Twórz stabilne, wydajne i szybkie aplikacje!

Optymalizacja wydajności aplikacji na Android

Hervé Guihot

Helion
Apress®

Tytuł oryginału: Pro Android Apps Performance Optimization

Tłumaczenie: Tomasz Walczak

ISBN: 978-83-246-5555-7

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

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

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

Printed in Poland.

Spis treści

ROZDZIAŁ 8

Grafika

Często trzeba poświęcić dużo czasu na projektowanie wyglądu aplikacji. Niezależnie od tego, czy jest nią klient pocztowy oparty na standardowych kontrolkach Androida, czy gra wykorzystująca standard OpenGL ES, wygląd programu jest jedną z pierwszych rzeczy, na jakie użytkownicy zwracają uwagę w trakcie przeglądania sklepów Android Market lub Amazon Appstore.

Jednak aplikacja, która wprawdzie świetnie wygląda, ale jest wolno wyświetlana lub aktualizowana, prawdopodobnie nie stanie się hitem. Podobnie gra z wysokiej jakości renderingiem, ale niską liczbą klatek, pewnie nie spodoba się użytkownikom. Sukces aplikacji zależy od jej ocen, dlatego ważne jest, aby nie skupiać się tylko na pierwszym wrażeniu.

W tym rozdziale poznasz podstawowe metody optymalizowania układów za pomocą różnych technik i narzędzi. Dowiesz się też, jak optymalizować rendering z wykorzystaniem standardu OpenGL ES, aby zwiększyć liczbę klatek lub obniżyć zużycie energii.

Optymalizowanie układów

Prawdopodobnie znasz już układy w formacie XML i metodę `setContentView()`. Na listingu 8.1 pokazano typowy sposób korzystania z tej metody. Choć zdaniem wielu programistów definiowanie układów jest proste (zwłaszcza za pomocą graficznego interfejsu układów, dostępnego w środowisku Eclipse), łatwo można utworzyć wysoce nieoptymalny układ. W tym podrozdziale poznasz kilka podstawowych sposobów na upraszczanie układów i przyspieszanie rozwijania układów do klas.

Listing 8.1. Typowy sposób wywoływania metody setContentView()

```
public class MyActivity extends Activity {
    private static final String TAG = "MyActivity";

    /** Wywoływana, gdy aktywność jest tworzona po raz pierwszy */
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // Wywołanie metody setContentView() w celu rozwinięcia
        // i skonfigurowania układu (zdefiniowanego w pliku main.xml)
        setContentView(R.layout.main);
        ...
```

```
    }

    ...
}
```

Choć wywołanie jest tu bardzo proste, uruchomienie metody `setContentView()` prowadzi do wykonania na zapleczu kilku operacji. Oto one:

- Android wczytuje dane zasobów aplikacji (z pliku APK, zapisanego albo w pamięci wewnętrznej, albo na karcie SD).
- Ma miejsce przetwarzanie danych zasobów i rozwijanie układu do klasy.
- Rozwinięty układ staje się widokiem najwyższego poziomu w aktywności.

Czas wykonywania tej metody zależy od złożoności układu. Im więcej jest danych zasobów, tym dłużej trwa przetwarzanie. Im więcej egzemplarzy klas trzeba utworzyć, tym dłuższe jest rozwijanie układu.

W trakcie tworzenia androidowego projektu w środowisku Eclipse w pliku *main.xml* generowany jest układ domyślny, przedstawiony na listingu 8.2. Tekst widoku `TextView` jest zdefiniowany w pliku *strings.xml*.

Listing 8.2. Układ domyślny

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

    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/hello" />

</LinearLayout>
```

W tablecie Samsung Galaxy Tab 10.1 wywołanie metody `setContentView()` z listingu 8.1 dla przedstawionego tu układu zajmuje około 17 milisekund. To szybko, jednak warto zauważyć, że układ jest niezwykle prosty. W typowych aplikacjach na Androida zwykle układy są inne. Tu trzeba utworzyć egzemplarze tylko dwóch klas (`LinearLayout` i `TextView`), a w pliku XML ustawione są nieliczne właściwości.

■ **Uwaga** Układy można tworzyć także programowo, jednak zaleca się stosowanie formatu XML.

Po dodaniu do układu domyślnego innych kontrolek (na przykład `ScrollView`, `EditText` i `ProgressBar`), tak że ich łączna liczba wynosi 30, wywołanie metody `setContentView()` zajmuje ponad 163 milisekundy.

Jak widać, czas rozwijania układu rośnie niemal liniowo wraz z liczbą kontrolek. Ponadto wykonywanie metody `setContentView()` zajmuje prawie 99% czasu od początku działania metody `onCreate()` do końca działania metody `onResume()`.

Możesz przeprowadzić własne pomiary po dodaniu kontrolek do układu lub ich usunięciu. Modyfikowanie układu za pomocą graficznego widoku plików XML w środowisku Eclipse jest bardzo łatwe. Jeśli już zdefiniowałeś układ aplikacji, powinieneś przede wszystkim zmierzyć czas jego rozwijania. Ponieważ rozwijanie układu zwykle odbywa się w metodzie `onCreate()` aktywności, czas rozwijania ma bezpośredni wpływ na czas uruchamiania aktywności, a także całej aplikacji. Dlatego zaleca się minimalizowanie czasu rozwijania układów.

Aby uzyskać pożądane skutki, można zastosować kilka technik. Większość z nich oparta jest na jednej zasadzie — zmniejszaniu liczby tworzonych obiektów. Można to zrobić przez zastosowanie różnych układów dających ostatecznie ten sam efekt wizualny, przez wyeliminowanie zbędnych obiektów lub przez odroczenie momentu tworzenia obiektów.

Układ RelativeLayout

Programiści aplikacji zwykle najpierw poznają układy liniowe (`LinearLayout`). Układ tego rodzaju jest częścią układu domyślnego z listingu 8.1, dlatego jest jednym z pierwszych, z jakimi stykają się programiści widoków `ViewGroup`. Łatwo też go zrozumieć, ponieważ układ liniowy jest kontenerem na kontrolki ułożone obok siebie lub jedna nad drugą.

Większość programistów poznających Android zaczyna od zagnieżdżania układów liniowych w celu uzyskania pożądanych efektów. Na listingu 8.3 przedstawiono przykładowe zagnieżdżone układy liniowe.

Listing 8.3. Zagnieżdżony układ liniowy

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

    <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal" >
        <TextView
            android:id="@+id/text1"
            android:layout_width="wrap_content" android:layout_height="wrap_content"
            android:text="str1"
android:textAppearance="?android:attr/textAppearanceLarge" />
        <TextView
            android:id="@+id/text2"
            android:layout_width="wrap_content" android:layout_height="wrap_content"
            android:text="str2"
android:textAppearance="?android:attr/textAppearanceLarge" />
    </LinearLayout>

    <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal" >
        <TextView
            android:id="@+id/text3"
            android:layout_width="wrap_content" android:layout_height="wrap_content"
            android:text="str3"
android:textAppearance="?android:attr/textAppearanceLarge" />
        <TextView
            android:id="@+id/text4"
            android:layout_width="wrap_content" android:layout_height="wrap_content"
            android:text="str4"
android:textAppearance="?android:attr/textAppearanceLarge" />
    </LinearLayout>

    <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
```

```
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal" >
        <TextView
            android:id="@+id/text5"
            android:layout_width="wrap_content" android:layout_height="wrap_content"
            android:text="str5"
android:textAppearance="?android:attr/textAppearanceLarge" />
        <TextView
            android:id="@+id/text6"
            android:layout_width="wrap_content" android:layout_height="wrap_content"
            android:text="str6"
android:textAppearance="?android:attr/textAppearanceLarge" />
    </LinearLayout>

</LinearLayout>
```

Podstawowe widoki w tym układzie to sześć widoków tekstowych. Cztery układy liniowe służą tu do określenia pozycji elementów.

W przedstawionym układzie występują dwa problemy. Oto one:

- Ponieważ układy liniowe są tu zagnieżdżone, hierarchia układów ma więcej poziomów, co spowalnia obsługę układu i kluczy.
- Cztery z dziesięciu obiektów służą tylko do wyznaczania pozycji.

Oba problemy można łatwo rozwiązać przez zastąpienie układów liniowych jednym układem względnym (`RelativeLayout`), co pokazano na listingu 8.4.

Listing 8.4. Układ względny

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

    <TextView
        android:id="@+id/text1"
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
        android:layout_alignParentTop="true"
        android:text="@string/str1"
android:textAppearance="?android:attr/textAppearanceLarge" />
    <TextView
        android:id="@+id/text2"
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:layout_toRightOf="@id/text1"
        android:layout_alignParentTop="true"
        android:text="@string/str2"
android:textAppearance="?android:attr/textAppearanceLarge" />
    <TextView
        android:id="@+id/text3"
        android:layout_alignParentLeft="true"
        android:layout_below="@id/text1"
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:text="@string/str3"
android:textAppearance="?android:attr/textAppearanceLarge" />
    <TextView
```

```
        android:id="@+id/text4"
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:layout_toRightOf="@id/text3"
        android:layout_below="@id/text2"
        android:text="@string/str4"
android:textAppearance="?android:attr/textAppearanceLarge" />
    <TextView
        android:id="@+id/text5"
        android:layout_alignParentLeft="true"
        android:layout_below="@id/text3"
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:text="@string/str5"
android:textAppearance="?android:attr/textAppearanceLarge" />
    <TextView
        android:id="@+id/text6"
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:layout_toRightOf="@id/text5"
        android:layout_below="@id/text4"
        android:text="@string/str6"
android:textAppearance="?android:attr/textAppearanceLarge" />

</RelativeLayout>
```

Jak widać, wszystkie sześć układów tekstowych umieszczono teraz w jednym układzie względnym. Dlatego wystarczy utworzyć siedem obiektów zamiast dziesięciu. Ponadto układ nie ma tylu poziomów co wcześniej — widoki tekstowe znajdują się teraz na drugim poziomie. Do określenia pozycji kontrolek służą atrybuty `layout_***`. W Androidzie zdefiniowanych jest wiele atrybutów tego typu, które można wykorzystać do określania pozycji różnych elementów układu. Oto przykładowe atrybuty tego rodzaju:

- `layout_above`,
- `layout_alignBaseline`,
- `layout_alignBottom`,
- `layout_alignLeft`,
- `layout_alignRight`,
- `layout_alignTop`,
- `layout_alignParentBottom`,
- `layout_alignParentLeft`,
- `layout_alignParentRight`,
- `layout_alignParentTop`,
- `layout_alignWithParentIfMissing`,
- `layout_below`,
- `layout_centerHorizontal`,
- `layout_centerInParent`,
- `layout_centerVertical`,
- `layout_column`,
- `layout_columnSpan`,
- `layout_gravity`,
- `layout_height`,

- layout_margin,
- layout_marginBottom,
- layout_marginLeft,
- layout_marginRight,
- layout_marginTop,
- layout_row,
- layout_rowSpan,
- layout_scale,
- layout_span,
- layout_toLeftOf,
- layout_toRightOf,
- layout_weight,
- layout_width,
- layout_x,
- layout_y.

■ **Uwaga** Niektóre atrybuty są charakterystyczne dla układów określonego rodzaju. Na przykład atrybuty layout_column, layout_columnSpan, layout_row i layout_rowSpan są przeznaczone do stosowania w układach opartych na siatce.

Układy względne są ważne zwłaszcza przy wyświetlaniu elementów list, ponieważ aplikacje nieraz pokazują na ekranie jednocześnie dziesięć, a nawet więcej takich elementów.

Scalanie układów

Inny sposób na zmniejszenie liczby poziomów w hierarchii układów to scalanie układów za pomocą znacznika <merge/>. Stosunkowo często głównym elementem układu jest układ ramowy (FrameLayout). Tak właśnie jest na listingu 8.5.

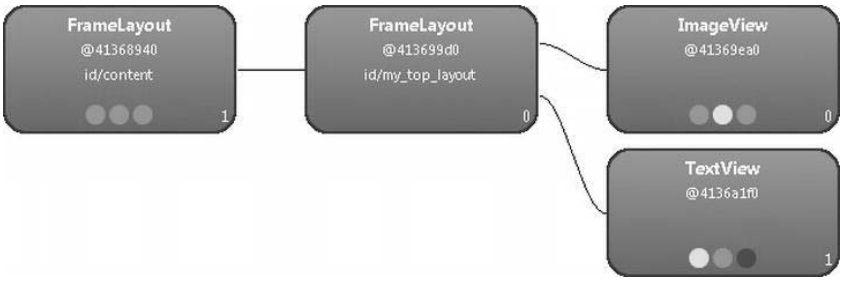
Listing 8.5. Układ ramowy

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:id="@+id/my_top_layout" >

    <ImageView
        android:layout_width="fill_parent"
        android:layout_height="fill_parent" />

    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/hello" />

</FrameLayout>
```

Ponieważ elementem nadrzędnym widoku zawartości aktywności jest także układ `FrameLayout`, ostatecznie w układzie znajdują się dwa obiekty `FrameLayout`. Oto one:

- obiekt `FrameLayout` dodany przez programistę,
- element nadrzędny widoku zawartości aktywności, czyli drugi obiekt `FrameLayout` (ma on tylko jeden element podrzędny — pierwszy obiekt `FrameLayout`).

Na rysunku 8.1 pokazano układ, który powstanie, jeśli układ `FrameLayout` zdefiniowany przez programistę ma dwa elementy podrzędne — `ImageView` i `TextView`.



Rysunek 8.1. *Układ FrameLayout będący układem podrzędnym innego układu FrameLayout*

Występuje tu jeden nadmiarowy układ `FrameLayout`. Liczbę poziomów układu można zmniejszyć przez zastąpienie własnego układu `FrameLayout` znacznikiem `<merge />`. Wtedy Android dołącza elementy podrzędne znacznika `<merge />` do nadrzędnego układu `FrameLayout`. Na listingu 8.6 pokazano nowy układ w formacie XML.

Listing 8.6. Znacznik <merge />

```
<?xml version="1.0" encoding="utf-8"?>
<merge xmlns:android="http://schemas.android.com/apk/res/android">

    <ImageView
        android:layout_width="fill_parent"
        android:layout_height="fill_parent" />

    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/hello" />

</merge>
```

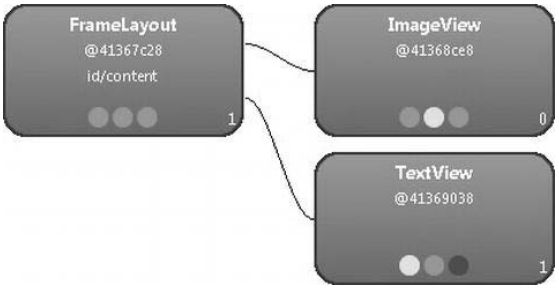
Jak widać, wystarczy zastąpić znacznik `<FrameLayout />` znacznikiem `<merge />`. Na rysunku 8.2 pokazano uzyskany układ.

Ponowne wykorzystanie układów

Android udostępnia znacznik `<include />`, który można stosować w układach w formacie XML w podobny sposób jak dyrektywę `#include` w kodzie w językach C i C++. Znacznik `<include />` powoduje dołączenie innego układu w formacie XML, co pokazano na listingu 8.7.

Znacznik `<include />` stosuje się w dwóch sytuacjach:

- gdy programista chce wielokrotnie wykorzystać ten sam układ;
- jeśli układ ma powtarzalne elementy, a oprócz nich części zależne od konfiguracji urządzenia (np. od orientacji ekranu — poziomej lub pionowej).



Rysunek 8.2. *Hierarchia po zastąpieniu znacznika <FrameLayout /> znacznikiem <merge />*

Na listingu 8.7 pokazano, że układ można dołączyć wielokrotnie, a przy tym zmienić niektóre jego parametry.

Listing 8.7. Wielokrotne dołączanie układu

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >

    <include android:id="@+id/myid1" android:layout="@layout/mylayout"
android:layout_margin="9dip" />
    <include android:id="@+id/myid2" android:layout="@layout/mylayout"
android:layout_margin="3dip" />
    <include android:id="@+id/myid3" android:layout="@layout/mylayout" />

</LinearLayout>
```

Na listingu 8.8 pokazano, w jaki sposób dołączyć układ tylko raz. Jednak w zależności od orientacji ekranu dołączany jest układ *layout-land/mylayout.xml* lub *layout-port/mylayout.xml*. Przyjęto tu, że istnieją dwie wersje pliku *mylayout.xml* — jedna z katalogu *res/layout-land* i druga z katalogu *res/layout-port*.

Listing 8.8. Dołączanie układu w zależności od orientacji ekranu

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >

    <include android:id="@+id/myid" android:layout="@layout/mylayout" />

</LinearLayout>
```

Gdy układ jest dołączany, można zmienić niektóre jego parametry, takie jak:

- identyfikator widoku głównego (`android:id`),
- parametry układu (`android:layout_*`).

■ **Uwaga** Nie trzeba zmieniać parametrów układu. W znaczniku `<include />` trzeba podać tylko atrybut `android:layout`.

Jak pokazano na listingu 8.8 dołączanie układu odbywa się dynamicznie w momencie jego rozwijania. Gdyby dołączanie odbywało się w czasie kompilacji, Android nie wiedziałby, który

z dwóch układów powinien dołączyć (*layout-land/mylayout.xml* czy *layout-port/mylayout.xml*). Dyrektywa `#include` w językach C i C++ działa inaczej, ponieważ jest obsługiwana przez preprocesor w czasie kompilacji.

Namiastki widoków

W rozdziale 1. wyjaśniono, że leniwe inicjowanie to wygodna technika służąca do odraczania tworzenia obiektów, zwiększania wydajności i potencjalnie zmniejszania zużycia pamięci (gdy okazuje się, że dane obiekty nigdy nie są potrzebne).

W Androidzie leniwe inicjowanie jest możliwe dzięki namiastce widoków — klasie `ViewStub`. Reprezentuje ona prosty, ukryty widok, który można zastosować w układzie, aby umożliwić leniwe rozwijanie zasobów układu, wtedy gdy są potrzebne. Listing 8.9 to zmodyfikowana wersja listingu 8.8. Pokazano tu, jak zastosować klasę `ViewStub` w układzie w formacie XML.

Listing 8.9. Klasa ViewStub w XML-owym układzie

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >

    <ViewStub
        android:id="@+id/mystubid"
        android:inflatedId="@+id/myid"
        android:layout="@layout/mylayout" />

</LinearLayout>
```

Przedstawiony tu układ `LinearLayout` w trakcie rozwijania obejmuje tylko jeden element podrzędny — `ViewStub`. Układ powiązany z obiektem `ViewStub`, `@layout/mylayout`, może być bardzo skomplikowany. Jego rozwinięcie może więc zająć dużo czasu (na razie układ ten jednak nie jest rozwijany). Układ z pliku *mylayout.xml* można rozwinąć w kodzie na dwa sposoby, przedstawione na listingach 8.10 i 8.11.

Listing 8.10. Rozwijanie układu w kodzie

```
ViewStub stub = (ViewStub) findViewById(R.id.mystubid);
// inflatedView będzie układem zdefiniowanym w pliku mylayout.xml
View inflatedView = stub.inflate();
```

Listing 8.11. Rozwijanie układu w kodzie za pomocą metody setVisibility()

```
View view = findViewById(R.id.mystubid);
view.setVisibility(View.VISIBLE); // Namiastka widoku zastępowana rozwiniętym układem
view = findViewById(R.id.myid); // W tym miejscu trzeba pobrać rozwinięty widok
```

Wprawdzie pierwszy sposób rozwijania układu (listing 8.10) wydaje się wygodniejszy, jednak zdaniem niektórych programistów występuje w nim drobna wada. W kodzie wiadomo, że widok jest namiastką, dlatego trzeba bezpośrednio rozwinąć układ. W większości sytuacji nie stanowi to problemu, dlatego zaleca się stosowanie tej techniki do rozwijania układu, jeśli w układzie aplikacji używana jest klasa `ViewStub`.

Druga metoda rozwijania układu (listing 8.11) jest bardziej uniwersalna, ponieważ nie trzeba używać tu klasy `ViewStub`. Jednak w przedstawionym tu rozwiązaniu nadal wiadomo, że stosowana jest klasa `ViewStub` (ponieważ w kodzie występują różne identyfikatory — `R.id.mystubid` i `R.id.myid`). Aby kod był w pełni uniwersalny, układ trzeba zdefiniować w sposób pokazany na listingu 8.12, a rozwijanie układu powinno wyglądać tak jak na listingu 8.13. Listing 8.12 jest prawie identyczny z listingiem 8.9. Różnica polega na tym, że tworzony jest tylko jeden identyfikator (`R.id.myid`)

zamiast dwóch. Podobnie listing 8.13 jest bardzo podobny do listingu 8.11 — zastąpiono jedynie identyfikator `R.id.mystubid` identyfikatorem `R.id.myid`.

Listing 8.12. Klasa ViewStub w XML-owym układzie bez zmiany identyfikatora

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >

    <ViewStub
        android:id="@+id/myid"
        android:layout="@layout/mylayout" />

</LinearLayout>
```

Listing 8.13. Rozwijanie układu w kodzie za pomocą metody setVisibility()

```
View view = findViewById(R.id.myid);
view.setVisibility(View.VISIBLE); // Namiastka widoku zastępowana rozwiniętym układem
view = findViewById(R.id.myid); // W tym miejscu trzeba pobrać rozwinięty widok
```

Kod z listingu 8.13 jest prawidłowy niezależnie od tego, czy w układzie występuje klasa `ViewStub`. Jeśli kod ma być uniwersalny i działać poprawnie niezależnie od stosowania klasy `ViewStub`, zaleca się używanie opisanego podejścia. Jednak dwa wywołania metody `findViewById()` są niewydajne. Aby częściowo rozwiązać problem, warto wykorzystać fakt, że obiekt `ViewStub` jest usuwany z rodzica w momencie wywołania metody `setVisibility(View.VISIBLE)`. Dlatego można najpierw (przed ponownym wywołaniem metody `findViewById()`) sprawdzić, czy widok nadal ma rodzica. Choć nie jest to idealne rozwiązanie (ponieważ jeśli klasa `ViewStub` występuje w układzie, to i tak trzeba dwukrotnie wywołać metodę `findViewById()`), uzyskano przynajmniej to, że metoda jest wywoływana jednokrotnie, jeśli programista nie używa klasy `ViewStub`. Nowa wersja kodu znajduje się na listingu 8.14.

Listing 8.14. Jednokrotne wywołanie metody findViewById() (jeśli jest to możliwe)

```
View view = findViewById(R.id.myid);
// Namiastka jest zastępowana rozwiniętym układem (jeśli w układzie występuje namiastka)
view.setVisibility(View.VISIBLE);
if (view.getParent() == null) {
    // Użyto namiastki, dlatego trzeba znaleźć zastępujący ją rozwinięty widok
    view = findViewById(R.id.myid);
} else {
    // Nie trzeba nic robić. Odpowiedni jest widok znaleziony za pierwszym razem
}
```

Kod z listingów 8.13 i 8.14 jest niestandardowy i zwykle nie trzeba stosować przedstawionego tu podejścia. Przeważnie dopuszczalne jest uwzględnianie w kodzie tego, że w układach używana jest klasa `ViewStubs`, dlatego można zastosować prosty sposób rozwijania namiastek, pokazany na listingu 8.10.

Narzędzia wspomagające tworzenie układów

W pakiecie SDK dostępne są dwa łatwe w użyciu narzędzia, mianowicie hierarchyviewer i layoutopt, które wspomagają tworzenie optymalnych układów. Te i inne narzędzia znajdują się w katalogu *tools* pakietu SDK.

Przeglądarka hierarchii

W pakiecie SDK Androida znajduje się bardzo przydatne narzędzie do wyświetlania i analizowania układu aplikacji. To narzędzie to hierarchyviewer. Rysunki 8.1 i 8.2 wygenerowano właśnie przy użyciu tego narzędzia. Nie tylko wyświetla ono szczegółowy układ aplikacji, ale też określa, ile czasu potrzeba na wymierzenie, rozmieszczenie i wyświetlenie każdej kontrolki. Wykrywa też, dla których kontrolek ten czas jest długi.

Narzędzie hierarchyviewer można stosować niezależnie lub bezpośrednio w środowisku Eclipse (w perspektywie *Hierarchy View*).

Narzędzie layoutopt

Innym narzędziem z pakietu SDK pomocnym przy tworzeniu układów jest `layoutopt`. Narzędzie to analizuje pliki układu i zaleca zmiany zwiększające jego wydajność.

Uruchomienie narzędzia `layoutopt` dla układu z listingu 8.5 pozwala uzyskać następujące informacje:

```
The root-level <FrameLayout/> can be replaced with <merge/>
```

(czyli: element główny `<FrameLayout/>` można zastąpić elementem `<merge/>`).

Okazuje się, że właśnie taką zmianę wprowadzono na listingu 8.6. Uruchomienie narzędzia layoutopt dla kodu z listingu 8.6 nie prowadzi do uzyskania żadnych zaleceń.

■ **Wskazówka** Aby uzyskać najlepsze efekty, należy zastosować najnowszą wersję narzędzia `layoutopt`.

Przed udostępnieniem aplikacji popraw wszelkie problemy wykryte przez narzędzie `layoutopt`. Niezoptymalizowane układy mogą spowalniać działanie aplikacji, a rozwiązanie wskazanych usterek jest zwykle proste.

Standard OpenGL ES

Renderowanie grafiki trójwymiarowej staje się coraz ważniejszą funkcją współczesnych urządzeń i aplikacji androidowych. Choć potrzeba sporo czasu, aby zostać ekspertem do renderowania grafiki trójwymiarowej, w najbliższych punktach poznasz łatwe w stosowaniu techniki, a także podstawowe zagadnienia, o których należy wiedzieć przed rozpoczęciem pisania kodu związanego z grafiką trójwymiarową. Jeśli chcesz dowiedzieć się czegoś więcej o korzystaniu ze standardu OpenGL ES w Androidzie, zajrzyj do książki *Pro OpenGL ES for Android* Mike'a Smithwicka i Mayanka Vermy.

Większość nowych urządzeń z Androidem obsługuje zarówno standard OpenGL ES 1.1, jak i OpenGL ES 2.0, natomiast starsze urządzenia są zgodne tylko z tym pierwszym. W grudniu 2011 roku około 90% urządzeń łączących się ze sklepem Android Market obsługiwało standard OpenGL ES 2.0.

Choć OpenGL ES to standard opracowany przez organizację Khronos Group, istnieje kilka jego implementacji. Oto zgodne ze standardem OpenGL ES procesory graficzne najczęściej spotykane w urządzeniach z Androidem:

- ARM Mali (np. Mali 400-MP4),
- Imagination Technologies PowerVR (np. PowerVR SGX543),
- Nvidia GeForce (Nvidia Tegra),
- Qualcomm Adreno (kupiony od AMD; wcześniej nosił nazwę ATI Imageon; jest zintegrowany z układami Qualcomm Snapdragon).

Rozszerzenia

Standard OpenGL obsługuje rozszerzenia, co pozwala na wbudowanie w niektóre procesory graficzne nowych funkcji. Na przykład w tabletach Samsung Galaxy Tab 10.1 (z procesorem graficznym Nvidia Tegra 2) obsługiwane są następujące rozszerzenia:

- GL_NV_platform_binary,
- GL_OES_rgb8_rgba8,
- GL_OES_EGL_sync,
- GL_OES_fbo_render_mipmap,
- GL_NV_depth_nonlinear,
- GL_NV_draw_path,
- GL_NV_texture_npot_2D_mipmap,
- GL_OES_EGL_image,
- GL_OES_EGL_image_external,
- GL_OES_vertex_half_float,
- GL_NV_framebuffer_vertex_attrib_array,
- GL_NV_coverage_sample,
- GL_OES_mapbuffer,
- GL_ARB_draw_buffers,
- GL_EXT_Cg_shaders,
- GL_EXT_packed_float,
- GL_OES_texture_half_float,
- GL_OES_texture_ float,
- GL_EXT_texture_array,
- GL_OES_compressed_ETC1_RGB8_texture,
- GL_EXT_texture_compression_latc,
- GL_EXT_texture_compression_dxt1,
- GL_EXT_texture_compression_s3tc,
- GL_EXT_texture_filter_anisotropic,
- GL_NV_get_text_image,
- GL_NV_read_buffer,
- GL_NV_shader_framebuffer_fetch,
- GL_NV_fbo_color_attachements,
- GL_EXT_bgra,
- GL_EXT_texture_format_BGRA8888,
- GL_EXT_unpack_subimage,
- GL_NV_texture_compression_st3c_update.

Na listingu 8.15 pokazano, jak pobrać listę rozszerzeń obsługiwanych przez urządzenie. Ponieważ standard OpenGL ES 2.0 obecnie nie jest obsługiwany przez emulator Androida, kod należy uruchomić na urządzeniu.

Listing 8.15. Rozszerzenia standardu OpenGL ES

```
// Lista rozszerzeń zwracana jako jeden łańcuch znaków (można znaleźć w nim konkretne rozszerzenie)

String extensions = GLES20.glGetString(GLES20.GL_EXTENSIONS);

Log.d(TAG, "Rozszerzenia: " + extensions);
```

Aby z powodzeniem wykonać kod z listingu 8.15 i pobrać listę rozszerzeń, potrzebny jest kontekst związany ze standardem OpenGL. Kod ten można uruchomić na przykład w metodzie `onSurfaceChanged()` klasy `GLSurfaceView.Renderer`. Wywołanie kodu poza odpowiednim kontekstem spowoduje wyświetlenie w narzędziu `LogCat` komunikatu *call to OpenGL ES API with no current context*.

Przedstawione wcześniej rozszerzenia można podzielić na kilka grup:

- GL_OES_*,
- GL_ARB_*,
- GL_EXT_*,
- GL_NV_*.

Rozszerzenia z rodziny GL_OES_* zostały zatwierdzone przez grupę roboczą odpowiedzialną za OpenGL ES w operacji Khronos. Choć rozszerzenia te nie są wymagane w standardzie OpenGL, są powszechnie dostępne.

Rozszerzenia GL_ARB_* zostały zatwierdzone przez jednostkę OpenGL Architecture Review Board. Rozszerzenia GL_EXT_* są obsługiwane przez dużą część producentów, a rozszerzenia GL_NV_* są charakterystyczne dla sprzętu Nvidii. Na podstawie obsługiwanych rozszerzeń można zwykle stwierdzić, która firma jest producentem procesora graficznego zastosowanego w urządzeniu.

- Rozszerzenia procesorów ARM mają przedrostek GL_ARM.
- Rozszerzenia procesorów firmy Imagination Technologies mają przedrostek GL_IMG.
- Rozszerzenia procesorów Nvidii mają przedrostek GL_NV.
- Rozszerzenia procesorów firmy Qualcomm mają przedrostki GL_QUALCOMM, GL_AMD i GL_ATI.

Uwaga Więcej informacji znajdziesz na stronach *http://www.khronos.org/registry/gles* i *http://www.opengl.org/registry*.

Ponieważ nie wszystkie urządzenia obsługują te same rozszerzenia (warto przypomnieć tu o fragmentacji rynku), trzeba zachować dużą ostrożność przy optymalizowaniu kodu pod kątem konkretnego urządzenia. Czasem kod, który działa w jednym urządzeniu, nie będzie działał w innym. Typowymi przykładami technik obsługiwanych tylko w niektórych urządzeniach są tekstury typu NPOT (ang. *Non-Power-Of-Two*) i format kompresji tekstur. Jeśli chcesz, aby kod był zgodny także z urządzeniami bez Androida, zwróć uwagę na rozszerzenia obsługiwane przez docelowe urządzenia. Obecnie urządzenia Apple'a (iPhone, iPod i iPad) są oparte na procesorze PowerVR firmy Imagination Technologies, jednak w przyszłych urządzeniach Apple'a mogą pojawić się inne procesory.

Kompresja tekstur

Od tekstur zależy wygląd aplikacji wykorzystujących OpenGL ES. Choć korzystanie z nieskompresowanych tekstur jest proste, tekstury tego rodzaju powodują szybki wzrost wielkości aplikacji. Na przykład nieskompresowana tekstura o wymiarach 256×256 w formacie RGBA8888 może zajmować 256 kilobajtów pamięci.

Nieskompresowane tekstury negatywnie wpływają na wydajność, ponieważ trudniej jest je zapisać w pamięci podręcznej niż tekstury skompresowane (z uwagi na wielkość). Ponadto wymagają dłuższego dostępu do pamięci, co wpływa także na zużycie energii. Aplikacje z nieskompresowanymi teksturami są również większe, dlatego dłużej trwa ich pobieranie i instalowanie. Oprócz tego ilość pamięci w urządzeniach z Androidem jest zwykle ograniczona, dlatego w aplikacjach należy zajmować jej jak najmniej.

Na poniższej liście wymieniono formaty kompresji tekstur obsługiwane w urządzeniu Samsung Galaxy Tab 10.1:

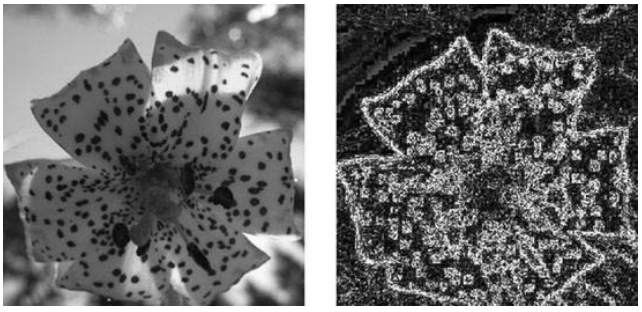
- GL_OES_compressed_ETC1_RGB8_texture,
- GL_EXT_texture_compression_latc,
- GL_EXT_texture_compression_dxt1,
- GL_EXT_texture_compression_s3tc.

Format kompresji ETC1 został opracowany w firmie Ericsson (ETC to akronim nazwy *Ericsson Texture Compression*) i jest obsługiwany przez większość urządzeń z Androidem, w tym przez wszystkie takie urządzenia zgodne ze standardem OpenGL ES 2.0. Dlatego jest to najbezpieczniejszy format ze względu na liczbę obsługiwanych urządzeń. Używa się w nim 4 bitów na piksel zamiast 24 (format ETC1 nie obsługuje kanału alfa), co zapewnia sześciokrotny poziom kompresji. Jeśli dodać 16-bajtowy nagłówek, tekstura o wymiarach 256×256 w formacie ETC1 zajmuje 32 784 bajty.

Istnieje wiele narzędzi do tworzenia tekstur w formacie ETC1. W pakiecie SDK Androida znajduje się uruchamiane z wiersza poleceń narzędzie etc1tool. Służy ono do przekształcania obrazów w formacie PNG na skompresowane obrazy w formacie ETC1. Poniższa instrukcja kompresuje plik *lassen.png* (183 kilobajty) i pokazuje różnicę między pierwotnym a skompresowanym plikiem:

```
etc1tool lassen.png --encode --showDifference lassen_diff.png
```

Jeśli programista nie podał pliku wynikowego, nazwa pliku jest generowana na podstawie pliku wejściowego. Tu plik wynikowy to *lassen.pkm* (zajmuje 33 kilobajty). Na rysunku 8.3 pokazano różnicę między skompresowaną grafiką a oryginałem.



Rysunek 8.3. *Pierwotny rysunek (po lewej) i różnica w stosunku do skompresowanej wersji (po prawej)*

■ **Uwaga** Za pomocą narzędzia etc1tool można też przekształcić plik w formacie ETC1 na obraz w formacie PNG. Więcej informacji o tym narzędziu znajdziesz na stronie *http://d.android.com/guide/developing/tools/etc1tool.html*.

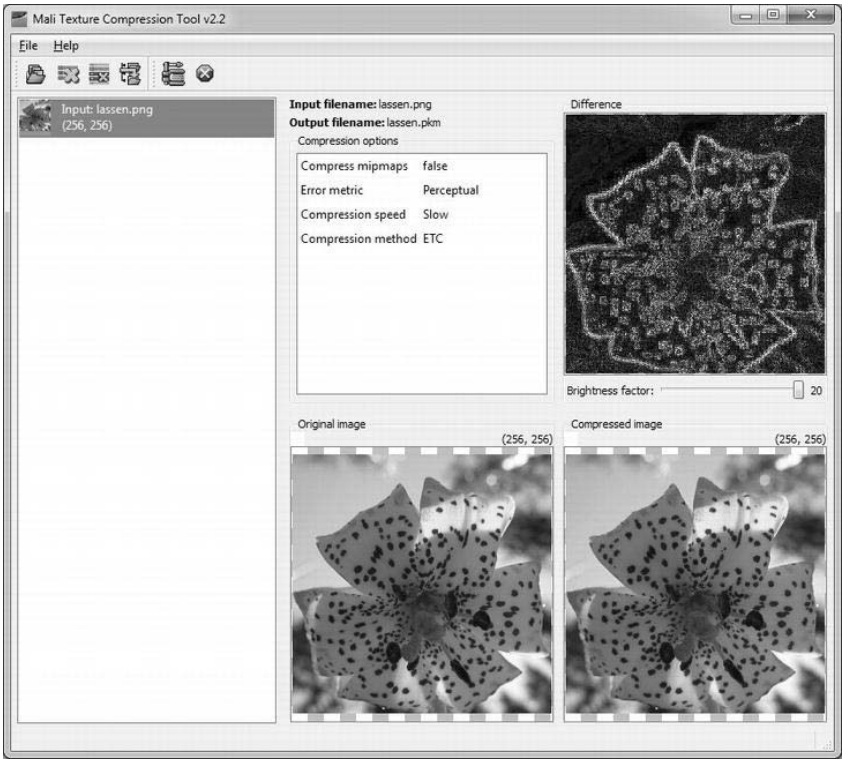
Niestety, narzędzie etc1tool z pakietu SDK Androida udostępnia niewiele opcji. Dlatego programiści, którzy chcą zachować kontrolę nad kompresją i jakością grafiki, powinni stosować inne narzędzie — etcpack. Można je pobrać z witryny firmy Ericsson (*http://devtools.ericsson.com/etc*). Pozwala ono ustawić następujące opcje:

- szybkość kompresji (mniejsza szybkość to wyższa jakość),
- miarę błędu (percepcyjna lub niepercepcyjna),
- orientację.

Jeśli tekstury są generowane „stacjonarnie” (czyli nie na samym urządzeniu z Androidem), zawsze należy zastosować jak najwyższą jakość kompresji. Ponieważ ludzkie oko jest wrażliwsze na kolor zielony niż na czerwony lub niebieski, należy ponadto wybrać tryb percepcyjny dla miary błędu (algorytm zapewnia wtedy większą wierność koloru zielonego względem oryginału kosztem kolorów czerwonego i niebieskiego, co zmniejsza maksymalny stosunek sygnału do szumu).

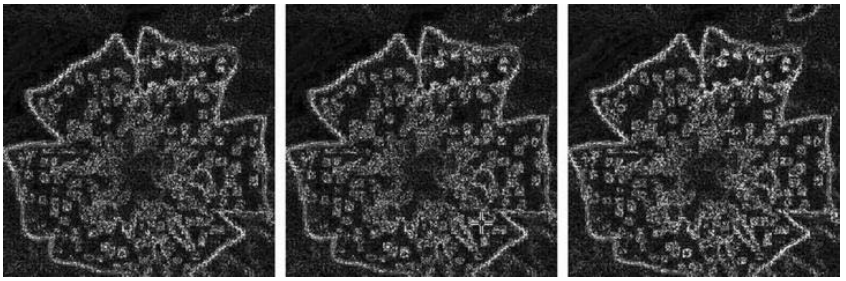
Choć różnice nie zawsze są widoczne, nie ma powodu, aby udostępniać aplikację z teksturami niższej jakości. W formacie ETC1 tekstury wyższej jakości mają ten sam rozmiar jak te o niższej jakości. Choć generowanie tekstur wyższej jakości trwa dłużej (dlatego na etapie programowania wielu programistów decyduje się korzystać z tekstur niższej jakości), przed udostępnieniem aplikacji zawsze należy pamiętać o zastosowaniu tekstur wyższej jakości.

Jeszcze inna możliwość to narzędzie ARM Mali GPU Compression Tool. Można je pobrać bezpłatnie ze strony *http://www.malideveloper.com/developer-resources/tools*. Narzędzie to udostępnia podobne opcje jak etcpack, jednak ma interfejs graficzny (przy czym dostępna jest też wersja uruchamiana z wiersza poleceń). Na rysunku 8.4 pokazano narzędzie ARM Mali GPU Compression Tool modyfikujące opisywany już wcześniej plik.



Rysunek 8.4. *Narzędzie ARM Mali GPU, przeznaczone do kompresji grafiki*

Na rysunku 8.5 pokazano różnicę w jakości dla kompresji wysokiej jakości (powolnej) i niskiej jakości (szybkiej). Widać, że różnice między grafikami w różnych trybach nie są wyraźne.



Rysunek 8.5. *Wysoka jakość (po lewej), średnia jakość (pośrodku) i niska jakość (po prawej)*

W witrynie dla programistów oprogramowania na procesory ARM Mali dostępnych jest wiele narzędzi. Liczne narzędzia do tworzenia i debugowania kodu można znaleźć w witrynach firm ARM, Imagination Technologies, Nvidia i Qualcomm:

- *http://www.malideveloper.com/developer-resources/tools*,
- *http://www.imgtec.com/powervr/insider/sdkdownloads*,
- *http://developer.nvidia.com/tegra-android-development-pack*,
- *http://developer.qualcomm.com/develop/mobile-technologies/graphics-optimization-adreno*.

Na przykład firma Imagination Technologies także udostępnia narzędzie do tworzenia skompresowanych tekstur, PVRTexTool, a w witrynie Nvidii znajduje się kompletny pakiet SDK (obejmujący pakiet SDK Androida, pakiet NDK, środowisko Eclipse i przykładowy kod).

■ **Uwaga** W niektórych z wymienionych witryn trzeba się zarejestrować, aby uzyskać dostęp do narzędzi i dokumentów.

Począwszy od Androida 2.2 (poziom 8. interfejsu API) dostępne są następujące klasy pomagające w korzystaniu z tekstur w formacie ETC1:

- `android.opengl.ETC1`,
- `android.opengl.ETC1Util`,
- `android.opengl.ETC1Util.ETC1Texture`.

Aplikacja może kompresować grafikę do formatu ETC1 dynamicznie. Służą do tego metody `ETC1.encodeImage()` i `ETC1Util.compressTexture()`. Dynamiczna kompresja jest przydatna, jeśli tekstur ETC1 nie można wygenerować w trybie „stacjonarnym”, na przykład gdy tekstury są oparte na plikach graficznych zapisanych w urządzeniu (np. zdjęcie jednej z osób z listy kontaktów).

■ **Wskazówka** Choć format ETC1 nie obsługuje przezroczystości (w skompresowanej teksturze nie ma kanału alfa), można wykorzystać inną, jednokanałową teksturę obejmującą tylko informacje o przezroczystości i połączyć obie tekstury w programie cieniowania.

Inne formaty kompresji tekstur

Choć ETC1 to najpopularniejszy format kompresji tekstur, istnieją też inne, obsługiwane przez niektóre urządzenia. Oto wybrane formaty:

- PowerVR Texture Compression (PVRTC),

- ATI Texture Compression (ATC lub ATITC),
- S3 Texture Compression (S3TC) z wersjami od DXT1 do DXT5.

Warto wypróbować różne formaty w zależności od urządzeń docelowych. Ponieważ dany procesor graficzny może być zoptymalizowany pod kątem powiązanego z nim formatu kompresji (np. procesor graficzny PowerVR jest zoptymalizowany na potrzeby formatu PVRTC), lepsze efekty można uzyskać za pomocą specyficznych formatów kompresji niż przy stosowaniu bardziej standardowego formatu ETC1.

Uwaga Do generowania tekstur w formacie PVRTC można też stosować narzędzie texturetool z systemu iOS Apple'a.

Manifest

W manifeście aplikacji wykorzystującej OpenGL należy określić dwie rzeczy:

- którą wersję standardu OpenGL urządzenie ma obsługiwać,
- które formaty kompresji wykorzystano w aplikacji.

Na listingu 8.16 pokazano, jak określić, że urządzenie ma być zgodne ze standardem OpenGL ES 2.0, a aplikacja obsługuje tylko dwa formaty kompresji: ETC1 i PVRTC.

Listing 8.16. Manifest aplikacji korzystającej z OpenGL

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.apress.proandroid.opengl"
    android:versionCode="1" android:versionName="1.0" >

    <uses-sdk android:minSdkVersion="8" />

    <uses-feature android:glEsVersion="0x00020000" />

    <supports-gl-texture android:name="GL_OES_compressed_ETC1_RGB8_texture" />
    <supports-gl-texture android:name="GL_IMG_texture_compression_pvrtc" />
    ...

</manifest>
```

Uwaga Wersja standardu OpenGL ES jest zapisywana w formacie 16.16, dlatego OpenGL ES 2.0 to 0x0002000.

W sklepie Android Market opisane informacje są używane do filtrowania aplikacji dostępnych dla łączących się urządzeń. Na przykład w urządzeniu zgodnym tylko ze standardem OpenGL ES 1.1 nie są widoczne aplikacje wymagające OpenGL ES 2.0. Podobnie w urządzeniach obsługujących tylko format ETC1 nie pojawiają się aplikacje korzystające tylko z formatów PVRTC i ATC.

Jeśli w manifeście aplikacji nie określono żadnego formatu kompresji tekstur, sklep Android Market nie stosuje filtrów dotyczących takiego formatu (z uwagi na założenie, że program obsługuje wszystkie możliwe formaty). Dlatego może się okazać, że użytkownik zainstaluje aplikację, która nie będzie działać. Może to prowadzić do przyznawania niskich ocen programowi.

Mipmapy

Obiekty w trójwymiarowej scenie często pojawiają się w tle i nie zajmują dużo przestrzeni na ekranie. Dlatego stosowanie tekstury o wymiarach 256×256 dla obiektu zajmującego na ekranie obszar 10×10 pikseli prowadzi do marnotrawienia zasobów (pamięci i przepustowości łącza). Mipmapy pozwalają rozwiązać ten problem przez określenie różnych poziomów szczegółowości w teksturach, co pokazano na rysunku 8.6.

Rysunek 8.6. *Mipmapy o wymiarach od 256×256 do 1×1*

Choć zestaw bitmap zajmuje zwykle około 33% więcej pamięci niż pojedynczy pierwotny rysunek, pozwala nie tylko poprawić wydajność, ale też jakość grafiki.

Na rysunku 8.6 widać, że każdy rysunek to wersja pierwotnej grafiki, ale na różnym poziomie szczegółowości. Oczywiście trudno jest dostrzec kwiat w wersji o wymiarach 1×1.

Do generowania mipmap można zastosować narzędzie ARM Mali GPU Texture Compression Tool. Zamiast tworzyć jeden plik *.pkm*, narzędzie to generuje obrazy na wszystkich poziomach szczegółowości (aż do grafiki o wymiarach 1×1) i dla każdego poziomu tworzy odrębny plik *.pkm*. W aplikacji trzeba następnie wczytywać te poziomy jeden po drugim — wywołując na przykład metodę `ETC1Util.loadTexture()` lub `GLES20.glTexImage2D()`.

Ponieważ nie wszystkie urządzenia są identyczne, nie każde wymaga tego samego poziomu szczegółowości. Na przykład urządzenia Google TV o rozdzielczości HD 1920×1080 wymagają bardziej szczegółowego obrazu niż Samsung Nexus S o rozdzielczości WVGA 800×480. Dlatego dla urządzenia Google TV potrzebna może być tekstura o wymiarach 256×256, natomiast dla urządzenia Nexus S wystarczająca jest tekstura o wymiarach 128×128.

■ **Uwaga** Przy wyborze tekstury, którą aplikacja ma zastosować, nie należy uwzględniać tylko rozdzielczości ekranu. Na przykład w telewizorach Google TV firmy Sony (rozdzielczość 1920×1080) używany jest procesor PowerVR SGX535 o szybkości 400MHz, który jest mniej wydajny niż procesor PowerVR SGX540 o szybkości 384MHz z nowszego urządzenia Samsung Galaxy Nexus (rozdzielczość 1280×720).

Sposób renderowania tekstur zależy też od parametrów ustawionych za pomocą jednej z metod `glTexParameter()` (np. metody `glTexParameteri()`). Można na przykład ustawić funkcję dostosowującą wielkość tekstury. W tym celu należy wywołać instrukcję `glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, param)`, gdzie `param` to jedna z następujących wartości:

- `GL_NEAREST`,
- `GL_LINEAR`,

- `GL_NEAREST_MIPMAP_NEAREST`,
- `GL_NEAREST_MIPMAP_LINEAR`,
- `GL_LINEAR_MIPMAP_NEAREST`,
- `GL_LINEAR_MIPMAP_LINEAR`.

Choć funkcja `GL_NEAREST` jest zwykle szybsza niż `GL_LINEAR`, a `GL_LINEAR` jest szybsza od czterech pozostałych, wolniejsze funkcje zapewniają wyższą jakość grafiki. Parametry aplikacji mogą zależeć od preferencji użytkownika (jeśli programista umożliwił konfigurowanie jakości renderowania), jednak użytkownicy często nie mają wystarczającej cierpliwości, aby wypróbować różne ustawienia w celu znalezienia kombinacji, która dobrze się sprawdza w ich urządzeniach. Dlatego w aplikacji należy postarać się ustalić optymalną konfigurację ustawień OpenGL.

W OpenGL można skonfigurować wiele ustawień. Zgodnie z niektórymi ustawieniami domyślnymi jakość grafiki jest traktowana priorytetowo względem wydajności. Warto więc wiedzieć, jakie opcje można zmienić w aplikacji. Na przykład w dokumentacji na stronie *http://www.khronos.org/opengles/sdk/docs/man/* opisano różne parametry, które można ustawić dla tekstur.

Różne pliki APK

Ponieważ pożądana może być obsługa różnych formatów kompresji tekstur (w celu optymalizacji programu pod kątem różnych urządzeń), a mipmapy zajmują sporo miejsca, aplikacja może przekroczyć limit pamięci określony w sklepie Android Market (obecnie limit ten wynosi 50 megabajtów).

W takiej sytuacji programista ma trzy możliwości:

- Zmniejszenie wielkości aplikacji, na przykład przez obsługę tylko formatu ETC1.
- Pobieranie tekstur ze zdalnego serwera po zainstalowaniu aplikacji.
- Wygenerowanie różnych plików APK (każdy z odrębnym zestawem tekstur).

W sklepie Android Market można opublikować różne pliki APK aplikacji, każdy przeznaczony dla innej konfiguracji. W jednym pliku APK można używać tylko tekstur w formacie ETC1, a w innym stosować same tekstury PVRTC (zoptymalizowane pod kątem androidowych urządzeń z procesorem PowerVR). Takie pliki APK mogą być dostępne w sklepie Android Market pod jedną nazwą, a sklep odpowiada za wybór pliku odpowiedniego dla danego urządzenia. Użytkownicy nie muszą samodzielnie określać pliku APK do pobrania i zainstalowania, ponieważ wybór odbywa się automatycznie i niezauważalnie dla odbiorcy.

Uwaga Nie wszystkie sklepy z aplikacjami na Android obsługują tę funkcję, dlatego jeśli planujesz dystrybucję programu w różnych sklepach, staraj się udostępniać jeden plik APK dla wszystkich urządzeń.

Oczywiście tekstury nie są jedynym powodem, dla którego trzeba (lub warto) udostępniać różne pliki APK. Czasem warto na przykład utworzyć mniejszy plik APK dla starszych urządzeń i większy, z dodatkowymi funkcjami, dla nowszego sprzętu. Choć udostępnianie różnych plików APK jest możliwe, komplikuje ono pielęgnowanie kodu i proces rozpowszechniania aplikacji. Dlatego jeśli to możliwe, lepiej jest rozpowszechniać pojedynczy plik APK.

Programy cieniowania

Standard OpenGL ES 2.0 obsługuje język OpenGL ES Shading Language, co pozwala zastąpić potok przekształceń opartych na stałych funkcjach i potok fragmentów znany z wersji OpenGL ES 1.x.

Wspomniany język oparty jest na C i zapewnia większą kontrolę nad potokiem operacji OpenGL, ponieważ umożliwia pisanie własnych programów cieniowania dla wierzchołków i fragmentów.

Programy cieniowania, podobnie jak wszystkie programy języka C, mogą być zarówno bardzo proste, jak i niezwykle skomplikowane. Choć nie istnieją uniwersalne reguły, należy dążyć do redukowania złożoności programów cieniowania, ponieważ mogą mieć one istotny wpływ na wydajność.

Złożoność sceny

Renderowanie złożonych scen trwa oczywiście dłużej niż prostych. Dlatego łatwym sposobem na zwiększenie liczby klatek jest uproszczenie renderowanych scen (przy zachowaniu akceptowalnej jakości grafiki). W omówieniu tekstur wspomniano już, że bardziej oddalone obiekty mogą być mniej szczegółowe i składać się z mniejszej liczby trójkątów. Prostsze obiekty zajmują mniej pamięci i w mniejszym stopniu obciążają łącza.

Odrzucanie

Choć procesory graficzne dobrze radzą sobie z geometrią i potrafią ustalić, które obiekty mają być widoczne, w aplikacji należy próbować odrzucać obiekty znajdujące się poza bryłą widzenia. Pozwala to wyeliminować polecenia wyświetlania obiektów, które zostaną odrzucone (ponieważ są niewidoczne).

Istnieje wiele technik odrzucania obiektów, a nawet trójkątów. Choć omawianie tych metod wykracza poza zakres tej książki, wspomnę, że niższa od oczekiwanej liczba klatek może wynikać ze złych algorytmów odrzucania. Stosunkowo łatwo można na przykład szybko wyeliminować obiekty znajdujące się za kamerą.

■ **Uwaga** Przeważnie warto włączyć odrzucanie tylnych ścian, aby nie wyświetlać trójkątów znajdujących się na niewidocznych od strony kamery ścianach obiektów.

Tryb renderowania

Renderer w OpenGL domyślnie renderuje scenę niezależnie od zmian w jej wyglądzie. Czasem między klatkami wygląd sceny się nie zmienia. Dlatego warto nakazać rendererowi generowanie sceny tylko na żądanie. Efekt ten można uzyskać za pomocą trybu renderowania, ustawianego w klasie `GLSurfaceView`.

Aby ustawić tryb renderowania w klasie `GLSurfaceView`, należy wywołać metodę `setRenderMode()` i przekazać do niej jedną z dwóch następujących wartości:

- `RENDERMODE_CONTINUOUSLY`,
- `RENDERMODE_WHEN_DIRTY`.

Jeśli tryb renderowania jest ustawiony na `RENDERMODE_WHEN_DIRTY`, renderer generuje scenę w momencie tworzenia powierzchni lub przy wywołaniu metody `GLSurfaceView.requestRender()`.

Zużycie energii

Jedną z wielkich zalet współczesnych procesorów graficznych jest możliwość ich uśpienia lub przynajmniej bardzo szybkiego zwolnienia w okresach bezczynności. Procesor graficzny może na przykład (częściowo lub całkowicie) wstrzymać działanie między dwiema klatkami, jeśli nie ma do wykonania żadnych operacji. Prowadzi to do zmniejszenia zużycia energii i wydłużenia czasu pracy na baterii.

Po uzyskaniu akceptowalnej liczby klatek można kontynuować optymalizowanie kodu — choćby po to, aby zmniejszyć zużycie energii. Im szybciej klatki są renderowane, tym szybciej procesor graficzny przechodzi w stan bezczynności i tym dłużej wytrzymuje bateria (a użytkownik może dłużej korzystać z aplikacji). W niektórych programach doskonałym sposobem na wydłużenie czasu pracy na baterii przy korzystaniu z OpenGL jest renderowanie klatek tylko przy zmianie wyglądu sceny. Umożliwia to opisany wcześniej tryb renderowania `RENDERMODE_WHEN_DIRTY`.

Podsumowanie

Najnowsze urządzenia z Androidem mają bardzo duże możliwości i doskonale obsługują grafikę — zarówno dwu-, jak i trójwymiarową. Choć niektóre optymalizacje są mniej istotne niż jeszcze kilka lat temu, pojawiają się nowe wyzwania, które związane są z wyższą rozdzielczością obsługiwaną przez urządzenia, niemal powszechną zgodnością ze standardem OpenGL ES 2.0 i coraz wyższymi oczekiwaniami użytkowników. W tym rozdziale tylko pokrótce omówiono standard OpenGL ES i przedstawiono wybrane, łatwe do zaimplementowania techniki, które pozwalają uzyskać istotne korzyści. Na szczęście istnieje wiele źródeł, z których zarówno początkujący, jak i zaawansowani dowiedzą się czegoś więcej o standardzie OpenGL. Warto też zajrzeć do dokumentacji udostępnianej przez producentów procesorów graficznych (firmy ARM, Imagination Technologies, Nvidia, Qualcomm), gdzie opisane są sposoby optymalizowania renderowania pod kątem poszczególnych procesorów tego typu.

Skorowidz

C

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

Q

R

S

Ś

T

U

V

W

X

Z

Platforma Android wciąż się rozwija, a użytkownicy oczekują coraz bardziej zaawansowanych aplikacji. Istotne jest to, że muszą one działać perfekcyjnie — nie ma tu miejsca na zacięcia, spowolnienia lub brak odpowiedzi. Programiści często muszą dopracowywać, poprawiać i optymalizować wydajność tworzonych programów. Zagadnienia te zyskują na znaczeniu, ponieważ aplikacje stają się coraz bardziej złożone. Wydajniejsze programy pozwalają programiście aplikacji na Android uzyskać wyższe oceny i ostatecznie odnieść większy sukces. Jak to osiągnąć? Jest to zadanie trudne, ale wykonalne!

Z tego podręcznika dowiesz się, jak dopracować aplikacje na Android; zapewnić ich stabilność, wydajność i szybkość działania. Zobaczysz, jak pisać aplikacje w Javie, C i obu tych językach z użyciem pakietów SDK i NDK Androida. W trakcie lektury odkryjesz, jak optymalizować kod oparty na OpenGL, jak zmniejszyć zużycie pamięci oraz zużycie baterii przez Twoją aplikację. Ponadto nauczysz się testować wydajność aplikacji oraz korzystać z nowości, jaką jest RenderScript. Po zapoznaniu się z tą książką staniesz się lepszym programistą, a jakość Twoich aplikacji wzrośnie.

Sięgnij po tę książkę i:

- **zoptymalizuj aplikację w Javie za pomocą pakietu SDK**
- **świadomie używaj zasobów urządzenia**
- **wydajnie korzystaj z wielowątkowości i synchronizacji**
- **testuj aplikację pod kątem wydajności**
- **zagwarantuj najlepsze wrażenia jej użytkownikom!**

Wydajność to jedna z najważniejszych cech aplikacji — zadbaj o nią!

ISBN 978-83-246-5555-7
9 788324 655557
Cena: 49,00 zł