centerY = height/2.0f;
```

```
      setDuration(2500);
      setFillAfter(true);
      setInterpolator(new LinearInterpolator());
   }
   @Override
   protected void applyTransformation(float interpolatedTime, Transformation t) {
      final Matrix matrix = t.getMatrix();
      matrix.setScale(interpolatedTime, interpolatedTime);
      matrix.preTranslate(-centerX, -centerY);
      matrix.postTranslate(centerX, centerY);
   }
}
```

Metody `preTranslate` oraz `postTranslate` konfigurują macierz przed operacją skalowania oraz po tej operacji. Jest to proces równoważny utworzeniu zespołu trzech macierzy transformacji. Następujący kod:

```
matrix.setScale(interpolatedTime, interpolatedTime);
matrix.preTranslate(-centerX, -centerY);
matrix.postTranslate(centerX, centerY);
```

jest równoważny instrukcjom:

```
przejdź do innego środka
skaluj
przejdź do oryginalnego środka
```

Taki wzorzec metod `pre` i `post` jest stosowany bardzo często. Podobne wyniki można osiągnąć za pomocą innych metod klasy `Matrix`, ta technika jest jednak najpopularniejsza — a do tego jest zwięzła. Pozostałe techniki również zostaną omówione pod koniec rozdziału.

Co ważniejsze, klasa `Matrix` umożliwia nie tylko skalowanie widoku, lecz również przenoszenie go za pomocą metod `translate` oraz zmianę jego orientacji za pomocą metod `rotate`. Można sprawdzić te metody i przekonać się, jak wyglądają ich efekty. W rzeczywistości wszystkie animacje omówione w podrozdziale „Animacja układu graficznego" są implementowane wewnętrznie za pomocą metod klasy `Matrix`.

Zastosowanie klasy Camera do symulowania głębi w obrazie dwuwymiarowym

Pakiet graficzny w Androidzie zawiera jeszcze jedną klasę związaną z animacją — a dokładniej z transformacją — klasę `Camera`. Można ją wykorzystać do symulowania głębi poprzez rzutowanie obrazu dwuwymiarowego, poruszającego się w przestrzeni trójwymiarowej po płaszczyźnie. Możemy na przykład wysłać nasz widok `ListView` o 10 pikseli w głąb ekranu po osi *z* i obrócić ją o 30 stopni wokół osi *y*. Na listingu 16.19 podajemy przykład modyfikowania macierzy za pomocą klasy `Camera`:

Listing 16.19. Zastosowanie klasy Camera

```
...
public class ViewAnimation extends Animation {
   float centerX, centerY;
   Camera camera = new Camera();
   public ViewAnimation1(float cx, float cy){
```

```
        centerX = cx;
        centerY = cy;
    }
    @Override
    public void initialize(int width, int height, int parentWidth, int parentHeight) {
        super.initialize(width, height, parentWidth, parentHeight);
        setDuration(2500);
        setFillAfter(true);
        setInterpolator(new LinearInterpolator());
    }
    @Override
    protected void applyTransformation(float interpolatedTime, Transformation t) {
        applyTransformationNew(interpolatedTime,t);
    }
    protected void applyTransformationNew(float interpolatedTime, Transformation t)
    {
        final Matrix matrix = t.getMatrix();
        camera.save();
        camera.translate(0.0f, 0.0f, (1300 - 1300.0f * interpolatedTime));
        camera.rotateY(360 * interpolatedTime);
        camera.getMatrix(matrix);

        matrix.preTranslate(-centerX, -centerY);
        matrix.postTranslate(centerX, centerY);
        camera.restore();
    }
}
```

Animacja widoku `ListView` przebiega tu w następujący sposób: najpierw jest on umieszczony w odległości 1300 pikseli od ekranu po osi *z*, a następnie wraca do płaszczyzny, w której oś *z* przyjmuje wartość 0. W międzyczasie zostaje on również obrócony od 0 do 360 stopni wokół osi *y*. Zobaczmy, w jaki sposób w kodzie jest zdefiniowane to zachowanie, opisane w poniższej metodzie:

```
camera.translate(0.0f, 0.0f, (1300 - 1300.0f * interpolatedTime));
```

Metoda ta powoduje, że obiekt `camera` przesuwa się w taki sposób, iż przy wartości `0` parametru `interpolatedTime` (początek animacji) wartość *z* będzie wynosiła `1300`. Podczas trwania animacji wartość *z* będzie systematycznie malała aż do samego końca, gdy wartość parametru `interpolatedTime` wyniesie `1`, a tym samym wartość parametru *z* będzie równa `0`.

Metoda `camera.rotateY(360 * interpolatedTime)` wykorzystuje możliwość obracania bryły w trójwymiarze wokół wybranej osi przez obiekt `camera`. Na początku animacji jej wartość wynosi `0`. Na końcu animacji przybierze wartość `360`.

Metoda `camera.getMatrix(matrix)` pobiera operacje dotychczas wykonane na obiekcie `Camera` i narzuca je przekazanej macierzy transformacji. W tym momencie klasa `matrix` posiada wszystkie translacje potrzebne do uzyskania końcowego efektu, zapewnione przez klasę `Camera`. Teraz klasa `Camera` schodzi z widoku (niezamierzona gra słów), ponieważ w macierzy zostały zaimplementowane wszystkie niezbędne operacje. Wykonujemy teraz operacje `pre` i `post` w celu przesunięcia środka widoku i sprowadzenia go z powrotem. Na koniec przywracamy obiekt `Camera` do pierwotnego, uprzednio zachowanego stanu.

Po wstawieniu kodu do naszego przykładu zobaczymy kontrolkę `ListView` zbliżającą się ze środka widoku w stronę użytkownika, przy okazji wirującą, dokładnie tak jak zaplanowaliśmy.

Część naszej analizy wiążącej się z animacją widoku dotyczyła sposobu animowania dowolnego widoku poprzez rozszerzenie klasy `Animation` i zastosowanie jej wobec tego widoku. Poza modyfikowaniem matryc (bezpośrednio i za pomocą klasy `Camera`) klasa `Animation` umożliwia też wykrywanie poszczególnych etapów animacji. Właśnie tym się teraz zajmiemy.

Analiza interfejsu AnimationListener

Android wykorzystuje interfejs nasłuchujący `AnimationListener` do monitorowania zdarzeń animacji (listing 16.20). Możemy nasłuchiwać tych zdarzeń poprzez zaimplementowanie interfejsu `AnimationListener` i skonfigurowanie tej implementacji wobec klasy `Animation`.

Listing 16.20. Implementacja interfejsu AnimationListener

```
public class ViewAnimationListener
implements Animation.AnimationListener {

   private ViewAnimationListener(){}

   public void onAnimationStart(Animation animation)
   {
      Log.d("Przykładowa animacja", "onAnimationStart");
   }
   public void onAnimationEnd(Animation animation)
   {
      Log.d("Przykładowa animacja", "onAnimationEnd");
   }
   public void onAnimationRepeat(Animation animation)
   {
      Log.d("Przykładowa animacja", "onAnimationRepeat");
   }
}
```

Klasa `ViewAnimationListener` służy jedynie do tworzenia dzienników komunikatów. Możemy zaktualizować metodę `animateListView` w naszym przykładzie animacji widoku (listing 16.17), żeby dołączyć obiekt nasłuchujący animację:

```
private void animateListView()
{
   ListView lv = (ListView)this.findViewById(R.id.list_view_id);
   ViewAnimation animation = new ViewAnimation();
   animation.setAnimationListener(new ViewAnimationListener()):
   lv.startAnimation(animation);
}
```

Kilka uwag na temat macierzy transformacji

Jak pokazaliśmy w tym rozdziale, macierze stanowią podstawę przekształcania widoków i przetwarzania animacji. Teraz omówimy w skrócie niektóre kluczowe metody klasy `Matrix`. Poniżej zostały wymienione podstawowe operacje na macierzach:

```
matrix.reset();
matrix.setScale();
matrix.setTranslate()
matrix.setRotate();
matrix.setSkew();
```

Pierwsza operacja przekształca macierz do postaci macierzy jednostkowej, która po zastosowaniu nie wprowadza zmian w widoku. Operacja `setScale` jest odpowiedzialna za zmianę rozmiaru, `setTranslate` powoduje przesunięcie pozycji obiektu imitujące ruch, a `setRotate` służy do zmiany orientacji. Operacja `setSkew` pozwala na wykrzywienie widoku.

Można powiązać ze sobą macierze lub je wspólnie powielać, aby utworzyć efekt złożony z wielu transformacji. Rozpatrzmy następujący przykład, w którym `m1`, `m2` oraz `m3` są macierzami jednostkowymi:

```
m1.setScale();
m2.setTranlate()
m3.concat(m1,m2)
```

Transformacja widoku przez macierz `m1` i następująca po niej transformacja widoku przez macierz `m2` są tożsame z transformacją tego samego widoku przez macierz `m3`. Zwróćmy uwagę, że metody typu `set` zastępują poprzednie transformacje, natomiast `m3.concat(m1,m2)` nie jest tym samym co `m3.concat(m2,m1)`.

Pokazaliśmy już sposób postępowania podczas stosowania metod `preTranslate` oraz `post`
`↪Translate` wobec zmiany macierzy transformacji. W rzeczywistości metody `pre` i `post` nie są przeznaczone wyłącznie dla operacji `translate`, lecz tego typu odmiany są dostępne dla każdego rodzaju metod transformacji typu `set`. Ostatecznie metoda `preTranslate`, taka jak `m1.preTranslate(m2)`, jest równoważna operacji:

```
m1.concat(m2,m1)
```

W analogiczny sposób metoda `m1.postTranslate(m2)` jest tożsama operacji:

```
m1.concat(m1,m2)
```

Po rozszerzeniu ekwiwalentem poniższego kodu:

```
matrix.setScale(interpolatedTime, interpolatedTime);
matrix.preTranslate(-centerX, -centerY);
matrix.postTranslate(centerX, centerY);
```

jest:

```
Matrix matrixPreTranslate = new Matrix();
matrixPreTranslate.setTranslate(-centerX, -centerY);

Matrix matrixPostTranslate = new Matrix();
matrixPostTranslate.setTranslate(cetnerX, centerY);

matrix.concat(matrixPreTranslate,matrix);
matrix.postTranslate(matrix,matrixPostTranslate);
```

Odnośniki

Poniżej prezentujemy przydatne odnośniki do materiałów, dzięki którym jeszcze lepiej zrozumiemy koncepcje zawarte w tym rozdziale:

- *http://developer.android.com/reference/android/view/animation/package-summary.html* — znajdziemy tu różnorodne interfejsy związane z animacją, w tym również interpolatory.

- *http://developer.android.com/guide/topics/resources/animation-resource.html* — omówienie znaczników XML stosowanych w różnych odmianach animacji.
- *ftp://ftp.helion.pl/przyklady/and3ta.zip* — znajdziemy tu projekt do pobrania, przygotowany specjalnie do tego rozdziału. Jest to plik umieszczony w katalogu *ProAndroid3_R16_Animacje*.

Podsumowanie

W tym rozdziale zaprezentowaliśmy ciekawy sposób uatrakcyjnienia interfejsu użytkownika poprzez zastosowanie animacji. Omówiliśmy wszystkie podstawowe typy animacji obsługiwane w Androidzie: animację poklatkową, animację układu graficznego oraz animację widoku. Opisaliśmy także dodatkowe pojęcia dotyczące animacji, między innymi interpolatory i macierze transformacji.

Skoro Czytelnik poznał już podstawy, proponujemy przejrzeć przykładowe interfejsy API, udostępnione w zestawie Android SDK, aby przeanalizować pliki XML definiujące różne typy animacji. Poruszymy jeszcze temat animacji w rozdziale 20., poświęconym rysowaniu i animowaniu za pomocą technologii OpenGL. Natomiast w rozdziale 29. możemy się zapoznać z ogólnym omówieniem animacji opartej na właściwościach, stosowanej wraz z fragmentami.

Skorowidz

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

Ś

T

U

V

W

X

Z

Ż

System operacyjny Android podbił rynek smartfonów, a teraz z dnia na dzień rośnie jego popularność wśród użytkowników tabletów. Sympatię zdobył sobie dzięki niezwykle przyjaznemu interfejsowi, szerokim możliwościom dostosowania do własnych potrzeb oraz niewyobrażalnej liczbie dostępnych aplikacji. W Android Market opublikowano ich już ponad 500 tysięcy! Wśród nich każdy znajdzie coś dla siebie niezależnie od tego, czy ma to być gra, czy unikalna aplikacja użytkowa.

Oparty na powszechnie znanym języku Java, posiadający obszerną, bogatą w przykłady dokumentację wprost zachęca do przygotowania ciekawej aplikacji, na której można zarobić konkretne pieniądze. Dzięki tej rewelacyjnej książce poświęconej Androidowi w wersji 3 w mig opanujesz jego tajniki. Na samym początku poznasz historię Androida oraz dowiesz się, jak przygotować środowisko pracy. Następnie zaznajomisz się ze strukturą aplikacji, sposobem korzystania z zasobów oraz dostawców treści. W dalszych rozdziałach nauczysz się budować estetyczny, funkcjonalny i atrakcyjny interfejs użytkownika, zapamiętywać preferencje użytkowników oraz korzystać z usług HTTP. Książka ta jest kompletnym i unikalnym kompendium wiedzy na temat Androida. Powinna znaleźć się na półce każdego dewelopera tworzącego oprogramowanie dla platformy Android!

- **Przygotowanie środowiska pracy**
- **Pisanie aplikacji opartych na środowisku Java**
- **Projektowanie i budowanie interfejsu użytkownika**
- **Wysyłanie i odbieranie komunikatów**
- **Tworzenie animacji dwuwymiarowej**
- **Korzystanie z usług geolokalizacyjnych**
- **Przetwarzanie tekstu na mowę**
- **Publikacja aplikacji w Android Market**

Zacznij tworzyć oprogramowanie dla najbardziej obiecującej platformy dla urządzeń mobilnych!

ISBN 978-83-246-3586-3

9 788324 635863

Cena: 149,00 zł