Learn about the most popular 3D graphics engine for game and graphical apps development

Learn OpenGL ES

For Mobile Game and Graphics Development

Prateek Mehta

Apress®

For your convenience Apress has placed some of the front matter material after the index. Please use the Bookmarks and Contents at a Glance links to access them.

Apress®

Contents at a Glance

Chapter 1

Benefits of the New API

In this chapter I introduce you to OpenGL ES 2.0, and account for its increasing popularity compared to older graphic rendering APIs for embedded devices. I describe OpenGL ES 2.0's support from computer-graphics communities and leading embedded and mobile device vendors, which helps to ensure its increasing popularity. Finally, I show how easy it is to get started with ES 2.0 on Android devices, when we take our first step towards game development, by creating a blank OpenGL surface view.

This chapter assumes you have some experience of setting up Android Software Development Kit (SDK) for Eclipse and installing SDK Platform for various API levels from SDK Manager.

Modern Graphic-rendering API

OpenGL ES (Open Graphics Library for Embedded Systems) is an API (Application Programming Interface) for rendering 3D graphics on embedded devices, such as mobiles, tablets, and gaming consoles.

The OpenGL ES 1.0 and ES 1.1 APIs (referred to jointly as OpenGL ES 1.x) were released by the non-profit *Khronos Group* as a fixed-function graphic-rendering API. OpenGL ES 1.x API does not provide graphics application developers full access to underlying hardware, because most rendering functions in this API are hard-coded, leading to popular names—"fixed-function graphic rendering API" or "fixed-function pipeline."

Unlike OpenGL ES 1.x API, OpenGL ES 2.0 API was released as a programmable graphic-rendering API (programmable pipeline), giving developers full access to the underlying hardware through *shaders* (discussed in Chapter 3).

Graphics rendered through a fixed-function pipeline involve device-provided algorithms for most rendering effects. These algorithms (and the rendering functions based on them) cannot be modified. They are fixed because they were made for special purpose graphics cards, for a specific data-flow. Because of the fixed functionality of OpenGL ES 1.x API, graphics hardware could be optimized for faster rendering.

In contrast, a programmable graphic-rendering API is a more flexible API and requires a general purpose graphics card, enabling graphic developers to unleash the huge potential of modern GPUs. Technically, the programmable pipeline is slower than the fixed function pipeline; however, graphics rendered using the programmable pipeline can be greatly enhanced because of flexibility offered by new general purpose graphics cards. OpenGL ES 2.0 combines *GLSL* (*OpenGL Shading Language*) with a modified subset of OpenGL ES 1.1 that has removed any fixed functionality. Chapter 3 discusses OpenGL Shading Language.

> **Note** *GLSL* is the OpenGL Shading Language for programming vertex and fragment shaders. Shaders are programs in programmable pipelines that help users work on two separate aspects of object rendering: vertex marking and color filling.

With OpenGL ES 2.0, enhancements in various effects, such as lighting/shading effects (as shown in Figure 1-1—a basic shading example), no longer have any restrictions, compared to ES 1.x. What is required is transformation of creative ideas for any such effects into algorithms, then into custom functions executed on the graphics card, which would be impossible in ES 1.x.

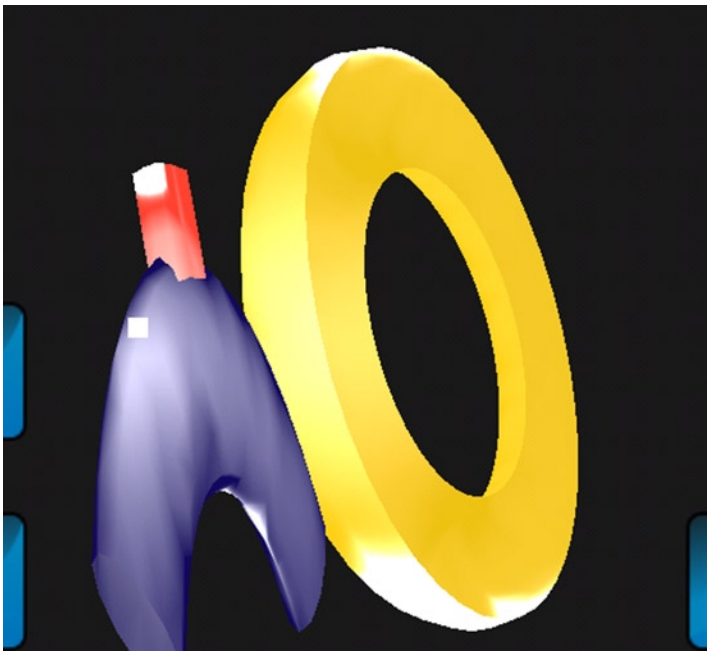

***Figure 1-1.** ADS (Ambient Diffuse Specular) shading in OpenGL ES 2.0*

OpenGL ES 2.0 is derived from the larger OpenGL 2.0 API, the programmable pipeline for rendering 3D graphics on desktops. ES 2.0 is a suitable subset of OpenGL, optimized for resource constrained display devices, such as mobiles, tablets, and gaming consoles. ES 2.0 contains only the most useful methods from OpenGL 2.0 API, with redundant techniques removed. This allows OpenGL ES 2.0 on handheld devices to deliver rich game content like its parent API.

Devices Love It

As of October 1, 2012, more than 90% of all Android devices were running version 2.0 of OpenGL ES. Devices running version 2.0 are also capable of emulating version 1.1. However, an activity in Android cannot use both versions together, stemming from the fact that OpenGL ES 2.0 API is not backwards compatible with ES 1.x. Note that, although an *activity* cannot use both versions together, an *application* can still use them together. (Information about OpenGL ES version distribution across Android devices is available at http://developer.android.com/about/dashboards/index.html, and Figure 1–2 shows a chart representing that distribution.)

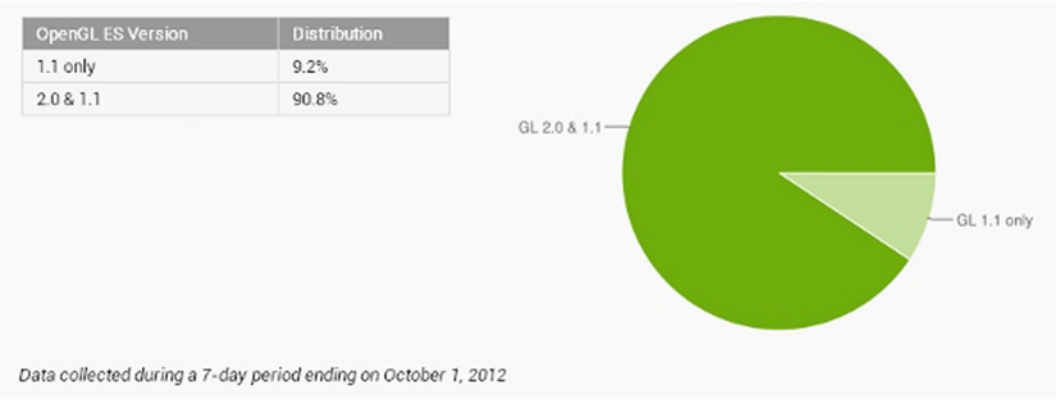

***Figure 1-2.** OpenGL ES version distribution*

> **Note** To demonstrate the use of both ES 1.x and ES 2.0 APIs in an application, the *GLES ACTIVITY* application is provided in the source code for this chapter. This application contains activities `Main` and `Second`. The `Main` activity uses ES 1.x, whereas the `Second` activity uses ES 2.0. To load this application into your Eclipse workspace, under "File Menu," select "Import," and then import the archive file `glesactivity.zip` from the Chapter1 folder.

OpenGL ES 2.0 constitutes such a huge share of distribution (Figure 1-2), because of widespread support from leading CPU and GPU manufacturing industries. (A complete list of companies with their conformant ES 1.x/2.0 products can be found at http://www.khronos.org/conformance/adopters/conformant-products#opengles.) The following vendors have actively participated in consolidating support for OpenGL ES 2.0 on Android since 2010:

(Leading GPU manufacturers)

- NVIDIA
- AMD
- Imagination Technologies

(Leading CPU manufacturers)

- ARM
- Texas Instruments
- STMicroelectronics

Implementer companies make use of the *Khronos developed technologies* at no cost in license fees. However, they do not claim that a product is "compliant," unless the technologies enter and pass conformance testing. The following are the implementers of OpenGL ES 2.0 for various embedded devices:

- Intel
- Marvell
- NVIDIA
- Creative Technology Ltd.
- QUALCOMM
- MediaTek Inc.
- Apple, Inc.
- NOKIA OYJ
- Digital Media Professionals
- Panasonic

Note Although most embedded platforms are up and running with OpenGL ES 2.0, the Khronos Group announced on August 6th, 2012, the release of the OpenGL ES 3.0 specification, bringing significant functionality and portability enhancements to OpenGL ES API. OpenGL ES 3.0 is backwards compatible with OpenGL ES 2.0, enabling applications to incrementally add new visual features to applications. The full specification and reference materials are available for immediate download at http://www.khronos.org/registry/gles/.

Easy App Development: Let's Create an OpenGL Surface View

ES 2.0 applications can be easily developed for Android devices using the Android SDK. The best part about creating such applications using this SDK is that there is no need for any external library (something that can be quite burdensome for new ES 2.0 application developers on iPhone).

There is another way to create Android ES 2.0 applications—using the Android *Native Development Kit* (*NDK*). In some cases, NDK can make ES 2.0 applications faster than those made using SDK. NDK lets users code in native languages, such as C and C++. This makes it possible to use popular libraries written using C/C++, but only at the cost of increased complexity. Beginner ES 2.0 application

developers may find this difficult to deal with, which can ultimately make NDK counter-productive. NDK is typically a tool for advanced Android developers, but be assured the performance gap between most ES 2.0 applications created using SDK and NDK is becoming negligible.

> **Note** Do not use NDK simply because you like coding your applications in C/C++; use it only for cases in which performance is critical to your application. Also, remember that Dalvik VM is becoming faster, reducing the performance gap between SDK and NDK.

Determining OpenGL ES Version

To demonstrate the ease of developing ES 2.0 applications for Android devices, a quick example is given here for creating an OpenGL surface view. This view is different from the XML view (UI layout) you have generally created for most Android applications. (Chapter 3 contains a detailed account of OpenGL surface view.)

Before I discuss this example, you need to determine the version of OpenGL ES on your Android device. To do so, let's create a blank `Activity`:

1. In the Eclipse toolbar, click the icon to open wizard to create a new Android project.
2. Uncheck the "Create custom launcher icon" option, and click "Next," as shown in Figure 1-3.

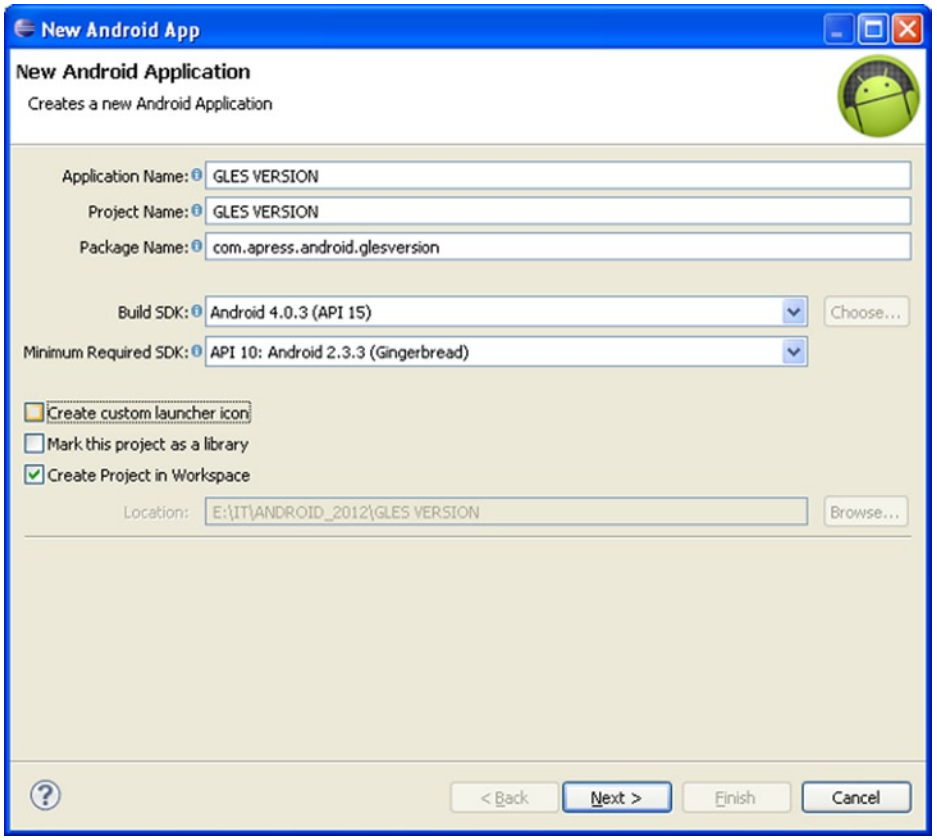

Figure 1-3. *Creating a new Android application*

> **Note** You might be accustomed to an older version of the SDK. The older version lacked some tools present in the newer version. Make sure you have these tools installed using your SDK Manager. If you prefer working offline, always allow time to update the SDK.

3. For "Create Activity," select BlankActivity and click "Next." Select MasterDetailFlow (Figure 1-4) only if you are experienced in developing applications for tablets. This book only addresses BlankActivity, because we are not developing for tablets.

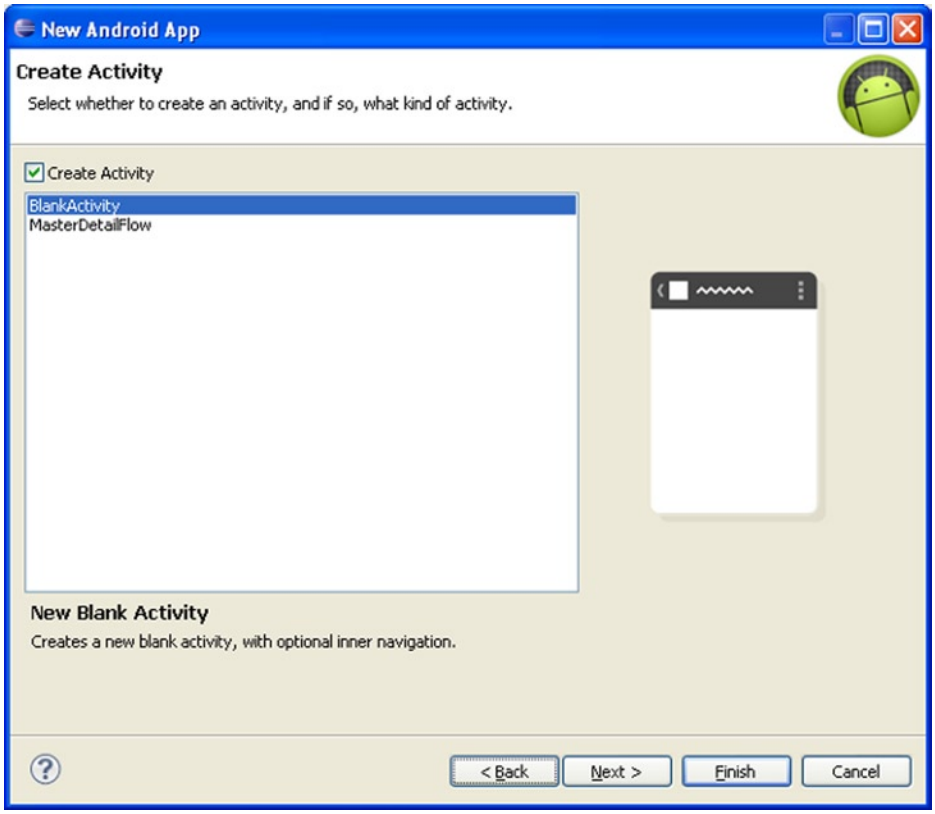

Figure 1-4. Selecting the type of Activity

4. Set the “Activity Name” and “Layout Name” as “Main” and “main,” respectively (Figure 1-5). In cases in which the Android application has only one activity, most coders name the Java file `Main.java`.

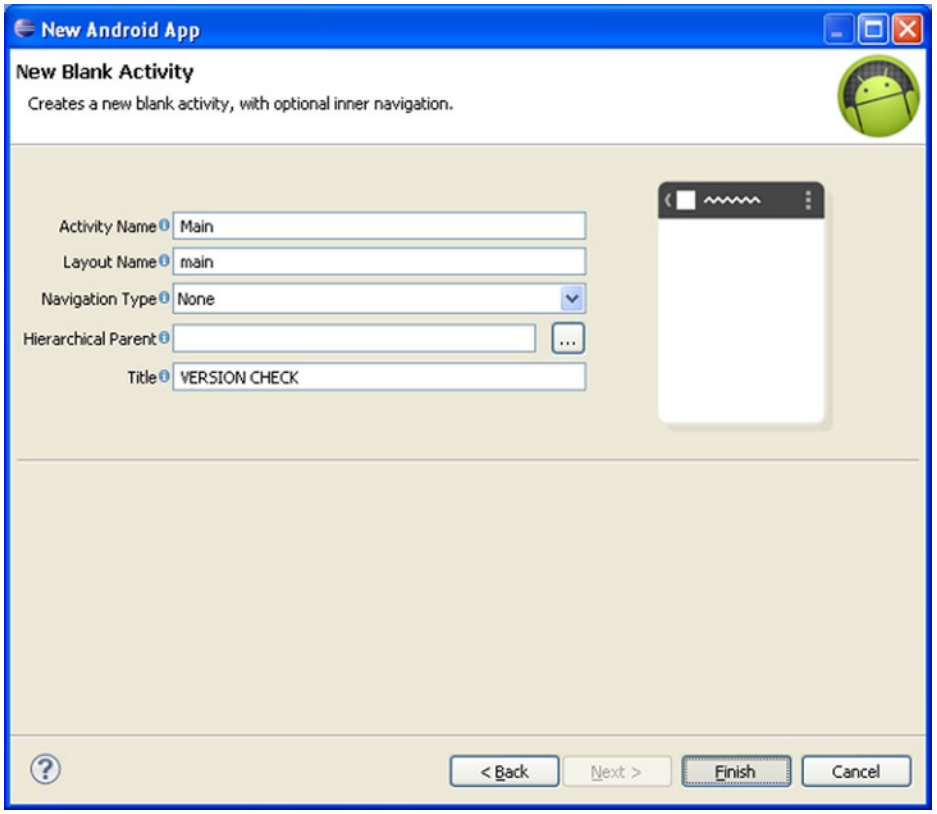

Figure 1-5. *Creating a new blank Activity*

5. Click “Finish” if you have already installed the “Android Support Library.” If you haven’t installed it, then click “Install/Update,” wait until it is installed, and then click “Finish” (please note that you might not get the option to install “Android Support Library” if using an older version of the ADT plugin).

After the blank `Activity` (`Main.java`) is created, SDK will show warnings for unused imports, as shown in Figure 1-6. To remove these warnings:

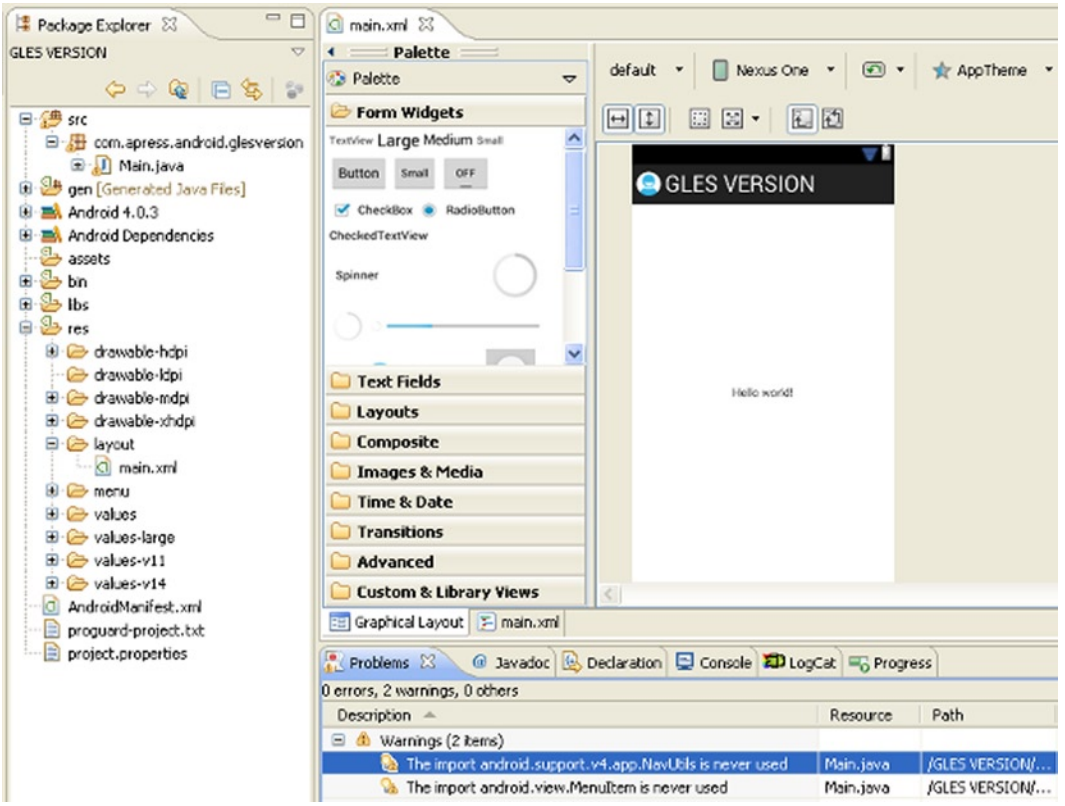

***Figure 1-6.** Project warnings*

1. In the *Problems* view, click the small plus-sign (+) button near “Warnings” and the list of warnings will be displayed.
2. Double click any warning. SDK will move the edit cursor to the line containing the warning.
3. Now, press *Ctrl* and *1* on the keyboard. SDK will then suggest ways to remove the warning(s).
4. Select the “Organize imports” (Figure 1-7) option, and the warnings will be removed.

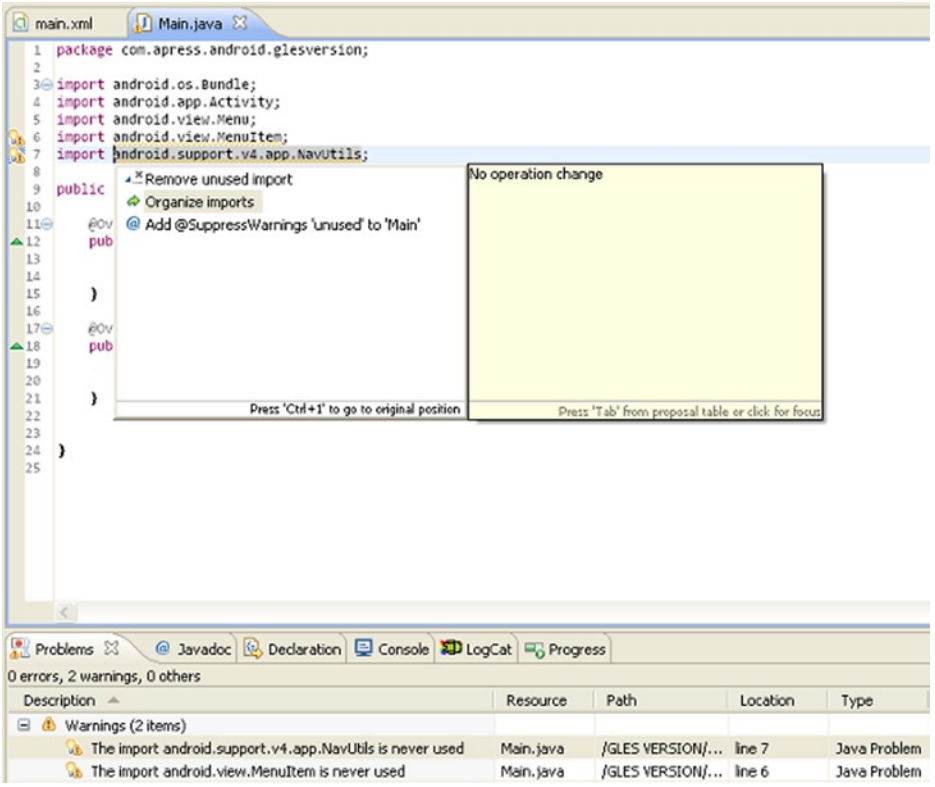

***Figure 1-7.** Organizing imports*

5. If warnings persist, clean the project by selecting the "Clean" option under "Project Menu" in Eclipse, as shown in Figure 1-8. Remember this step, because Eclipse might not update the project binaries after modification(s). Cleaning will update/refresh them.

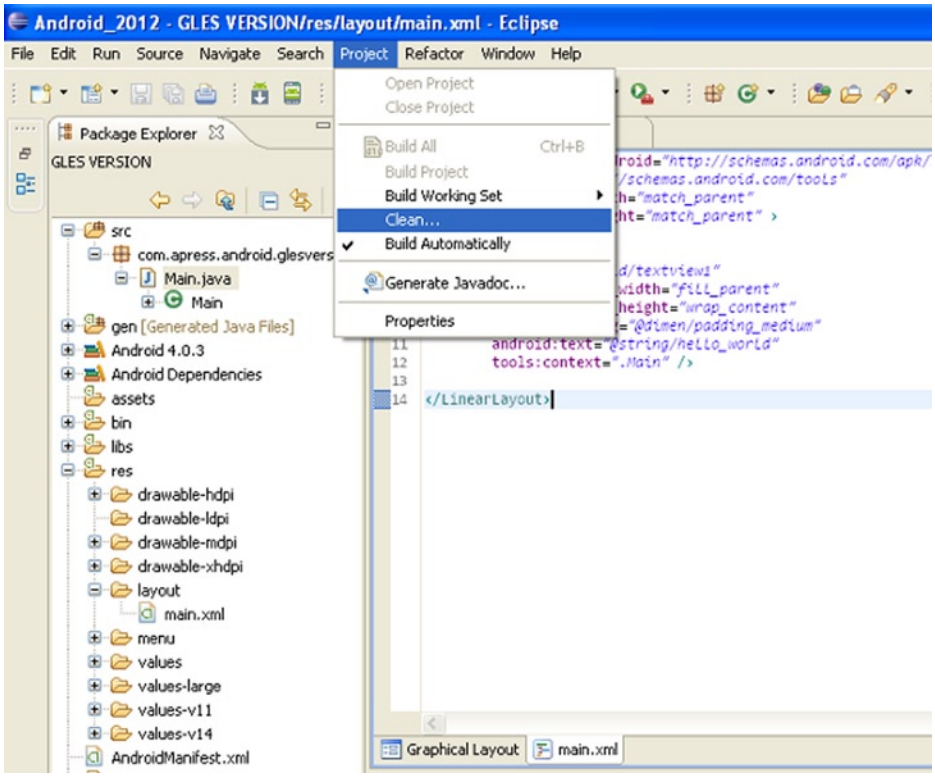

Figure 1-8. *Cleaning our project*

> **Note** Although it is not necessary to remove all the warnings from your application (because the application can still work with these warnings), get into the habit of clearing them, especially in cases in which unused imports or other redundant code can cause your application to be larger than necessary.
>
> The few lines that cause warnings may look insignificant now; however, later in the book, we will be dealing with examples in which those lines might add up to bloat the performance of your application. The Android *lint* tool always highlights such warnings and, in some cases, can optimize the binaries by itself. This does not happen always, however, so remember to clear those warnings.

After warnings have been removed, replace the entire (XML) UI layout in your project's `res/layout/main.xml` with the contents of Listing 1-1. Notice the main difference between Listing 1-1 and the default UI layout (of the blank `Activity` template) is the root tag `RelativeLayout`.

Listing 1-1. GLES VERSION/res/layout/main.xml

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <TextView
        android:id="@+id/textview1"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:padding="@dimen/padding_medium"
        android:text="@string/hello_world"
        tools:context=".Main" />

</LinearLayout>
```

Listing 1-1 places a `TextView` on the screen. This `TextView` is as wide as the screen in any orientation and has an `id` of "textview1." Additionally, its padding-dimensions and text are defined in the `dimens.xml` and `strings.xml` files, respectively, inside this project's *res/values* folder.

Now, replace the `onCreate` method of the blank `Activity` (`Main.java`) with the `onCreate` method from Listing 1-2.

Listing 1-2. GLES VERSION/src/com/apress/android/glesversion/Main.java

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

  final ActivityManager activityManager = (ActivityManager) getSystemService(Context.ACTIVITY_SERVICE);
  final ConfigurationInfo configurationInfo = activityManager.getDeviceConfigurationInfo();
  final boolean supportsEs2 = configurationInfo.reqGlEsVersion >= 0x20000;

  TextView tv = (TextView) findViewById(R.id.textview1);
  if (supportsEs2) {
    tv.setText("es2 is supported");
  } else {
    tv.setText("es2 is not supported");
  }
}
```

In the `onCreate` method (Listing 1-2), we obtain the device configuration attributes and use them to detect the version of OpenGL ES running on the device. Next, we find the `TextView` in the UI layout of our application by its `id` ("textview1") and use it to display the result using its `setText` method.

Now the application is ready for use. However, before running this application on a real device, we will test it on the Android Emulator. If you haven't created a virtual device yet, start the AVD Manager and complete the following steps:

1. Click "New" to open the window to create a new virtual device.
2. Name this virtual device "IceCreamSandwich". We are targeting (at least) the Ice Cream Sandwich emulator, so we will name it IceCreamSandwich. You may also modify this name to indicate the resolution of virtual device.
3. Under target, select API level 15, as shown in Figure 1-9.

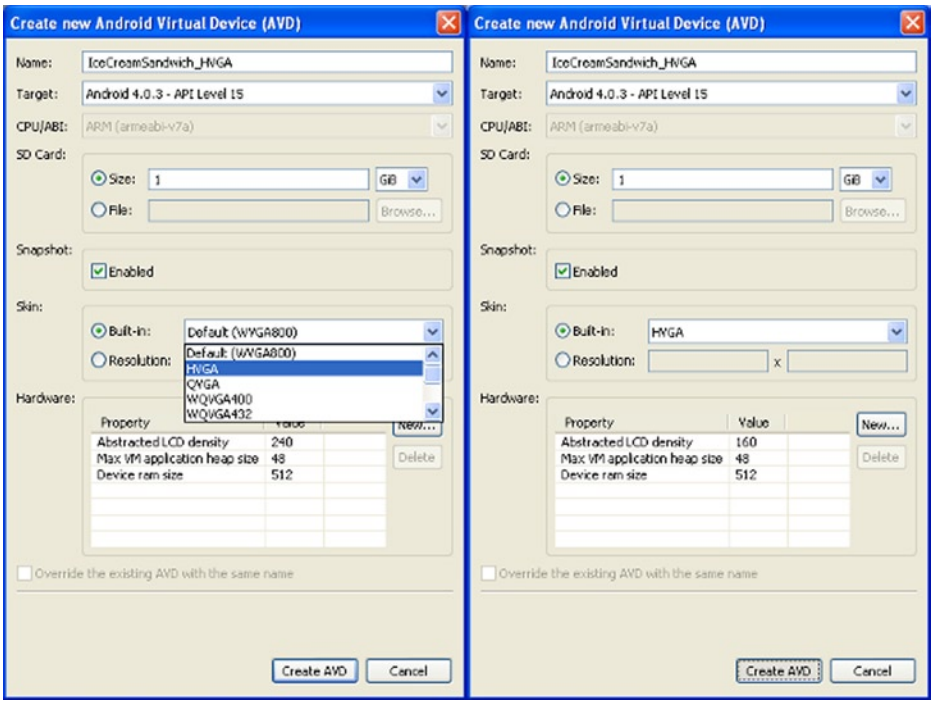

***Figure 1-9.** Using AVD Manager*

4. Enter the size for the SD card.
5. Enable "Snapshot" to avoid going through the Android bootup sequence every time you start the virtual device.
6. To create this virtual device at a specific resolution, select a built-in skin.
7. Click "Create AVD" to create the virtual device.

AVD Manager will take some time to prepare the virtual device. After the device is successfully created, it will be listed in the AVD Manager with a green tick at the beginning. Select the created virtual device and click "Start."

Let the device boot. With *Snapshot* enabled, the device will start from where it left off the next time. When the Home screen is visible in the virtual device (Figure 1-10), return to Eclipse and run the application.

Figure 1-10. *IceCreamSandwich on Android Emulator*

As of January 2013, Android Emulator supported ES 1.x only (some hosts allow Emulators to access their GPU for ES 2.0, but, for most, Android Emulator supports ES 1.x only—Figure 1-11).