Design, create, and optimize your graphics
for use in your Android apps

Pro Android Graphics

Wallace Jackson

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

Android Digital Imaging: Formats, Concepts, and Optimization

In this first chapter, you will see how digital imaging is implemented inside of the Android operating system. We will take a look at the digital image formats that are supported in Android, the classes that allow imagery to be formatted on the screen, and the basic digital imaging concepts that you will need to understand to follow what we will be doing in Android graphics design.

We will also take a look at how to optimize your digital image assets for Android application development. We'll explore digital image optimization both from an individual image asset data footprint standpoint and from an Android device type market coverage standpoint.

As you know, Android devices are no longer just smartphones, but run the gamut from watches to phones to tablets to game consoles to 4K iTV sets. The significance of this to the graphic design aspect of Android application development is that you must now create your digital image assets in a far wider range of pixels, from as low a resolution as 240 pixels to as high a resolution as 4096 pixels, and you must do this for each of your digital imaging assets.

We will look at the facilities that Android has in place for doing this as a part of your application development workflow and your asset referencing **XML** (eXtensible Markup Language) markup. Markup is different from Java code in that it uses "tags" much as **HTML** (HyperText Markup Language) does. XML is very similar to HTML in that it uses these tag structures, but different in that it is customizable, which is why Google has selected it for use in Android OS.

Since this is a Pro level book I assume that you have a decent level of experience with developing for the Android platform, and have already been through the learning process in an Android educational title such as my *Learn Android App Development* (Apress, 2013).

Let's get started by taking a look at what image formats Android supports.

Android's Digital Image Formats: Lossless Versus Lossy

Android supports several popular digital imagery formats, some which have been around for decades, such as the Compuserve GIF (Graphic Information Format) and the Joint Photographic Experts Group (JPEG) format, and some are more recent, such as the PNG (Portable Network Graphics) and WebP (developed by ON2 and acquired and made open source by Google).

I will talk about these in their order of origin, from the oldest (and thus least desirable) GIF through the newest (and thus most advanced) WebP format. **Compuserve GIF** is fully supported by the Android OS, but is not recommended. GIF is a **lossless** digital image file format, as it does not throw away image data to achieve better compression results.

This is because the GIF compression algorithm is not as refined (read: powerful) as PNG, and it only supports **indexed** color, which you will be learning about in detail in this chapter. That said, if all your image assets are already created and in GIF format, you will be able to use them with no problem (other than the mediocre resulting quality) in your Android applications.

The next oldest digital image file format that Android supports is **JPEG**, which uses a **truecolor** color depth instead of an indexed color depth. We will be covering color theory and color depth soon.

JPEG is said to be a **lossy** digital image file format, as it throws away (loses) image data in order to achieve smaller file sizes. It is important to note that this original image data is **unrecoverable after compression** has taken place, so make sure to save an original uncompressed image file.

If you zoom into a JPEG image after compression, you will see a discolored area effect that clearly was not present in the original imagery. These degraded areas in the image data are termed **compression artifacts** in the digital imaging industry and will only occur in lossy image compression.

The most recommended image format in Android is called the **PNG** (Portable Network Graphic) file format. PNG has both an indexed color version, called **PNG8**, and a truecolor version, called **PNG24**. The PNG8 and PNG24 extensions represent the **bit depth** of color support, which we will be getting into in a little bit. PNG is pronounced "ping" in the digital image industry.

PNG is the recommended format to use for Android because it has decent compression, and because it is lossless, and thus has both high image quality and a reasonable level of compression efficiency.

The most recent image format was added to Android when Google acquired ON2 and is called the **WebP** image format. The format is supported under Android 2.3.7 for image read, or playback, support and in Android 4.0 or later for image write, or file saving, support. WebP is a static (image) version of the WebM video file format, also known in the industry as the **VP8** codec. You will be learning all about **codecs** and **compression** in a later section.

Android View and ViewGroup Classes: Image Containers

Everything in this section is just a review of Android Java class concepts and constructs, which you, as an intermediate level Android programmer, probably understand already. Android OS has a class that is dedicated to displaying digital imagery and digital video called the **View** class. The View class is subclassed directly from the **java.lang.Object** class; it is designed to hold

imagery and video, and to format it for display within your user interface screen designs. If you wish to review what the View class can do, visit the following URL:

http://developer.android.com/reference/android/view/View.html

All user interface elements are based on (subclassed from) the View class, and are called **widgets** and have their own package called **android.widget**, as most developers know. If you are not that familiar with Views and Widgets, you might consider going through the *Learn Android App Development* book before embarking on this one. If you wish to review what Android Widgets can do, visit the following URL:

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

The **ViewGroup** class is also subclassed from the View class. It is used to provide developers with the user interface element container that they can use to design their screen layout and organize their user interface widget View objects. If you wish to review the various types of Android ViewGroup Screen Layout Container classes, visit the following URL:

http://developer.android.com/reference/android/view/ViewGroup.html

Views, ViewGroups and widgets in Android are usually defined using XML. This is set up this way so that designers can work right alongside the coders in the application development, as XML is far easier to code in than Java is.

In fact, XML isn't really programming code at all; it's markup, and, just like HTML5, it uses tags, nested tags, and tag parameters to build constructs that are later used in your Android application.

Not only is XML utilized in Android to create user interface screen design but also menu structures, string constants, and to define your application version, components, and permissions inside the AndroidManifest.xml file.

The process of turning your XML data structures into Java-code–compatible objects that can be used with your Android application Java components is called **inflating** XML markup, and Android has a number of inflater classes that perform this function, usually in component startup methods, such as the onCreate() method. You will see this in some detail throughout the Java coding examples in this book, as it bridges our XML markup and Java code.

The Foundation of Digital Images: Pixels and Aspect Ratio

Digital images are made up of two-dimensional arrays of pixels, which is short for picture (pix) elements (els). The number of pixels in an image is expressed by its **resolution**, which is the number of pixels in both the Height (H) and Width (W) dimensions.

To find the number of pixels in an image, simply multiply the Width pixels by the Height pixels. For instance, an HDTV 1920 x 1080 image will contain 2,073,600 pixels, or slightly more than 2 million pixels. Two million pixels could also be referred to as two megapixels.

The more pixels that are in an image, the higher its resolution; just like with digital cameras, the more megapixels are in the data bank, the higher the quality level that can be achieved. Android supports everything from low resolution 320 x 240 pixel display screens (Android Watches and

smaller flip-phones) to medium resolution 854 x 480 pixel display screens (mini-tablets and smartphones), up to high resolution 1280 x 720 pixel display screens (HD smartphones and mid-level tablets), and extra high resolution 1920 x 1080 pixel display screens (large tablets and iTV sets). Android 4.3 adds support for 4K resolution iTVs, which feature 4096 by 2160 resolution.

A slightly more complicated aspect (no pun intended) of image resolution is the image **aspect ratio**, a concept that also applies to display screens. This is the ratio of width to height, or **W:H**, and will define how square or rectangular (popularly termed widescreen) an image or a display screen is.

A 1:1 aspect ratio display (or image) is perfectly square, as is a 2:2 or a 3:3 aspect ratio image. You see, it is the ratio between the two numbers that defines the shape of the image or screen, not the numbers themselves. An example of an Android device that has a 1:1 square aspect ratio would be an Android SmartWatch.

Most Android screens are **HDTV** aspect ratio, which is **16:9**, but some are a little less wide, as in 16:10 (or 8:5 if you prefer). Wider screens will also surely appear, so look for 16:8 (or 2:1, if you prefer) ultra-wide screens that have a 2160 by 1080 resolution LCD or LED display.

The aspect ratio is usually expressed as the smallest pair of numbers that can be achieved (reached) on either side of the aspect ratio colon. If you paid attention in high school when you were learning about lowest common denominators, then this aspect ratio should be fairly easy to calculate.

I usually do this by continuing to divide each side by two. So, taking the fairly odd-ball 1280 x 1024 SXGA resolution as an example, half of 1280 x 1024 is 640 x 512, and half of that is 320 x 256, half of that is 160 x 128, half of that is 80 x 64, half of that is 40 x 32, half of that is 20 x 16, half of that is 10 x 8, and half of that is 5 x 4, so an SXGA screen is a 5:4 aspect ratio.

Original PC screens primarily offered a 4:3 aspect ratio; early CRT tube TVs were nearly square, featuring a 3:2 aspect ratio. The current market trend is certainly towards wider screens and higher resolution displays; however, the new Android Watches may change that back towards square aspect ratios.

The Color of Digital Images: Color Theory and Color Depth

Now you know about digital image pixels and how they are arranged into 2D rectangular arrays at a specific aspect ratio defining the rectangular shape. So the next logical aspect (again, no pun intended) to look into is how each of those pixels gain their **color values**.

Color values for image pixels are defined by the amount of three different colors, **red**, **green** and **blue** (**RGB**), which are present in varying amounts in each pixel. Android display screens utilize **additive color**, which is where the wavelengths of light for each RGB color plane are summed together in order to create millions of different color values.

Additive color, which is utilized in LCD or LED displays, is the opposite of subtractive color, which is utilized in print. To show the difference, under a subtractive color model, mixing red with green (inks) will yield a purplish color, whereas in an additive color model, mixing red with green (light) creates a bright yellow color result.

There are **256 levels** of each RGB color for each pixel, or **8-bits** of color intensity variation, for each of these red, green, and blue values, from a minimum of zero (off, no color contributed) to a maximum of 255 (fully on, maximum color contributed). The number of bits used to represent color in a digital image is referred to as the **color depth** of that image.

There are several common color depths used in the digital imaging industry, and I will outline the most common ones here along with their formats. The lowest color depth exists in an **8-bit indexed color** image, which has **256** color values, and uses the GIF and PNG8 image formats to contain this indexed color type of digital image data.

A medium color depth image features a **16-bit** color depth and thus contains **65,536** colors (calculated as 256 x 256); it is supported in the TARGA (TGA) and Tagged Image File Format (TIFF) digital image formats.

Note that Android does not support any of the 16-bit color depth digital image file formats (TGA or TIFF), which I think is an omission, as 16-bit color depth support would greatly enhance a developer image data footprint optimization, a subject which we will be covering later on in the chapter.

A high color depth image features a **24-bit** color depth and thus contains over 16 million colors. This is calculated as 256 x 256 x 256 and equals **16,777,216** colors. File formats supporting 24-bit color include JPEG (or JPG), PNG, TGA, TIFF and WebP.

Using 24-bit color depth will give you the highest quality level, which is why Android prefers the use of a PNG24 or a JPEG image file format. Since PNG24 is lossless, it has the highest quality compression (lowest original data loss) along with the highest quality color depth, and so PNG24 is the preferred digital image format to use, as it produces the highest quality.

Representing Colors in Android: Hexadecimal Notation

So now that you know what color depth is, and that color is represented as a combination of three different red, green, and blue **color channels** within any given image, we need to look at how we are to represent these three RGB color channel values.

It is also important to note that in Android, color is not only used in 2D digital imagery, also called **bitmap** imagery, but also in 2D illustrations, commonly known as **vector** imagery, as well as in **color settings**, such as the background color for a user interface screen or text color.

In Android, different levels of RGB color intensity values are represented using **hexadecimal notation**, the **Base 16** computer notation invented decades ago to represent 16 bits of data value. Unlike Base 10, which counts from zero through 9, Base 16 counts from zero through F, where F represents a Base 10 value of 15 (or if you are a programmer you could count from 0–15, which also gives 16 decimal data values, either way you prefer to look at it). See Table 1-1 for some examples.

***Table 1-1.** Hexadecimal Values and Corresponding Decimal Values*

Hexadecimal Values:	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Decimal Values:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

A hexadecimal value in Android always starts with a **pound sign**, like this: **#FFFFFF**. This hexadecimal data color value represents a color of white. As each slot in this 24-bit hexadecimal representation represents one Base 16 value, to get the 256 values you need for each RGB color will take 2 slots, as 16 x 16 equals 256. Thus for a 24-bit image you need six slots after the pound sign to hold each of the six hexadecimal data values.

The hexadecimal data slots represent the RGB values in a following format: **#RRGGBB**. So, for the color white, all red, green, and blue **channels** in this hexadecimal color data value representation are at the maximum **luminosity**.

If you additively sum all of these colors together you'll get white light. As mentioned, the color **yellow** is represented by the red and green channels being on and the blue channel being off, so the hexadecimal representation is **#FFFF00**, where both red and green channel slots are on (FF or 255), and blue channel slots are fully off (00 or a zero value).

It is important to note here that there is also a **32-bit** image color depth whose data values are represented using an **ARGB** color channel model, where the **A** stands for **alpha**, which is short for **alpha channel**. I will be going over the concept of alpha and alpha channels, as well as pixel blending, in great detail in the next section of this chapter.

The hexadecimal data slots for an **ARGB** value hold data in the following format: **#AARRGGBB**. So for the color white, all alpha, red, green, and blue channels in this hexadecimal color data value representation are at a maximum luminosity (or opacity), and the alpha channel is fully opaque, as represented by an FF value, so its hexadecimal value is **#FFFFFFFF**.

A 100% transparent alpha channel is represented by setting the alpha slots to zero; thus, a fully transparent image pixel is **#00FFFFFF**, or **#00000000**. If an alpha channel is transparent, color value doesn't matter!

Image Compositing: Alpha Channels and Blending Modes

In this section we will take a look at **compositing** digital images. This is the process of **blending** together more than one **layer** of a digital image in order to obtain a resulting image on the display that appears as though it is one final image, but which in fact is actually a collection of more than one seamlessly composited image **layers**.

To accomplish this, we need to have an **alpha channel** (transparency) value that we can utilize to precisely control the blending of that pixel with the pixel (in that same location) on the other layers above and below it.

Like the other RGB channels, the alpha channel also has 256 levels of transparency as represented by two slots in the hexadecimal representation for the ARGB data value, which has eight slots (32-bits) of data rather than the six slots used in a 24-bit image, which can be thought of as a 32-bit image with no alpha channel data.

Indeed, if there's no alpha channel data, why waste another 8 bits of data storage, even if it's filled with F's (or fully opaque pixel values, which essentially equate to unused alpha transparency values). So a 24-bit image has no alpha channel and is not going to be used for compositing, for instance the bottom plate in a compositing layer stack, whereas a 32-bit image is going to be used as a compositing layer on top of something else that will need the ability to show through (via transparency values) in some of the pixel locations in the image composite.

How does having an alpha channel and using image compositing factor into Android graphics design, you may be wondering. The primary advantage is the ability to split what looks like one single image into a number of component layers. The reason for doing this is to be able to apply Java programming logic to individual layer elements in order to control parts of your image that you could not otherwise control were it just one single 24-bit image.

There is another part of image compositing called **blending modes** that also factors heavily in professional image compositing capabilities. Any of you familiar with Photoshop or GIMP know that each layer can be set to use different blending modes that specify how the pixels for that layer are blended (mathematically) with the previous layers (underneath that layer). Add this mathematical pixel blending to the 256 level transparency control and you can achieve any compositing effect or result that you can imagine.

Blending modes are implemented in Android using the **PorterDuff** class, and give Android developers most of the same compositing modes that Photoshop or GIMP afford to digital imaging artisans. This makes Android a powerful image compositing engine just like Photoshop is, only controllable at a fine level using custom Java code. Some of Android's PorterDuff blending modes include ADD, SCREEN, OVERLAY, DARKEN, XOR, LIGHTEN, and MULTIPLY.

Digital Image Masking: A Popular Use for Alpha Channels

One of the primary applications for alpha channels is to **mask** out areas of an image for compositing. **Masking** is the process of cutting subject matter out of an image and placing it onto its own layer using an alpha channel.

This allows us to put image elements or subject material into use in other images, or even in animation, or to use it in special effects applications. Digital image software packages such as Photoshop and GIMP have many tools and features that are specifically there for use in masking and then image compositing. You can't really do effective image compositing without doing masking first, so it's an important area for graphics designers to master.

The art of masking has been around for a very long time. In fact, if you are familiar with the bluescreen and greenscreen backdrop that the weather forecasters use to seem like they are standing in front of the weather map (when they are really just in front of a green screen), then you recognize that masking techniques exist not only for digital imaging, but also for digital video and film production.

Masking can be done for you automatically using bluescreen or greenscreen backdrops and computer software that can automatically extract those exact color values in order to create a mask and an alpha channel (transparency), and this can also be done manually (by hand) in digital imaging by using the **selection** tools and **sharpening** and **blur** algorithms.

You'll learn a lot about this work process during this book, using popular open source software packages such as GIMP 2 and EditShare Lightworks 11. GIMP 2.8 is a digital image compositing software tool, and Lightworks 11 is a digital video editing software tool. You will also be using other types of tools, such as video compression software, during the book to get a feel for the wide range of software tools external to Android that need to be incorporated into the work process for Android graphics design.

Digital image compositing is a very complex and involved process, and thus it must span a number of chapters. The most important consideration in the masking process is getting smooth, sharp edges around your masked object, so that when you drop it onto a new background image it looks

as though it was photographed there in the first place. The key to this is in the selection work process, and using digital image software **selection tools** (there are a half-dozen of these, at least) in the proper way (work process) and in the proper usage scenarios.

For instance, if there are areas of **uniform color** around the object that you wish to mask (maybe you shot it against a bluescreen or greenscreen), you can use the **magic wand tool** and a proper **threshold** setting to select everything except the object, and then **invert** that **selection set** in order to obtain a selection set containing the object. Sometimes the correct way to approach something is in reverse, as you will see later in the book.

Other selection tools contain complex algorithms that can look at color changes between pixels, which can be very useful in **edge detection**. You can use edge detection in other types of selection tools, such as the Scissor Tool in GIMP 2.8.6, which allow you to drag your cursor along the edge of an object that you wish to mask while the tool's algorithm lays down a precise pixel-perfect placement of a selection edge, which you can later edit using control points.

Smoothing Edges in a Mask: The Concept of Anti-Aliasing

Anti-aliasing is a technique where two adjacent colors in an image which are on an edge between two colors are blended right on the edge to make the edge look smoother when the image is zoomed out. What this does is to trick the eye into seeing a smoother edge and gets rid of what is commonly called "the jaggies." Anti-aliasing provides very impressive results by using averaged color values of a few pixels along any edge that needs to be made smoother (by averaged I mean some color or spectrum of colors that is part of the way between the two colors that are colliding at a jagged edge in an image).

I created a simple example of this technique to show you visually what I mean. In Figure 1-1, you will see that I created a seemingly smooth red circle on a bright yellow background. I then zoomed into the edge of that circle and took a screenshot and placed it alongside of the zoomed out circle to show the anti-aliasing (orange) values of a color between (or made from) the red and yellow colors that border each other at the edge of the circle.

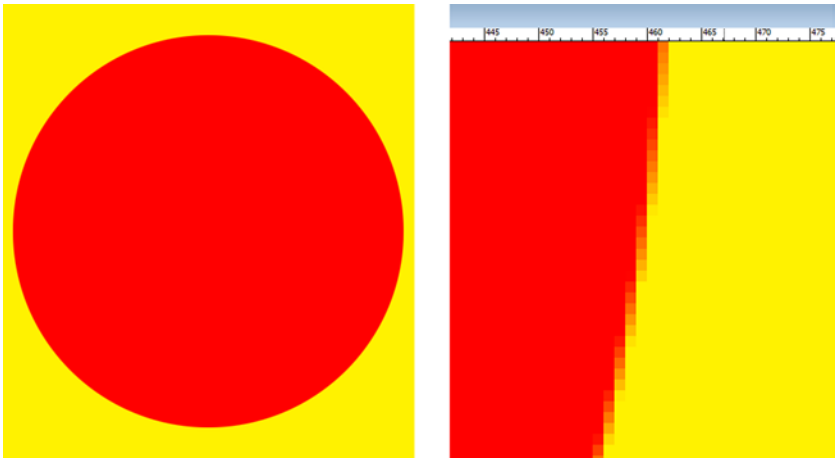


Figure 1-1. *A red circle on a yellow background (left) and a zoomed-in view (right) showing the anti-aliasing*

We will be looking at anti-aliasing in detail during the book. However, I wanted to cover all the key image concepts all in one place, and all in context, to provide a baseline knowledge foundation for you upfront. Hope you don't mind an initial chapter on theory before we start coding.

Optimizing Digital Images: Compression and Dithering

There are a number of factors that affect **image compression**, and some techniques that can be used to get a better quality result with a smaller **data footprint**. This is the objective in optimizing digital imagery: to get the smallest data footprint with the highest quality visual result.

We will start with the aspects that most affect the data footprint and examine how each of them contributes to data footprint optimization of any given digital image. Interestingly, these are similar to the order of the digital imaging concepts that we have covered so far in this chapter.

The single most critical contributor to the resulting file size, or data footprint, is the **number of pixels**, or the resolution, of a digital image. This is logical because each of these pixels need to be stored along with the color values for each of their channels. Thus, the smaller you can get your image resolution (while still having it look sharp), the smaller its resulting file size will be. This is what we call a "no brainer."

Raw (uncompressed) image size is calculated by **Width x Height x 3** for 24-bit RBG images, or possibly **Width x Height x 4** for a 32-bit ARGB image. Thus, an uncompressed truecolor 24-bit VGA image will have 640 x 480 x 3, equaling **921,600** bytes of original uncompressed data. If you divide 921,600 by 1024 (bytes in a kilobyte), you get the number of **Kilobytes** that is in a raw VGA image, and that number is an even **900KB**.

As you can see, image color depth is the next most critical contributor to the data footprint of an image, because the number of pixels in that image is multiplied by 1 (8-bit) or 2 (16-bit) or 3 (24-bit) or 4 (32-bit) color data channels. This is one of the reasons that indexed color (8-bit) images are still widely used, especially using the **PNG8** image format, which features a superior lossless compression algorithm than the GIF format utilizes.

Indexed color images can simulate truecolor images, if the colors that are used to make up the image do not vary too widely. Indexed color images use only 8 bits of data (256 colors) to define the image pixel colors, using a **palette** of 256 optimally selected colors rather than 3 RGB color channels.

Depending on how many colors are used in any given image, using only 256 colors to represent an image versus 16,777,216 can cause an effect called **banding**, where transfers between adjoining colors are not smooth. Indexed color images have an option to correct for this visually called **dithering**.

Dithering is a process of creating dot patterns along the edges of two adjoining colors in an image in order to trick the eye into thinking there is a third color used. This gives us a maximum perceptual amount of colors of 65,536 colors (256 x 256) but only if each of those 256 colors borders on each of the other 256 colors. Still, you can see the potential for creating additional colors, and you will be amazed at the results an indexed color image can achieve in some scenarios (with certain images).

Let's take a truecolor image, such as the one shown in Figure 1-2, and save it as a PNG8 indexed color image to show the dithering effect. We will take a look at the dithering effect on the driver's side rear fender on the Audi 3D image, as it contains a gradient of gray color.

Figure 1-2. A truecolor image source that uses 16.8 million colors, which we are going to optimize to PNG8 format

We will set the PNG8 image, shown in Figure 1-3, to use 5-bit color (32 colors) so that we can see the dithering effect clearly. As you can see, dot patterns are made between adjacent colors to create additional colors.

Figure 1-3. Showing the effect of dithering with an indexed color image compression setting of 32 colors (5-bit)

It is interesting to note that less than 256 colors can be used in an 8-bit indexed color image. This is done to reduce the data footprint; for instance, an image that can attain good results using only 32 colors is actually a 5-bit image and is technically a PNG5, even though the format is called PNG8.

Also notice that you can set the percentage of dithering used; I usually select either the 0% or 100% setting, but you can fine-tune the dithering effect anywhere between these two extreme values. You can also choose a dithering algorithm type; I use diffusion dithering, as it yields a smooth effect along irregularly shaped gradients such as those on the car fender.

Dithering, as you may imagine, adds data (patterns) that is more difficult to compress, and thus, it increases the data footprint by a few percentage points. Be sure to check the resulting filesize with and without dithering applied to see if it is worth the improved visual results that it affords.

The final concept (that you have learned about so far) that can increase the data footprint of the image is the alpha channel, as adding an alpha adds another 8-bit color channel (transparency) to the image being compressed.

However, if you need an alpha channel to define transparency in order to support future compositing needs with that image, there is not much choice but to include the alpha channel data. Just make sure not to use a 32-bit image format to contain a 24-bit image that has an empty (all zeroes, and completely transparent, and thus empty of alpha value data) alpha channel.

Finally, many alpha channels that are used to mask objects in an image will compress very well, as they are largely areas of white (opaque) and black (transparent) with some grey values along the edge between the two colors to **anti-alias** the mask. As a result, they provide a visually smooth edge transition between the object and the imagery used behind it.

Since in an alpha channel image mask the 8-bit transparency gradient from white to black defines transparency, the grey values on the edges of each object in the mask essentially average the colors of the object and its target background, which provides real-time anti-aliasing with any target background used.

Now it's time to get Android installed on your workstation, and then you can start developing graphics-oriented Android applications!

Download the Android Environment: Java and ADT Bundle

Let's get started by making sure you have the current Android development environment. This means having the latest version of Java, Eclipse, and the Android Developer Tools (ADT). You may already have the most recent ADT Bundle installed, but I am going to do this here simply to make sure you are set up and starting from the right place, before we undertake the complex development we are about to embark upon within this book. If you keep your ADT up to date on a daily basis, you can skip this section if you wish.

Since Java is used as the foundation for ADT, get that first. As of Android 4.3, the Android IDE still uses Java 6, and not Java 7, so make sure to get the correct version of the Java SDK. It is located here:

http://www.oracle.com/technetwork/java/javasebusiness/downloads/java-archive-downloads-javase6-419409.html

Scroll down towards the bottom of the page, and look for the **Java SE Development Kit 6u45** download link. This section of the screen is shown in Figure 1-4.

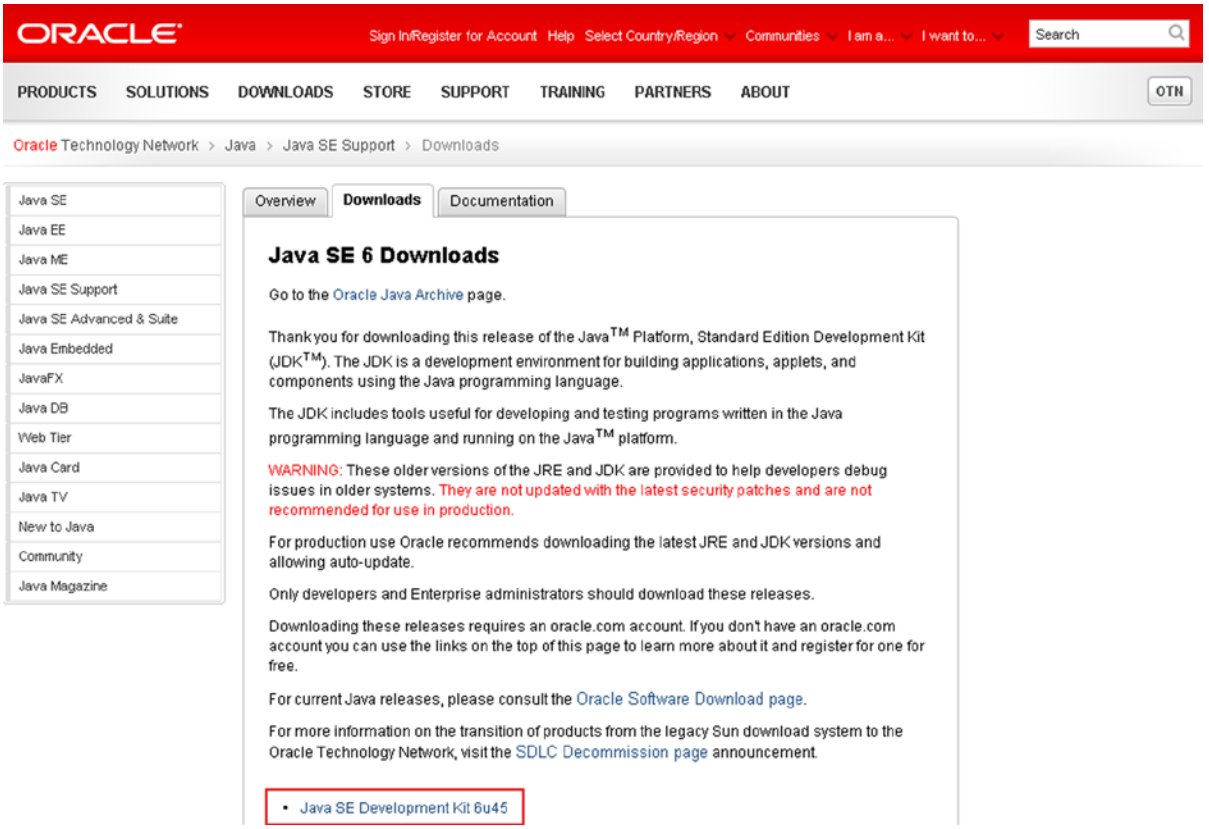
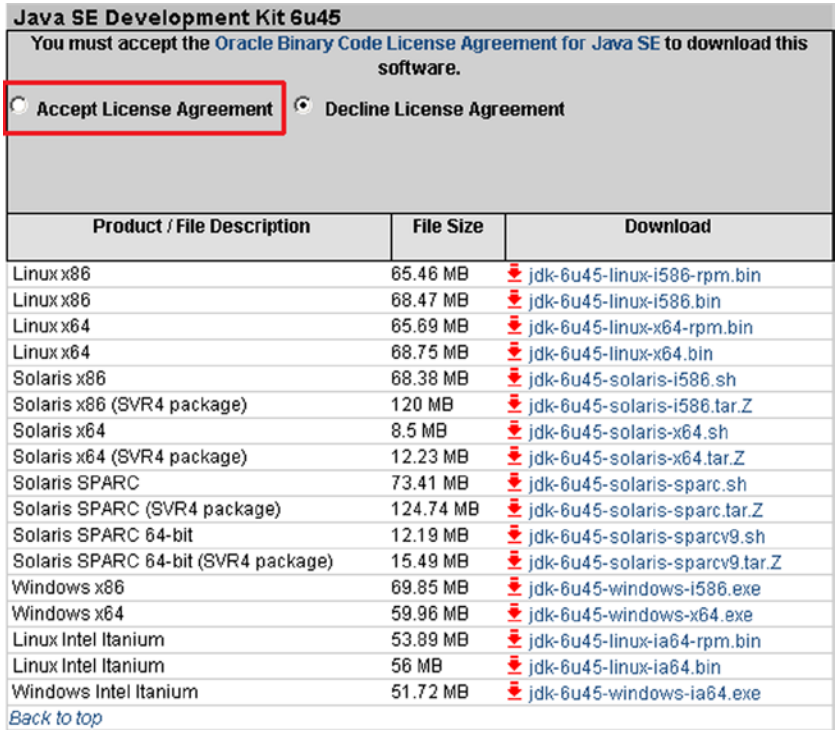


Figure 1-4. *Java SE 6 Download Section of the Oracle TechNetwork web site Java SE archives page*

Click the Java SE Development Kit 6u45 Download link on the bottom of the section and at the top of the download links screen. At the top of the downloads screen in the gray area shown in Figure 1-5, select the Accept License Agreement radio button option. Once you do this you will notice that the links at the right side will become bolder and can be clicked to invoke the download for your operating system.

If you are using a 64-bit OS such as Windows 7 64-bit or Windows 8 64-bit, which is what I am using, select the Windows x64 version of the EXE installer file to download.

If you are using a 32-bit OS such as Windows XP or Windows Vista 32-bit, select the Windows x86 version of the EXE installer file to download. Be sure to match the bit level of the software to the bit level capability of the OS that you are running. Figure 1-5 shows the download screen as it appears once the license agreement option radio button has been selected.



Java SE Development Kit 6u45

You must accept the Oracle Binary Code License Agreement for Java SE to download this software.

○ Accept License Agreement ◉ Decline License Agreement

Product / File Description	File Size	Download
Linux x86	65.46 MB	jdk-6u45-linux-i586-rpm.bin
Linux x86	68.47 MB	jdk-6u45-linux-i586.bin
Linux x64	65.69 MB	jdk-6u45-linux-x64-rpm.bin
Linux x64	68.75 MB	jdk-6u45-linux-x64.bin
Solaris x86	68.38 MB	jdk-6u45-solaris-i586.sh
Solaris x86 (SVR4 package)	120 MB	jdk-6u45-solaris-i586.tar.Z
Solaris x64	8.5 MB	jdk-6u45-solaris-x64.sh
Solaris x64 (SVR4 package)	12.23 MB	jdk-6u45-solaris-x64.tar.Z
Solaris SPARC	73.41 MB	jdk-6u45-solaris-sparc.sh
Solaris SPARC (SVR4 package)	124.74 MB	jdk-6u45-solaris-sparc.tar.Z
Solaris SPARC 64-bit	12.19 MB	jdk-6u45-solaris-sparcv9.sh
Solaris SPARC 64-bit (SVR4 package)	15.49 MB	jdk-6u45-solaris-sparcv9.tar.Z
Windows x86	69.85 MB	jdk-6u45-windows-i586.exe
Windows x64	59.96 MB	jdk-6u45-windows-x64.exe
Linux Intel Itanium	53.89 MB	jdk-6u45-linux-ia64-rpm.bin
Linux Intel Itanium	56 MB	jdk-6u45-linux-ia64.bin
Windows Intel Itanium	51.72 MB	jdk-6u45-windows-ia64.exe

Back to top

Figure 1-5. *Java SE 6 download links for Linux, Solaris, and Windows (once software agreement is accepted)*

Once the EXE file has finished downloading, make sure any previous version of Java 6 SDK is uninstalled by using the Windows Control Panel Add/Remove Programs dialog. Then find and launch an installer for the current version Java 6 SDK installer, and install the latest version of Java 6, so that you can install the Android Developer Tools ADT Bundle.

The Android Developer Tools (ADT) Bundle is comprised of the Eclipse Kepler 4.3 IDE (Integrated Development Environment) for Java and the Android Developer Tools Plug-ins already installed into the Eclipse IDE. This used to be done separately, and it took about a 50 step process to complete, so downloading and installing one pre-made bundle is significantly less work.

Next, you need to download the Android ADT Bundle from the Android Developer web site. In the past, developers had to assemble Eclipse and ADT plug-ins manually. Starting with Android 4.2, Jelly Bean + Google is now doing this for you, making installing an Android ADT IDE an order of magnitude easier that it was in the past. This is the URL to use to download an ADT Bundle:

```
http://developer.android.com/sdk/index.html
```

The screen shown in Figure 1-6 is what you should see on the Android SDK download page. Simply click the blue Download the SDK button to get started with the download process.

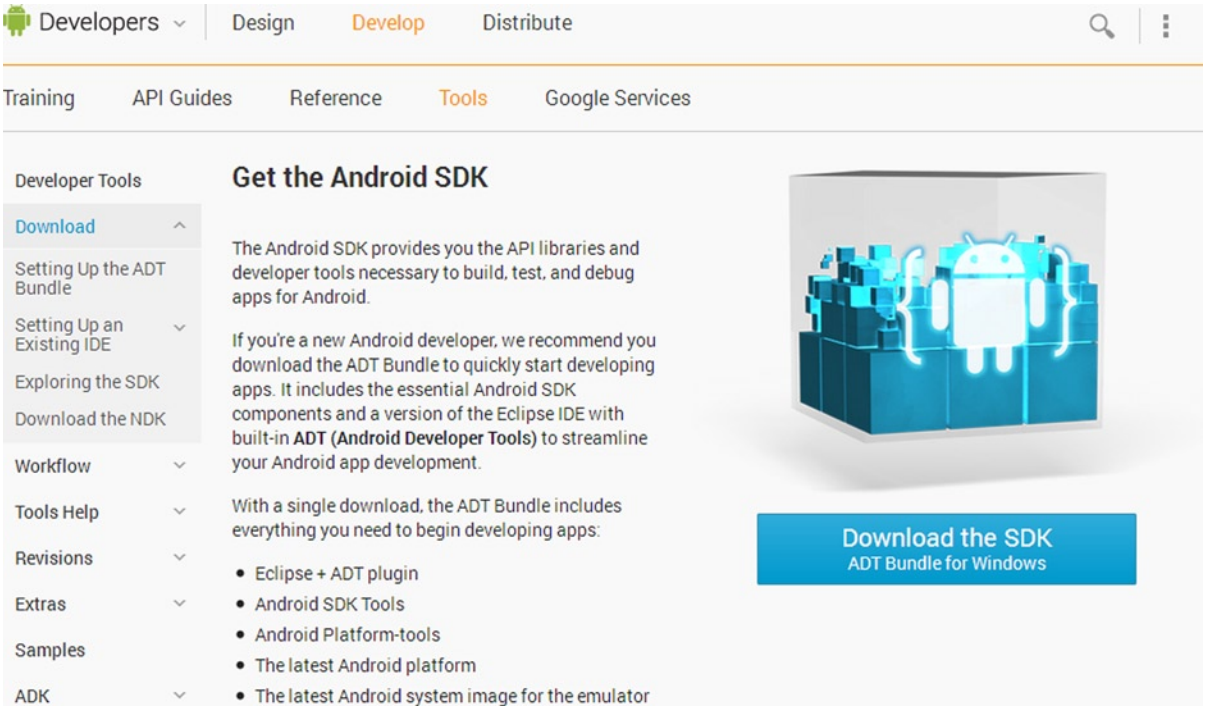


Figure 1-6. *ADT Bundle Download the SDK Button on Get the Android SDK page of the Android Developer site*

Once you click the Download the SDK button, you will be taken to the Licensing Terms and Conditions Agreement page, where you can read the terms and conditions of using the Android development environment and finally click the **I have read and agree with the above terms and conditions** checkbox.

Once you do this, the OS 32 or 64 bit-level selection radio buttons will be enabled so that you can select either the 32-bit or the 64-bit version of the Android ADT environment. Then the blue Download the SDK ADT Bundle for Windows (or your OS) will be enabled, and you can click it to start the installation file download process. This is the screen state that is shown in Figure 1-7.

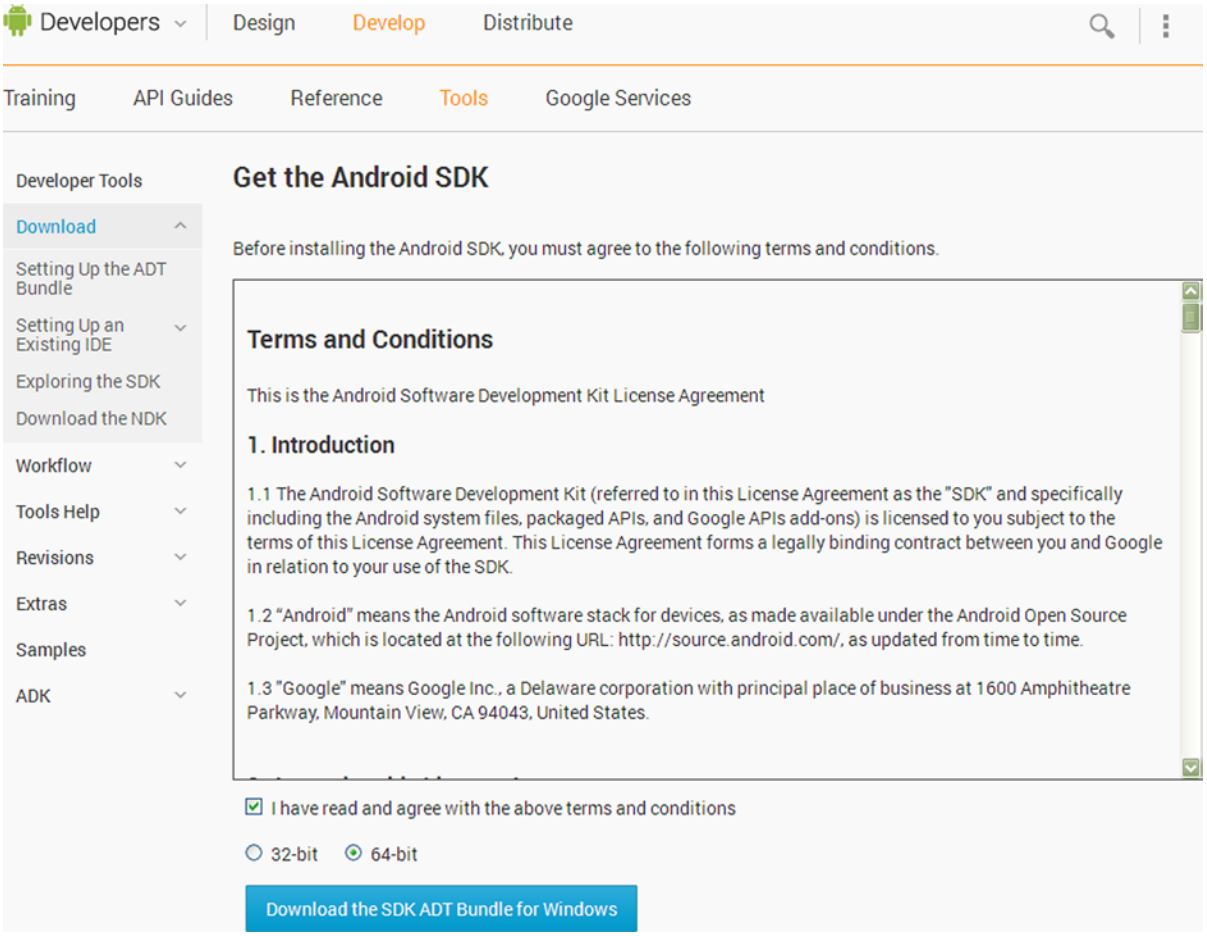


Figure 1-7. *Terms and Conditions page and SDK download options for 32-bit or 64-bit software environments*

Click the blue Download the SDK ADT Bundle button and save the ZIP file to your system downloads folder. Once the download is finished you can begin the installation process, which we will go through in detail in the next section of this chapter.

Now you are ready top unzip and install the ADT, and then update it to the latest version from inside of the Eclipse Java ADT IDE (after you install and launch it for the first time, of course). Are you getting excited yet?

Installing and Updating the Android Developer ADT Bundle

Open the Windows File Explorer utility, which should look like a folder icon with files in it (it is the second icon from the left in Figure 1-13). Next, find your **Downloads** folder, which should be showing at the top-left (underneath the Favorites section) of the file manager utility, as shown in Figure 1-8.

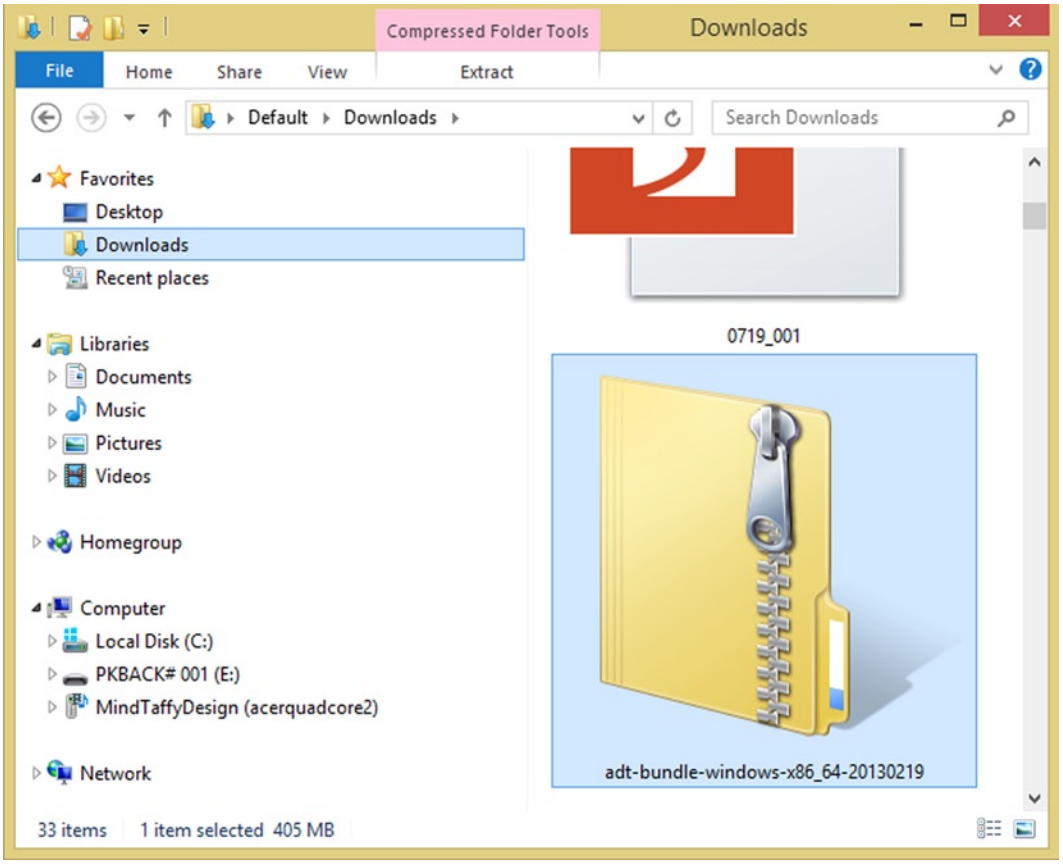


Figure 1-8. *Find the adt-bundle-windows-x86_64 ZIP file in Downloads*

Click the Downloads folder to highlight it in blue and find the ADT Bundle file that you just downloaded in the pane of files on the right side of the file management utility.

Right-click the `adt-bundle-windows-x86_64` file as shown in Figure 1-9 to bring up a context-sensitive menu and select the **Extract All** option.

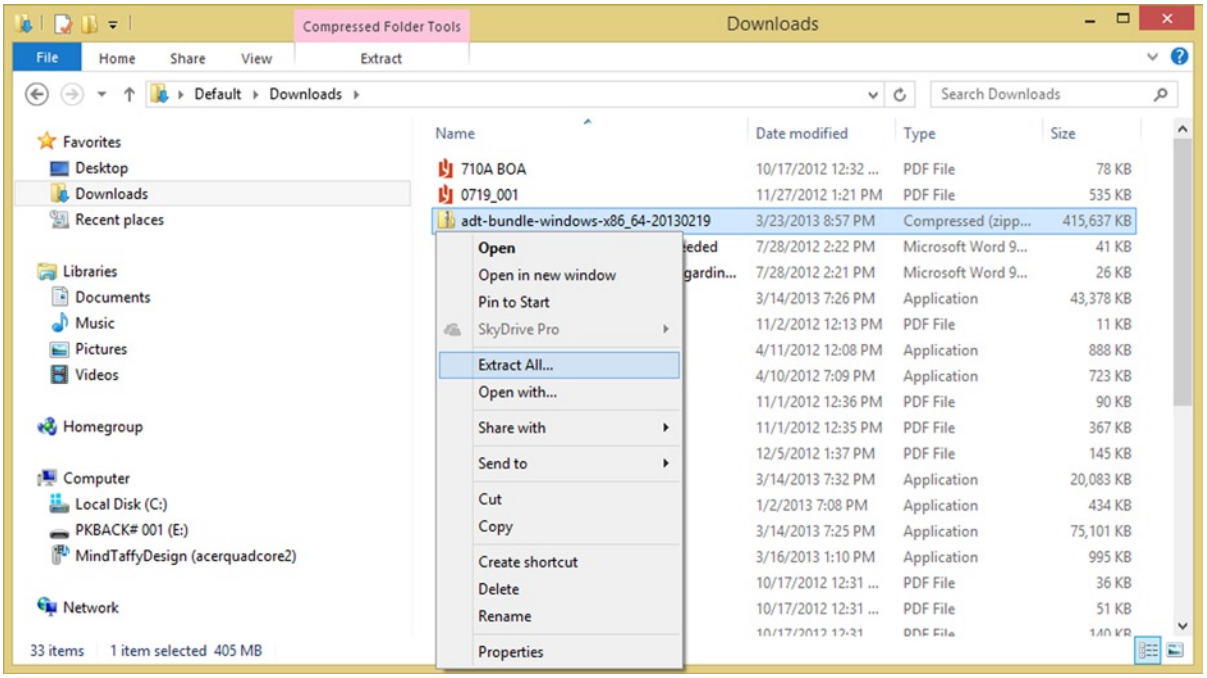


Figure 1-9. Right-click the adt-bundle ZIP file and select the Extract All option to begin the ADT installation

When the **Extract Compressed (Zipped) Folders** dialog appears, replace the default folder for installation with one of your own creation. I created an **Android** folder under my root **C:** hard drive (so, C:\Android) to keep my ADT IDE in, as that is a logical name for it. The before and after dialogs are shown in Figure 1-10, showing the difficult-to-remember path to my system Downloads folder and a new, easy-to-find **C:\Android** folder path.

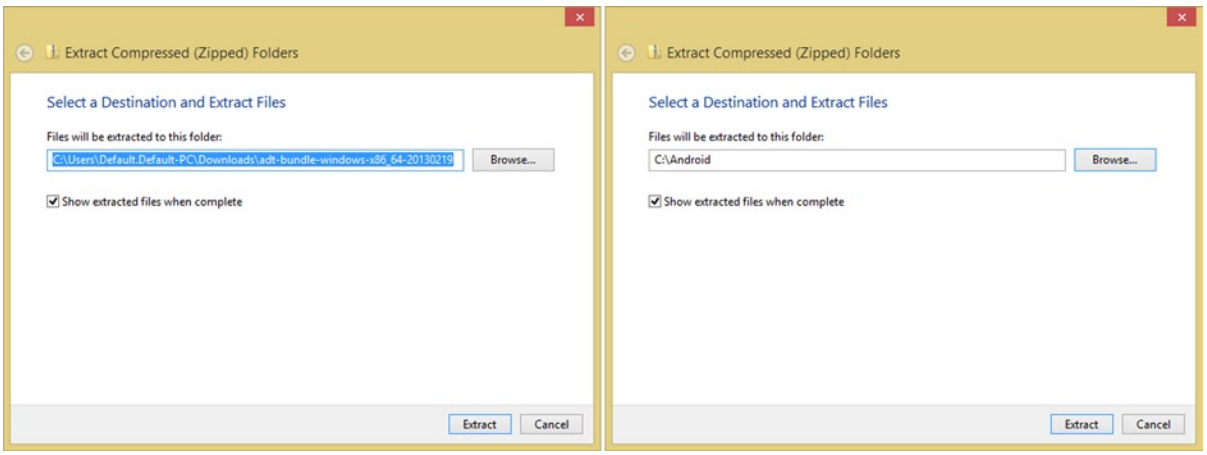


Figure 1-10. Change the target installation folder from your Downloads folder to an Android folder that you create

Once you click the **Extract** button, shown in Figure 1-10, you will see a **progress dialog** that shows the install as it is taking place. Click the **More Details** option located at the bottom left to see what files are being installed, as well as the Time remaining and the Items remaining counters, as shown in Figure 1-11. The 600MB installation takes from 15 to 60 minutes, depending upon the data transfer speed of your hard disk drive.

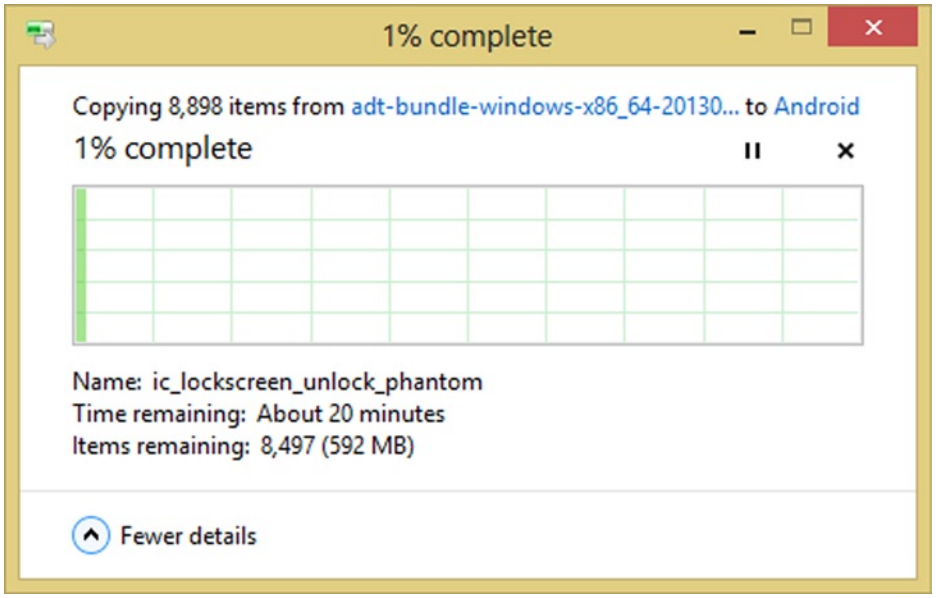


Figure 1-11. *Expanded More Details option showing which files are installing*

Once the installation is complete, go back into your Windows Explorer File Management Utility and look under the C:\Android folder (or whatever you decided to name it) and you will see the adt-bundle-windows-x86_64 folder, as shown in Figure 1-12. Open this and you will see an **eclipse** and an **sdk** sub-folder. Open those sub-folders as well, in order to see their sub-folders, so that you know what is in there.

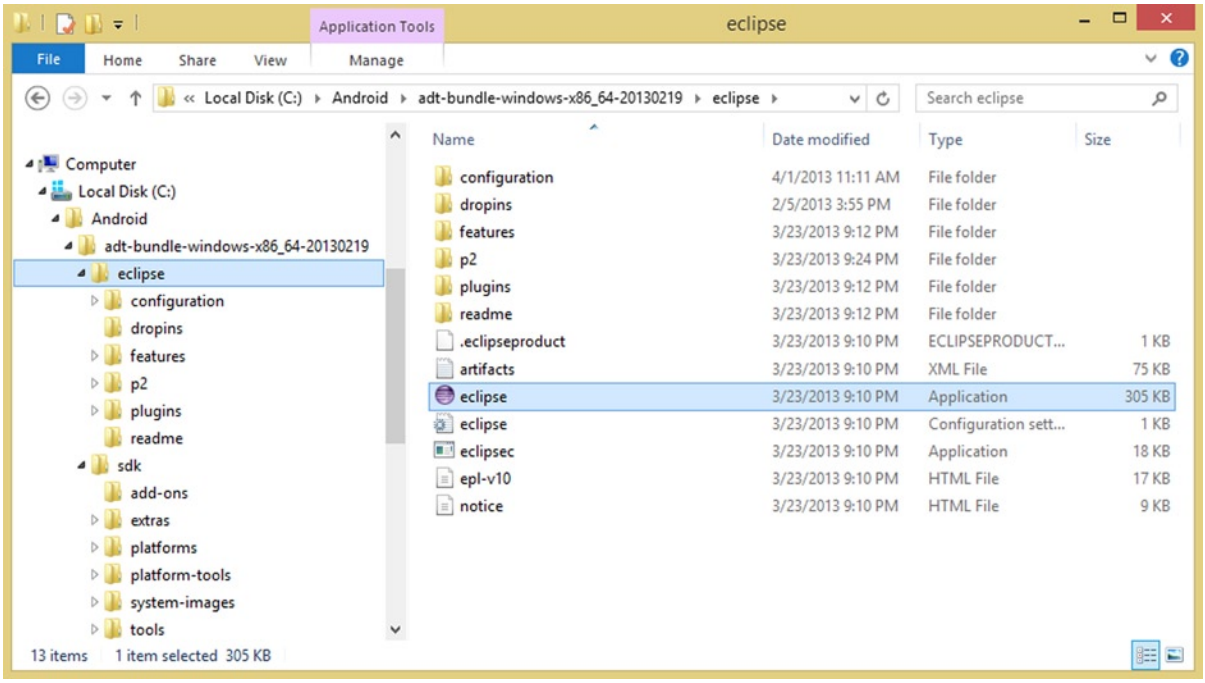


Figure 1-12. *Finding the Eclipse Application executable file in the ADT Bundle folder hierarchy you just installed*

Next, click the eclipse folder on the left side of your file management utility to show the file contents in the right side of the file manager. Find the eclipse Application executable file, which will be the one that has its own custom icon next to it, on the left. It is a purple sphere.

Click and drag the Eclipse icon to the bottom of your desktop (or wherever your Taskbar Launch area is mounted to your OS desktop), and hover it over your Installed Program Launch Icon Taskbar. Once you do this, you will see the **Pin to Taskbar** (Windows Vista, Windows 7) or **Pin to eclipse** (Windows 8) tool-tip message, as is shown in the top section of Figure 1-13.

Figure 1-13. *Dragging the Eclipse application onto the Windows Taskbar to invoke the pin operation*

Once this tool-tip message is showing, you can release the drag operation, and drop the eclipse purple sphere icon into your Taskbar area, where it will become a permanent application launch icon, as shown in the bottom section of Figure 1-13.

Now, all you have to do when I say “launch Eclipse ADT now, and let’s get started” is click your mouse once on the eclipse icon, and it will launch!

So let’s try it. Click the Eclipse software icon once in your Taskbar and launch the software for the first time. You will see the ADT Android Developer Tools start-up screen, as shown at the left side of Figure 1-14. Once the software loads into your system memory, you will see the **Workspace Launcher** dialog, shown on the right, with the **Select a workspace** work process, which will allow you to set your default Android development workspace location on your workstation hard disk drive.

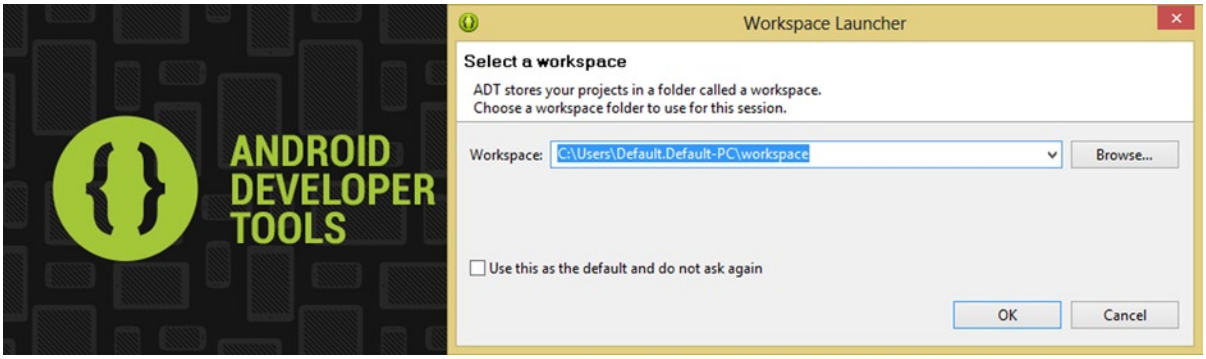


Figure 1-14. *Android Developer Tools start-up screen and Workspace Launcher dialog showing default workspace*

I accepted the default workspace location, which will be under your main hard disk drive letter (probably C:\) in your Users folder, under a sub-folder named using your PC’s assigned name; your Android development workspace folder will be underneath that.

When you create projects in Android ADT, they will appear as sub-folders underneath this workspace folder hierarchy, so you’ll be able to find all of your files using your File Management Utility software as well as using the Eclipse package Explorer project navigation pane, which you’ll be using quite a bit in this book, to learn about how Android implements Graphics.

Once you set your workspace location and click the OK button, the Eclipse Java ADT start-up **Welcome!** screen will appear, as shown in Figure 1-15. The first thing that you want to do is make sure your software is completely up to date, so click the **Help** menu at the top right of the screen, and select the **Check for Updates** option about two-thirds of the way down, as shown in Figure 1-15 (and highlighted in blue).

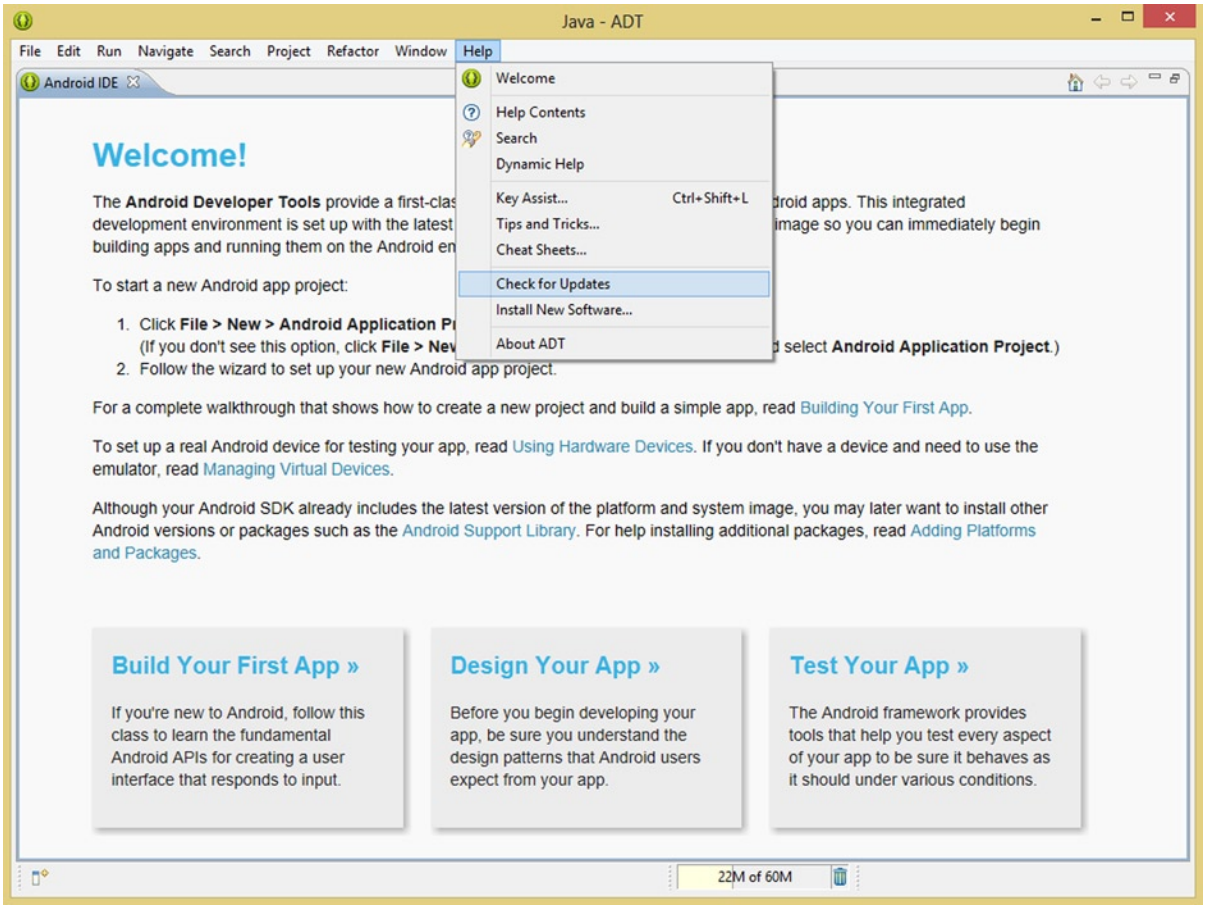


Figure 1-15. *Eclipse Java ADT Welcome screen and invoking the Help ➤ Check for Updates menu sequence*

Once you select this menu option, you will see the **Contacting Software Sites** dialog, shown in Figure 1-16 on the left-hand side. This shows the Checking for updates progress bar as it checks various Google Android software repository sites for any updated versions of the Eclipse ADT.

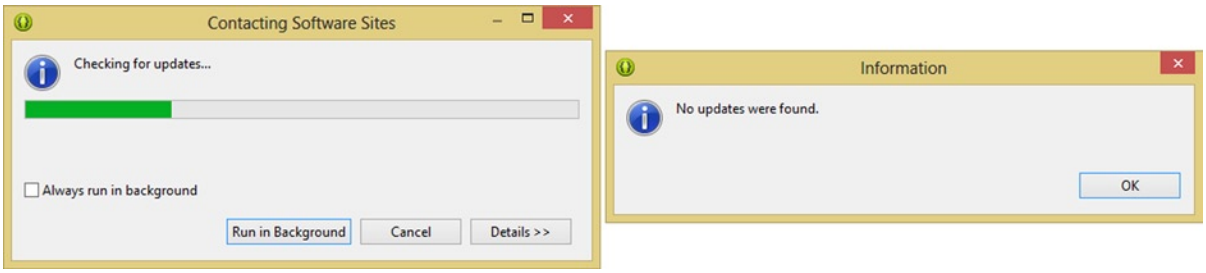


Figure 1-16. *The Contacting Software Sites dialog checking for updates to Eclipse ADT, with no updates found*

It is important to note that you must (still) be connected to the Internet for this type of real-time software package updating to be able to occur. In my case, since I had just downloaded the ADT development environment, no new updates were found, and so I got a dialog advising me of this fact.

If for some reason there are updates to the Eclipse ADT environment, which there shouldn't be if you just downloaded and installed it, simply follow the instructions to update any components of the ADT that need updating. In this way you will have the latest software development kit (SDK) versions at all times.

It is important to note that you can run the **Help ➤ Check for Updates** menu command sequence at any time. Some developers do this weekly or even daily to make sure that they have the most updated (bug-free and feature-laden) Android development environment possible at all times.

Summary

In this first chapter, I laid the foundation for the rest of the book by covering some of the key underlying principles of graphics design and digital imaging, and by making sure that you have the latest Java SDK and Android ADT SDK installed and updated so that we can start writing graphics design code.

Most of the concepts in this chapter also apply to digital video, 2D and 3D animation, and special effects creation. Thus, you won't have to duplicate any of this knowledge over the next few chapters, but it's important that you understand it here and now.

We first took a look at the various digital image file formats that are currently supported in the Android operating system. These include the outdated-but-still-supported Compuserve GIF format and the JPEG format, as well as the preferred PNG format and the new WebP format.

You learned about lossy and lossless digital image compression algorithms, and found out why Android prefers us to use the latter in order to get a higher quality visual result for graphics-intensive applications.

We then did a quick overview of the Android View and ViewGroup classes, which are used to hold and display digital images and digital video. We also reviewed the android.widget package, which holds many of the user interface classes that we will be utilizing frequently during this book.

Next, we looked at the building block of digital imaging and video, the pixel, and the fundamental concepts of resolution and aspect ratio. You learned how to calculate the number of pixels in an image in order to find its raw data footprint size. You learned all about aspect ratio and how it defines the shape of your image using a ratio of Width-to-Height multipliers.

Then we looked at how color is handled in digital images using some color theory and terminology. You also learned about color depth and additive color and how color is created using multiple color channels in an image.

Next, we looked at hexadecimal notation, and how colors are represented in the Android OS using two hexadecimal value slots per color channel. You learned that 24-bit RGB images use six slots and 32-bit ARGB images use eight slots. You learned that hexadecimal values are represented in Android by using a pound sign before the hexadecimal data values.

We then looked at digital image compositing and the concepts of alpha channels and pixel blending modes. We explored the power of holding different image elements on digital layers of transparency, using alpha channels, and algorithmically blending any pixel in an image with any of those layers via dozens of different blending modes in the Porter-Duff class in Android.

Next, we looked at the concept of using the alpha channel capabilities to create image masks, which allows us to extract subject material out of an image so we can later manipulate it individually using Java code or use it in compositing layers to create a more complicated images.

Then, we looked at the concept of anti-aliasing and how it allows us to achieve a smooth, professional compositing result by blending pixel color values at the edges between two different objects, or between an object and its background. We saw how using anti-aliasing in an alpha channel mask allows us to obtain a smooth composite between that object and a background image.

Next, we covered the main factors that are important in image compression, and you learned how to achieve a compact data footprint for these digital image assets. You learned about dithering and how it allows you to use 8-bit indexed color images with good results to reduce file size significantly.

Finally, you downloaded and installed the latest Java and Android ADT SDK software and then configured it for use on your workstation. You did this so that you are now completely ready to develop graphics-oriented Android application software within the rest of the chapters in this book.

In the next chapter, you will learn about digital video formats, concepts, and optimization. Thus, that chapter will be very similar to this one where you will get all the fundamental concepts regarding digital video under your belt, and you will also create the framework for your Android application, which you will be super-charging, from a graphics design perspective at least, during the duration of this Pro Android Graphics Design book.

Chapter 2

Android Digital Video: Formats, Concepts, and Optimization

In this second chapter, we will take a closer look at how digital video is implemented inside the Android operating system. We will build upon the concepts you learned in the previous chapter, as all of a digital image's characteristics apply equally to digital video.

After all, digital video is really just a series of moving digital images. This motion aspect of digital video introduces a high level of complexity, as a fourth dimension (time) is added into the mathematical equation. This makes working with digital video an order of magnitude more complex than working with digital images, especially for the codec, which you will be learning about very soon.

This is especially true when it comes to video compression, and thus, this chapter will focus even more on video codecs and the proper way to get the smallest possible data footprint using a new media data file format that's traditionally been (and continues to be) a gigabyte-laden monstrosity.

We will take a look at the digital video file formats that are currently supported in Android, as well as the MediaPlayer class, which allows video to be played back on the screen. You will learn all about the VideoView user interface widget, which Android has created to implement a MediaPlayer for our ease of use in a pre-built UI container.

You'll learn basic digital video concepts that you will need to understand in order to follow what we will be doing with digital video during this chapter. We will take a look at digital video optimization from a digital video asset data footprint standpoint (app size) as well as from Android device types (a market coverage standpoint). You'll create a **pro.android.graphics** Java package and a **GraphicsDesign** Android application in this chapter.

Android Digital Video Formats: MPEG4 H.264 and WebM VP8

Android supports the same two **open source** digital video formats that **HTML5** supports natively, namely **MPEG-4** (Motion Picture Experts Group) **H.264** and the ON2 **VP8** format, which was acquired by Google from ON2 Technologies and renamed **WebM.** WebM was subsequently released as an open source digital video file format and is now in Android OS as well as all browsers.

This is very convenient from a **content production** standpoint, as the video content that developers produce and optimize can be used in HTML5 engines such as browsers and HTML5-based OSes, as well as in Android applications.

This open source digital video format cross-platform support scenario will afford content developers with a "produce it once, deliver it everywhere" content production situation. This will reduce content development costs and increase developer revenues, as long as these economies of scale are taken advantage of by the professional video graphics developer.

Like I did in Chapter 1, I will cover the MPEG and WebM video file formats from oldest to most recent. The oldest format supported by Android is MPEG H.263, which should only be used for lower resolution, as it has the worst **compression-to-quality** result because it uses the oldest technology.

Since most Android devices these days have screens that are using a medium (854x480) to high (1280x720) resolution, if you are going to use the MPEG file format, you should utilize the **MPEG-4 H.264** format, which is the most widely used digital video file format in the world currently.

MPEG-4 H.264 format is used by commercial broadcasters, HTML5 web browser software, Mobile HTML5 Apps, and Android OS. All of these summed together are rapidly approaching a majority market share position, to say the least.

The **MPEG-4 H.264 AVC** (**Advanced Video Coding**) digital video file format is supported across all Android OS versions for video playback, and under **Android 3.0** and later OS versions for **video recording**. Recording video is only supported if the Android device has video camera hardware capability.

There is also an **MPEG4 SP** (Simple Profile) video format, which is supported for **commercial video** file playback. This format is available across every Android OS version for broadcast-type content playback (films, television programs, mini-series, TV series, workout videos, and similar products).

If you are an individual Android content producer, however, you will find that the MPEG-4 H.264 AVC format has far better compression, especially if you are using one of the more advanced (better) encoding suites, like the **Sorenson Squeeze Pro 9** software (which you'll be using later in the chapter).

MPEG-4 H.264 AVC will be the format that you will use for Android video content production if you decide to use the MPEG-4 digital video format in your Android application rather than the WebM (VP8) video format from Google. The reason for the MPEG4 SP support is most likely because many commercial videos and films were originally compressed using this older video format, and playback of these commercial video titles on user's Android devices is a popular use for these devices these days, due to a dearth of interactive content. But you are going to do something about that, aren't you?! Great!

File extensions supported for MPEG-4 video files include **.3GP** (SP) and **.MP4 (AVC)**. I prefer to use the latter (.MP4 AVC), as that is what I use in HTML5 apps, and it is more common to use the superior AVC format, but either type of file (extension) should work just fine in the Android OS.

The more modern (advanced) digital video format that Android now supports is called the **WebM** or **VP8** digital video format, and this format provides a higher quality result with a **smaller data footprint**. This is probably the reason why Google acquired ON2, the company that developed the VP8 codec. You will learn all about codecs later in this chapter.

Playback of WebM video is "natively" supported in Android **2.3.3** and later, so most the Android devices out there, including an Amazon Kindle Fire HD as well as the original Kindle Fire, should be able to support this higher quality digital video file format. This is because it's **natively a part of the operating system** (OS) software installed on your smartphone or tablet.

WebM also supports **video streaming**, which you'll learn about in a later section of this chapter. WebM video format streaming playback capability is supported if users have Android OS version **4.0** (or later). For this reason I recommend using MPEG-4 H.264 AVC for captive (non-streaming) video assets and WebM if you are going to be streaming video. We will cover advanced video concepts such as streaming and the like in the last section of this book.

Android VideoView and MediaPlayer Class: Video Players

There are two major classes in Android, both subclassed directly from the **java.lang.Object** superclass, that deal directly with digital video format playback. They are the **VideoView** widget class, from the **android.widget** package, and the **MediaPlayer** media class, from the **android.media** package.

Most of your basic digital video playback usage should be accomplished by using the **<VideoView>** XML tag and its parameters, and designed right into your user interface designs, as you will be doing later on in the chapter.

Android's VideoView class is a direct subclass of the Android **SurfaceView** class, which is a display class that provides a dedicated **drawing surface** embedded inside a specialized Android View hierarchy used for implementing an advanced, direct-to-the-screen drawing graphics pipeline.

A SurfaceView is **z-ordered** so that it lives behind the View windows that are holding the SurfaceView, and thus a SurfaceView cuts this viewport through its View window to allow content to be displayed to a user.

Z-order is a concept that we will get into later on in the book when you start working extensively with layered composites, but in a nutshell, the z-order is the **order in the layer stack** of a given image or video source.

Layers are arranged, at least conceptually, from the top layer down to the bottom layer. This is the reason an image (or video) asset's order in this layer hierarchy is called the z-order, as layers containing the x and y 2D image data are stacked along the 3D **z axis**. Forget about learning about 3D z axes and z-order later on in this book; it looks like I just explained it!

As you now know, 2D images and video only use an x and y axis to "address" their data, so you will need to look at z-order and compositing from a 3D standpoint, as layers in a composite need to exist along a third z axis.

A SurfaceView class is a subclass of the **View** class, which, as you know, is a subclass of the Java **Object** master class. The **VideoView** subclass is thus a specialized incarnation of the SurfaceView subclass, adding more methods to the SurfaceView class methods, and thus is farther down within