In this package, you will find:

A Biography of the authors of the book

A preview chapter from the book, Chapter NO.2 "Viewing the World"

A synopsis of the book’s content

Information on where to buy this book

About the Authors

**Jens Grubert** is a researcher at the Graz University of Technology. He has received his Bakkalaureus (2008) and Dipl.-Ing. with distinction (2009) at Otto-von-Guericke University Magdeburg, Germany. As a research manager at Fraunhofer Institute for Factory Operation and Automation IFF, Germany, he conducted evaluations of industrial Augmented Reality systems until August 2010. He has been involved in several academic and industrial projects over the past years and is the author of more than 20 international publications. His current research interests include mobile interfaces for situated media and user evaluations for consumer-oriented Augmented Reality interfaces in public spaces. He has over four years of experience in developing mobile Augmented Reality applications. He initiated the development of a natural feature tracking system that is now commercially used for creating Augmented Reality campaigns. Furthermore, he is teaching university courses about Distributed Systems, Computer Graphics, Virtual Reality, and Augmented Reality.

Website: [www.jensgrubert.com](http://www.jensgrubert.com).

I want to thank my family, specifically Carina Nahrstedt, for supporting me during the creation of this book.

For More Information:

Dr. Raphael Grasset is a senior researcher at the Institute for Computer Graphics and Vision. He was previously a senior researcher at the HIT Lab NZ and completed his Ph.D. in 2004. His main research interests include 3D interaction, computer-human interaction, augmented reality, mixed reality, visualization, and CSCW. His work is highly multidisciplinary; he has been involved in a large number of academic and industrial projects over the last decade. He is the author of more than 50 international publications, was previously a lecturer on Augmented Reality, and has supervised more than 50 students. He has more than 10 years of experience in Augmented Reality (AR) for a broad range of platforms (desktop, mobile, and the Web) and programming languages (C++, Python, and Java). He has contributed to the development of AR software libraries (ARToolKit, osgART, and Android AR), AR plugins (Esperient Creator and Google Sketchup), and has been involved in the development of numerous AR applications.

Website: www.raphaelgrasset.net.
Augmented Reality for Android Application Development

Augmented Reality offers the magic effect of blending the physical world with the virtual world and brings applications from your screen into your hands. Augmented Reality redefines advertising and gaming as well as education in an utterly new way; it will become a technology that needs to be mastered by mobile application developers. This book enables you to practically implement sensor-based and computer vision-based Augmented Reality applications on Android. Learn about the theoretical foundations and practical details of implemented Augmented Reality applications. Hands-on examples will enable you to quickly develop and deploy novel Augmented Reality applications on your own.

What This Book Covers

Chapter 1, Augmented Reality Concepts and Tools, introduces the two major Augmented Reality approaches: sensor-based and computer vision-based Augmented Reality.

Chapter 2, Viewing the World, introduces you to the first basic step in building Augmented Reality applications: capturing and displaying the real world on your device.

Chapter 3, Superimposing the World, helps you use JMonkeyEngine to overlay high-fidelity 3D models over the physical world.

Chapter 4, Locating in the World, provides the basic building blocks to implement your own Augmented Reality browser using sensors and GPS.

Chapter 5, Same as Hollywood – Virtual on Physical Objects, explains you the power of the VuforiaTM SDK for computer vision-based AR.

Chapter 6, Make It Interactive – Create the User Experience, explains how to make Augmented Reality applications interactive. Specifically, you will learn how to develop ray picking, proximity-based interaction, and 3D motion gesture-based interaction.

Chapter 7, Further Reading and Tips, introduces more advanced techniques to improve any AR application's development.

For More Information:
In this chapter, we will learn how to develop the first element of any mobile AR application: the view of the real world. To understand the concept of the view of the real world, we will take a look at the camera application you have installed on your mobile. Open any photo capture application (camera app) you have preinstalled on your android device, or you may have downloaded from the Google Play store (such as Camera Zoom FX, Vignette, and so on). What you can see on the viewfinder of the application is a real-time video stream captured by the camera and displayed on your screen.

If you move the device around while running the application, it seems like you were seeing the real world "through" the device. Actually, the camera seems to act like the eye of the device, perceiving the environment around you. This process is also used for mobile AR development to create a view of the real world. It's the concept of see-through video that we introduced in the previous chapter.

The display of the real world requires two main steps:

- Capturing an image from the camera (camera access)
- Displaying this image on the screen using a graphics library (camera display in JME)

This process is generally repeated in an infinite loop, creating the real-time aspect of the view of the physical world. In this chapter, we will discuss how to implement both of these techniques using two different graphics libraries: a low-level one (Android library) and a high-end one (JME 3D scene graph library). While the Android library allows you to quickly display the camera image, it is not designed to be combined with 3D graphics, which you want to augment on the video stream. Therefore, you will implement the camera display also using the JME library. We will also introduce challenges and hints for handling a variety of Android smartphones and their inbuilt cameras.

For More Information:
Understanding the camera

Phone manufacturers are always competing to equip your smartphone with the most advanced camera sensor, packing it with more features, such as higher resolution, better contrast, faster video capture, new autofocus mode, and so on. The consequence is that the capabilities (features) of the mobile phone cameras can differ significantly between smartphone models or brands. Thankfully, the Google Android API provides a generic wrapper for the underlying camera hardware unifying the access for the developer: the Android camera API. For your development, an efficient access to the camera needs a clear understanding of the camera capabilities (parameters and functions) available through the API. Underestimating this aspect will result in slow-running applications or pixelated images, affecting the user experience of your application.

Camera characteristics

Cameras on smartphones nowadays share many characteristics with digital point-and-shoot cameras. They generally support two operative modes: the still image mode (which is an instantaneous, singular capture of an image), or the video mode (which is a continuous, real-time capture of images).

Video and image modes differ in terms of capabilities: an image capture always has, for example, a higher resolution (more pixels) than video. While modern smartphones can easily achieve 8 megapixel in the still image mode, the video mode is restricted to 1080p (about 2 megapixels). In AR, we use the video mode in typically lower resolutions such as VGA (640 x 480) for efficiency reasons. Unlike a standard digital camera, we don't store any content on an external memory card; we just display the image on the screen. This mode has a special name in the Android API: the preview mode.

Some of the common settings (parameters) of the preview mode are:

- **Resolution**: It is the size of the captured image, which can be displayed on your screen. This is also called the size in the Android camera API. Resolution is defined in pixels in terms of width (x) and height (y) of the image. The ratio between them is called the *aspect ratio*, which gives a sense of how square an image is similar to TV resolution (such as 1:1, 4:3, or 16:9).
- **Frame rate**: It defines how fast an image can be captured. This is also called *Frames Per Second* (FPS).
- **White balance**: It determines what will be the white color on your image, mainly dependent on your environment light (for example, daylight for outdoor situation, incandescent at your home, fluorescent at your work, and so on).

For More Information:

• **Focus**: It defines which part of the image will appear sharp and which part will not be easily discernible (out of focus). Like any other camera, smartphone cameras also support autofocus mode.

• **Pixel format**: The captured image is converted to a specific image format, where the color (and luminance) of each pixel is stored under a specific format. The pixel format not only defines the type of color channels (such as RGB versus YCbCr), but also the storage size of each component (for example, 5, 8, or 16 bits). Some popular pixel formats are RGB888, RGB565, or YCbCr422. In the following figure, you can see common camera parameters, moving from the left to right: image resolution, frame rate for capturing image streams, focus of the camera, the pixel format for storing the images and the white balance:

Other important settings related to the camera workflow are:

• **Playback control**: Defines when you can start, pause, stop, or get the image content of your camera.

• **Buffer control**: A captured image is copied into the memory to be accessible to your application. There are different ways to store this image, for example, using a buffering system.

Configuring these settings correctly is the basic requirement for an AR application. While popular camera apps use only the preview mode for capturing a video or an image, the preview mode is the basis for the view of the real world in AR. Some of the things you need to remember for configuring these camera parameters are:

• The higher the resolution, the lower will be your frame rate, which means your application might look prettier if things do not move fast in the image, but will run more slowly. In contrast, you can have an application running fast but your image will look "blocky" (pixelated effect).

• If the white balance is not set properly, the appearance of digital models overlaid on the video image will not match and the AR experience will be diminished.

For More Information:
Viewing the World

- If the focus changes all the time (autofocus), you may not be able to analyze the content of the image and the other components of your application (such as tracking) may not work correctly.

- Cameras on mobile devices use compressed image formats and typically do not offer the same performance as high-end desktop webcams. When you combine your video image (often in RGB565 with 3D rendered content using RGB8888), you might notice the color differences between them.

- If you are doing heavy processing on your image, that can create a delay in your application. Additionally, if your application runs multiple processes concurrently, synchronizing your image capture process with the other processes is rather important.

We advise you to:

- Acquire and test a variety of Android devices and their cameras to get a sense of the camera capabilities and performances.

- Find a compromise between the resolution and frame rate. Standard resolution/frame rate combination used on desktop AR is 640 x 480 at 30 fps. Use it as a baseline for your mobile AR application and optimize from there to get a higher quality AR application for newer devices.

- Optimize the white balance if your AR application is only supposed to be run in a specific environment such as in daylight for an outdoor application.

- Controlling the focus has been one of the limiting aspects of Android smartphones (always on autofocus or configuration not available). Privilege a fixed focus over an autofocus, and optimize the focus range if you are developing a tabletop or room AR application (near focus) versus an outdoor AR application (far focus).

- Experiment with pixel formats, to get the best match with your rendered content.

- Try to use an advanced buffering system, if available, on your target device.

There are other major characteristics of the camera that are not available through the API (or only on some handheld devices), but are important to be considered during the development of your AR application. They are field of view, exposure time, and aperture.

We will only discuss one of them here: the field of view. The field of view corresponds to how much the camera sees from the real world, such as how much your eyes can see from left to right and top to bottom (human vision is around 120 degrees with a binocular vision). The field of view is measured in degrees, and varies largely between cameras (15 degrees to 60 degrees without distortion).

For More Information:
The larger your field of view is, the more you will capture the view of the real world and the better will be the experience. The field of view is dependent on the hardware characteristics of your camera (the sensor size and the focal length of the length). Estimating this field of view can be done with additional tools; we will explore this later on.

**Camera versus screen characteristics**

The camera and screen characteristics are generally not exactly the same on your mobile platform. The camera image can be, for example, larger than the screen resolution. The aspect ratio of the screen can also differ for one of the cameras. This is a technical challenge in AR as you want to find the best method to fit your camera image on the screen, to create a sense of AR display. You want to maximize the amount of information by putting as much of the camera image on your screen as possible. In the movie industry, they have a similar problem as the recorded format may differ from the playing media (for example, the cinemascope film on your 4:3 mobile device, the 4K movie resolution on your 1080p TV screen, and so on). To address this problem, you can use two fullscreen methods known as stretching and cropping, as shown in the following figure:

![Stretching and Cropping](image)

Stretching will adapt the camera image to the screen characteristics, at the risk of deforming the original format of the image (mainly its aspect ratio). Cropping will select a subarea of the image to be displayed and you will lose information (it basically zooms into the image until the whole screen is filled). Another approach will be to change the scale of your image, so that one dimension (width or height) of the screen and the image are the same. Here, the disadvantage is that you will lose the fullscreen display of your camera image (a black border will appear on the side of your image). None of the techniques are optimal, so you need to experiment what is more convenient for your application and your target devices.

**Accessing the camera in Android**

To start with, we will create a simple camera activity to get to know the principles of camera access in Android. While there are convenient Android applications that provide quick means for snapping a picture or recording a video through Android intents, we will get our hands dirty and use the Android camera API to get a customized camera access for our first application.

For More Information:

We will guide you, step-by-step, in creating your first app showing a live camera preview. This will include:

- Creating an Eclipse project
- Requesting relevant permissions in the Android Manifest file
- Creating SurfaceView to be able to capture the preview frames of the camera
- Creating an activity that displays the camera preview frames
- Setting camera parameters

**Downloading the example code**

You can download the example code files for all Packt books you have purchased from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you. You can also find the code files at https://github.com/arandroidbook/ar4android.

**Creating an Eclipse project**

Our first step is the setup process for creating an Android project in Eclipse. We will call our first project CameraAccessAndroid. Please note that the description of this subsection will be similar for all other examples that we will present in this book.

Start your Eclipse project and go to File | New | Android Application Project. In the following configuration dialog box, please fill in the appropriate fields as shown in the following screenshot:
Then, click on two more dialog boxes (Configure Project for selecting the file path to your project, Launcher Icon) by accepting the default values. Then, in the Create Activity dialog box, select the Create Activity checkbox and the BlankActivity option. In the following New Blank Activity dialog, fill into the Activity Name textbox, for example, with CameraAccessAndroidActivity and leave the Layout Name textbox to its default value. Finally, click on the Finish button and your project should be created and be visible in the project explorer.

Permissions in the Android manifest
For every AR application we will create, we will use the camera. With the Android API, you explicitly need to allow camera access in the Android manifest declaration of your application. In the top-level folder of your CameraAccessAndroid project, open the AndroidManifest.xml file in the text view. Then add the following permission:

```xml
<uses-permission android:name="android.permission.CAMERA" />
```

Besides this permission, the application also needs to at least declare the use of camera features:

```xml
<uses-feature android:name="android.hardware.camera" />
```

Since we want to run the AR application in fullscreen mode (for better immersion), add the following option into the activity tag:

```xml
android:theme="@android:style/Theme.NoTitleBar.Fullscreen"
```

Creating an activity that displays the camera
In its most basic form, our Activity class takes care of setting up the Camera instance. As a class member, you need to declare an instance of a Camera class:

```java
public class CameraAccessAndroidActivity extends Activity {
    private Camera mCamera;
}
```
The next step is to open the camera. To do that, we define a `getCameraInstance()` method:

```java
public static Camera getCameraInstance() {
    Camera c = null;
    try {
        c = Camera.open(0);
    } catch (Exception e) { ... }
    return c;
}
```

It is important that the `open()` call is surrounded by `try{}catch{}` blocks as the camera might currently be used by other processes and be unavailable. This method is called in the `onResume()` method of your `Activity` class:

```java
public void onResume() {
    super.onResume();
    stopPreview = false;
    mCamera = getCameraInstance();
    ...
}
```

It is also crucial to properly release the camera when you pause or exit your program. Otherwise it will be blocked if you open another (or the same) program. We define a `releaseCamera()` method for this:

```java
private void releaseCamera() {
    if (mCamera != null) {
        mCamera.release();
        mCamera = null;
    }
}
```

You then call this method in the `onPause()` method of your `Activity` class.

On some devices, it can be slow to open the camera. In this case, you can use an `AsyncTask` class to mitigate the problem.

### Setting camera parameters

You now have a basic workflow to start and stop your camera. The Android camera API also allows you to query and set various camera parameters that were discussed at the beginning of this chapter. Specifically, you should be careful not to use very high resolution images as they take a lot of processing power. For a typical mobile AR application, you do not want to have a higher video resolution of 640 x 480 (VGA).

<table>
<thead>
<tr>
<th>For More Information:</th>
</tr>
</thead>
</table>
As camera modules can be quite different, it is not advisable to hardcode the video resolution. Instead, it is a good practice to query the available resolutions of your camera sensor and only use the most optimal resolution for your application, if it is supported.

Let's say, you have predefined the video width you want in the mDesiredCameraPreviewWidth variable. You can then check if the value of the width resolution (and an associated video height) is supported by the camera using the following method:

```java
private void initializeCameraParameters() {
    Camera.Parameters parameters = mCamera.getParameters();
    List<Camera.Size> sizes = parameters.getSupportedPreviewSizes();
    int currentWidth = 0;
    int currentHeight = 0;
    boolean foundDesiredWidth = false;
    for (Camera.Size s : sizes) {
        if (s.width == mDesiredCameraPreviewWidth) {
            currentWidth = s.width;
            currentHeight = s.height;
            foundDesiredWidth = true;
            break;
        }
    }
    if (foundDesiredWidth) {
        parameters.setPreviewSize(currentWidth, currentHeight);
        mCamera.setParameters(parameters);
    }
}
```

The mCamera.getParameters() method is used to query the current camera parameters. The mCamera.getParameters() and getSupportedPreviewSizes() methods return the subset of available preview sizes and the parameters. setPreviewSize method is setting the new preview size. Finally, you have to call the mCamera.setParameters(parameters) method so that the requested changes are implemented. This initializeCameraParameters() method can then also be called in the onResume() method of your Activity class.

**Creating SurfaceView**

For your Augmented Reality application, you want to display a stream of live images from your back-facing camera on the screen. In a standard application, acquiring the video and displaying the video are two independent procedures. With the Android API, you explicitly need to have a separate SurfaceView to display the camera stream as well. The SurfaceView class is a dedicated drawing area that you can embed into your application.
So for our example, we need to derive a new class from the Android `SurfaceView` class (let's call it `CameraPreview`) and implement a `SurfaceHolder.Callback` interface. This interface is used to react to any events related to the surface, such as the creation, change, and destruction of the surface. Accessing the mobile camera is done through the `Camera` class. In the constructor, the Android `Camera` instance (defined previously) is passed:

```java
public class CameraPreview extends SurfaceView implements SurfaceHolder.Callback {
    private static final String TAG = "CameraPreview";
    private SurfaceHolder mHolder;
    private Camera mCamera;
    public CameraPreview(Context context, Camera camera) {
        super(context);
        mCamera = camera;
        mHolder = getHolder();
        mHolder.addCallback(this);
        mHolder.setType(SurfaceHolder.SURFACE_TYPE_PUSH_BUFFERS);
    }
}
```

In the `surfaceChanged` method, you take care of passing an initialized `SurfaceHolder` instance (that is the instance that holds the display surface) and starting the preview stream of the camera, which you later want to display (and process) in your own application. The stopping of the camera preview stream is important as well:

```java
public void surfaceChanged(SurfaceHolder holder, int format, int w, int h) {
    if (mHolder.getSurface() == null) {
        return;
    }
    try {
        mCamera.stopPreview();
    } catch (Exception e) {
    }
    try {
        mCamera.setPreviewDisplay(mHolder);
        mCamera.startPreview();
    } catch (Exception e) {
    }
}
```

The inherited methods, `surfaceCreated()` and `surfaceDestroyed()`, remain empty.

For More Information:
Having our CameraPreview class defined, we can declare it in the Activity class:

```java
private CameraPreview mPreview;
```

Then, instantiate it in the onResume() method:

```java
mPreview = new CameraPreview(this, mCamera);
setContentView(mPreview);
```

To test your application, you can do the same with your other project: please connect your testing device to your computer via a USB cable. In Eclipse, right-click on your project folder, CameraAccessAndroid, and in the pop-up menu go to Run As | Android Application. You should now be able to see the live camera view on your mobile screen as soon as the application is uploaded and started.

## Live camera view in JME

In the preceding example, you got a glimpse of how you can access the Android camera with a low-level graphics library (standard Android library). Since we want to perform Augmented Reality, we will need to have another technique to overlay the virtual content over the video view. There are different ways to do that, and the best method is certainly to use a common view, which will integrate the virtual and video content nicely. A powerful technique is to use a managed 3D graphics library based on a scenegraph model. A scenegraph is basically a data structure that helps you to build elaborate 3D scenes more easily than in plain OpenGL® by logically organizing basic building blocks, such as geometry or spatial transformations. As you installed JME in the first chapter, we will use this specific library offering all the characteristics we need for our AR development. In this subsection, we will explore how you can use JME to display the video. Different to our preceding example, the camera view will be integrated to the 3D scenegraph. In order to achieve this, you use the following steps:

1. Create a project with JME support.
2. Create the activity which sets up JME.
3. Create the JME application, which does the actual rendering of our 3D scene.

For creating the project with JME, you can follow the instructions in the Installing JMonkeyEngine section of Chapter 1, Augmented Reality Concepts and Tools. We will make a new project called CameraAccessJME.
Creating the JME activity

As an Android developer, you know that an Android activity is the main entry point to create your applications. However, JME is a platform-independent game engine that runs on many platforms with Java support. The creators of JME wanted to ease the process of integrating existing (and new) JME applications into Android as easily as possible. Therefore, they explicitly differentiated between the JME applications, which do the actual rendering of the scene (and could be used on other platforms as well), and the Android specific parts in the JME activity for setting up the environment to allow the JME application to run. The way they achieved this is to have a specific class called AndroidHarness, which takes the burden off the developer to configure the Android activity properly. For example, it maps touch events on your screen to mouse events in the JME application. One challenge in this approach is to forward Android-specific events, which are not common to other platforms in the JME application. Don't worry, we will show you how to do this for the camera images.

The first thing you want to do is create an Android activity derived from the AndroidHarness class, which we will call the CameraAccessJMEActivity method. Just like the CameraAccessAndroidActivity class, it holds instances of the Camera and CameraPreview classes. In contrast, it will also hold an instance of your actual JME application (discussed in the next section of this chapter) responsible for rendering your scene. You did not yet provide an actual instance of the class but only the fully qualified path name. The instance of your class is constructed at runtime through a reflection technique in the AndroidHarness super class:

```java
public CameraAccessJMEActivity() {
    appClass = "com.ar4android.CameraAccessJME";
}
```

During runtime, you can then access the actual instance by casting a general JME application class, which AndroidHarness stores in its app variable to your specific class, for example, through the (com.ar4android.CameraAccessJME) app.

As discussed at the beginning of this chapter, the camera can deliver the images in various pixel formats. Most rendering engines (and JME is no exception) cannot handle the wide variety of pixel formats but expect certain formats such as RGB565. The RGB565 format stores the red and blue components in 5 bits and the green component in 6 bits, thereby displaying 65536 colors in 16 bits per pixel. You can check if your camera supports this format in the initializeCameraParameters method by adding the following code:

```java
List<Integer> pixelFormats =
    parameters.getSupportedPreviewFormats();
for (Integer format : pixelFormats) {
```

For More Information:
if (format == ImageFormat.RGB_565) {
    pixelFormatConversionNeeded = false;
    parameters.setPreviewFormat(format);
    break;
}
}

In this code snippet, we query all available pixel formats (iterating over parameters.getSupportedPreviewFormats()) and set the pixel format of the RGB565 model if supported (and remember that we did this by setting the flag pixelFormatConversionNeeded).

As mentioned before, in contrast to the previous example, we will not directly render the SurfaceView class. Instead, we will copy the preview images from the camera in each frame. To prepare for this, we define the preparePreviewCallbackBuffer() method, which you will call in the onResume() method after creating your camera and setting its parameters. It allocates buffers to copy the camera images and forwarding it to JME:

```
public void preparePreviewCallbackBuffer() {
    mPreviewWidth = mCamera.getParameters().getPreviewSize().width;
    mPreviewHeight = mCamera.getParameters().
        getPreviewSize().height;
    int bufferSizeRGB565 = mPreviewWidth * mPreviewHeight * 2 + 4096;
    mPreviewBufferRGB565 = null;
    mPreviewBufferRGB565 = new byte[bufferSizeRGB565];
    mPreviewByteBufferRGB565 = ByteBuffer.allocateDirect(mPreviewBufferRGB565.length);
    cameraJMEImageRGB565 = new Image(Image.Format.RGB565,
        mPreviewWidth, mPreviewHeight, mPreviewByteBufferRGB565);
}
```

If your camera does not support RGB565, it may deliver the frame in the YCbCr format (Luminance, blue difference, red difference), which you have to convert to the RGB565 format. To do that, we will use a color space conversion method, which is really common in AR and for image processing. We provide an implementation of this method (yCbCrToRGB565(...) available in the sample project. A basic approach to use this method is to create different image buffers, where you will copy the source, intermediate, and final transformed image.

For More Information:
So for the conversion, the `mPreviewWidth`, `mPreviewHeight`, and `bitsPerPixel` variables are queried by calling the `getParameters()` method of your camera instance in the `preparePreviewCallbackBuffer()` method and determine the size of your byte arrays holding the image data. You will pass a JME image `(cameraJMEImageRGB565)` to the JME application, which is constructed from a Java `ByteBuffer` class, which itself just wraps the RGB565 byte array.

Having prepared the image buffers, we now need to access the content of the actual image. In Android, you do this by an implementation of the `Camera.PreviewCallback` interface. In the `onPreviewFrame(byte[] data, Camera c)` method of this object, you can get access to the actual camera image stored as a byte array:

```java
private final Camera.PreviewCallback mCameraCallback = new Camera.PreviewCallback() {
    public void onPreviewFrame(byte[] data, Camera c) {

        mPreviewByteBufferRGB565.clear();
        if (pixelFormatConversionNeeded) {
            yCbCrToRGB565(data, mPreviewWidth, mPreviewHeight, mPreviewBufferRGB565);
            mPreviewByteBufferRGB565.put(mPreviewBufferRGB565);
        }

        cameraJMEImageRGB565.setData(mPreviewByteBufferRGB565);
        if ((com.ar4android.CameraAccessJME) app != null) {
            ((com.ar4android.CameraAccessJME) app).setTexture(cameraJMEImageRGB565);
        }
    }
}
```

The `setTexture` method of the `CameraAccessJME` class simply copies the incoming data into a local image object.

Finally, you register your implementation of the `Camera.PreviewCallback` interface in the `onSurfaceChanged()` method of the `CameraPreview` class:

```java
mCamera.setPreviewCallback(mCameraPreviewCallback);
```
A faster method to retrieve the camera images, which avoids creating a new buffer in each frame, is to allocate a buffer before and use it with the methods, `mCamera.addCallbackBuffer()` and `mCamera.setPreviewCallbackWithBuffer()`. Please note that this approach might be incompatible with some devices.

Creating the JME application

As mentioned in the preceding section, the JME application is the place where the actual rendering of the scene takes place. It should not concern itself with the nitty-gritty details of the Android system, which were described earlier. JME provides you with a convenient way to initialize your application with many default settings. All you have to do is inherit from the `SimpleApplication` class, initialize your custom variables in `simpleInitApp()`, and eventually update them before a new frame is rendered in the `simpleUpdate()` method. For our purpose of rendering the camera background, we will create a custom `ViewPort` (a view inside the display window), and a virtual `Camera` (for rendering the observed scene), in the `initVideoBackground` method. The common method to display the video in a scene graph such as JME is to use the video image as a texture, which is placed on a quadrilateral mesh:

```java
public void initVideoBackground(int screenWidth, int screenHeight)
{
    Quad videoBGQuad = new Quad(1, 1, true);
    mVideoBGGeom = new Geometry("quad", videoBGQuad);
    float newWidth = 1.f * screenWidth / screenHeight;
    mVideoBGGeom.setLocalTranslation(-0.5f * newWidth, -0.5f, 0.f);
    mVideoBGGeom.setLocalScale(1.f * newWidth, 1.f, 1);
    mVideoBGMat = new Material(assetManager,
        "Common/MatDefs/Misc/Unshaded.j3md");
    mVideoBGGeom.setMaterial(mvideoBGMat);
    mCameraTexture = new Texture2D();
    Camera videoBGCam = cam.clone();
    videoBGCam.setParallelProjection(true);
    ViewPort videoBGVP = renderManager.createMainView("VideoBGView",
        videoBGCam);
    videoBGVP.attachScene(mVideoBGGeom);
    mSceneInitialized = true;
}
```

For More Information:
Let's have a more detailed look at this essential method for setting up our scenegraph for the rendering of the video background. You first create a quad shape and assign it to a JME Geometry object. To assure correct mapping between the screen and the camera, you scale and reposition the geometry according to the dimensions of the device's screen. You assign a material to the quad and also create a texture for it. Since we are doing 3D rendering, we need to define the camera looking at this quad. As we want the camera to only see the quad nicely placed in front of the camera without distortion, we create a custom viewport and an orthographic camera (this orthographic camera has no perspective foreshortening). Finally, we add the quad geometry to this viewport.

Now, we have our camera looking at the textured quad rendered fullscreen. All that is left to do is update the texture of the quad each time a new video frame is available from the camera. We will do this in the simpleUpdate() method, which is called regularly by the JME rendering engine:

```java
public void simpleUpdate(float tpf) {
if (mNewCameraFrameAvailable) {
   mCameraTexture.setImage(mCameraImage);
   mvideoBGMat.setTexture("ColorMap", mCameraTexture);
}
}
```

You may have noted the usage of the conditional test on the mNewCameraFrameAvailable variable. As the scenegraph renders its content with a different refresh rate (up to 60 fps, on a modern smartphone) than what a mobile camera can normally deliver (typically 20-30 fps), we use the mNewCameraFrameAvailable flag to only update the texture if a new image becomes available.

So this is it. With these steps implemented, you can compile and upload your application and should get a similar result as shown in the following figure:

For More Information:
Summary

In this chapter you got an introduction to the world of Android camera access and how to display camera images in the JME 3D rendering engine. You learned about various camera parameters and the compromises you have made (for example, between image size and frames per second) to get an efficient camera access. We also introduced the simplest way of displaying a camera view in an Android activity, but also explained why you need to go beyond this simple example to integrate the camera view and 3D graphics in a single application. Finally, we helped you through the implementation of a JME application, which renders the camera background. The knowledge you gained in this chapter is the beneficial basis to overlay the first 3D objects on the camera view—a topic we will discuss in the next chapter.
Where to buy this book


Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.