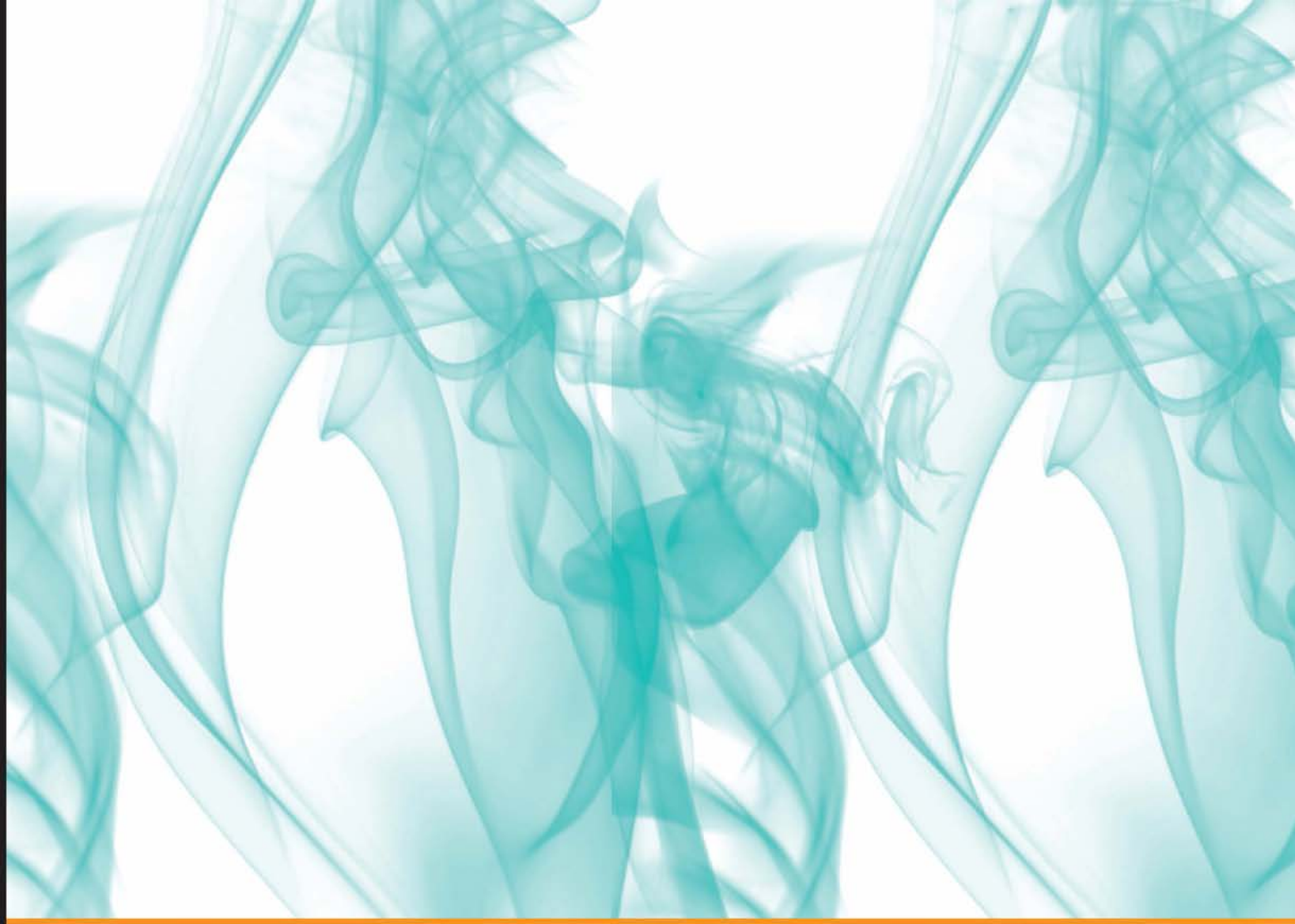


Community Experience Distilled

Augmented Reality for Android Application Development

Learn how to develop advanced Augmented Reality applications for Android

Jens Grubert
Dr. Raphael Grasset

[PACKT] PUBLISHING open source* community experience distilled

Augmented Reality for Android Application Development

Learn how to develop advanced Augmented Reality applications for Android

Jens Grubert

Dr. Raphael Grasset

[PACKT] open source
community experience distilled
PUBLISHING

BIRMINGHAM - MUMBAI

Augmented Reality for Android Application Development

First published: November 2013

Production Reference: 1191113

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78216-855-3

www.packtpub.com

Cover Image by Suresh Mogre (suresh.mogre.99@gmail.com)

Credits

Authors

Jens Grubert

Dr. Raphael Grasset

Reviewers

Peter Backx

Glauco Márdano

Acquisition Editor

Kunal Parikh

Owen Roberts

Commissioning Editor

Poonam Jain

Technical Editors

Monica John

Siddhi Rane

Sonali Vernekar

Copy Editors

Brandt D'Mello

Sarang Chari

Tanvi Gaitonde

Gladson Monteiro

Sayanee Mukherjee

Adithi Shetty

Project Coordinator

Sherin Padayatty

Proofreader

Simran Bhogal

Indexer

Rekha Nair

Production Coordinator

Alwin Roy

Cover Work

Alwin Roy

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`.

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

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`.

About the Reviewers

Peter Backx has an MoS and a PhD. in Computer Sciences from Ghent University. He is a software developer and architect. He uses technology to shape unique user experiences and build rock-solid, scalable software.

Peter works as a freelance consultant at `www.peated.be` and shares his knowledge and experiments on his blog `www.streamhead.com`.

Glauco Márdano is a 22-year-old who lives in Brazil and has a degree in Systems Analysis. He has worked for two years as a Java web programmer and he is now studying and getting certified in Java.

He has reviewed the *jMonkeyEngine 3.0 Beginners Guide* book.

I'd like to thank all from the jMonkeyEngine forum because I've learnt a lot of new things since I came across the forum and I'm very grateful for their support and activity. I'd like to thank the guys from Packt Publishing, too, and I'm very pleased to be a reviewer for this book.

---

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`http://PacktLib.PacktPub.com`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

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 Vuforia™ 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.

What you need for this book

If you want to develop Augmented Reality applications for Android, you can share a majority of tools with regular Android developers. Specifically, you can leverage the power of the widely supported **Android Developer Tools Bundle** (**ADT Bundle**). This includes:

- The Eclipse **Integrated Development Environment** (**IDE**)
- The **Android Developer Tools** (**ADT**) plugin for Eclipse
- The Android platform for your targeted devices (further platforms can be downloaded)
- The Android emulator with the latest system image

Besides this standard package common to many Android development environments, you will need:

- A snapshot of **JMonkeyEngine** (**JME**), Version 3 or higher
- **Qualcomm® Vuforia™ SDK** (**Vuforia™**)**,** version 2.6 or higher
- **Android Native Development Kit** (**Android NDK**), version r9 or higher

Who this book is for

If you are a mobile application developer for Android and want to get to the next level of mobile app development using Augmented Reality, then this book is for you. It is assumed that you are familiar with Android development tools and deployment. It is beneficial if you have experience on working with external libraries for Android, as we make use of JMonkeyEngine and the Vuforia™ SDK. If you have already used the Android NDK, then this is great but not mandatory.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "Finally, you register your implementation of the `Camera.PreviewCallback` interface in the `onSurfaceChanged()` method of the `CameraPreview` class."

A block of code is set as follows:

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

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: " in the pop-up menu, go to **Run As | 1 Android Application**."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

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`.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
Augmented Reality Concepts and Tools

Augmented Reality (**AR**) offers us a new way to interact with the physical (or real) world. It creates a modified version of our reality, enriched with digital (or virtual) information, on the screen of your desktop computer or mobile device. Merging and combining the virtual and the real can leverage a totally new range of user experience, going beyond what common apps are capable of. Can you imagine playing a first-person shooter in your own neighborhood, with monsters popping up at the corner of your street (as it is possible with ARQuake by *Bruce Thomas* at the University of South Australia, see left-hand side of the following screenshot)? Will it not be a thrilling moment to go to a natural history museum and see a dusty dinosaur skeleton coming virtually alive—flesh and bone—in front of your eyes? Or can you imagine reading a story to your kid and seeing some proud rooster appear and walk over the pages of a book (as it is possible with the AR version of the "House that Jack Built" written by *Gavin Bishop*, see the right-hand side of the following screenshot). In this book, we show you how to practically implement such experiences on the Android platform.

A decade ago, experienced researchers would have been among the few who were able to create these types of applications. They were generally limited to demonstration prototypes or in the production of an ad hoc project running for a limited period of time. Now, developing AR experiences has become a reality for a wide range of mobile software developers. Over the last few years, we have been spectators to great progresses in computational power, the miniaturization of sensors, as well as increasingly accessible and featured multimedia libraries. These advances allow developers to produce AR applications more easily than ever before. This already leads to an increasing number of AR applications flourishing on mobile app stores such as Google Play. While an enthusiastic programmer can easily stitch together some basic code snippets to create a facsimile of a basic AR application, they are generally poorly designed, with limited functionalities, and hardly reusable. To be able to create sophisticated AR applications, one has to understand what Augmented Reality truly is.

In this chapter, we will guide you toward a better understanding of AR. We will describe some of the major concepts of AR. We will then move on from these examples to the foundational software components for AR. Finally, we will introduce the development tools that we will use throughout this book, which will support our journey into creating productive and modular AR software architecture.

Ready to change your reality for Augmented Reality? Let's start.

A quick overview of AR concepts

As AR has become increasingly popular in the media over the last few years, unfortunately, several distorted notions of Augmented Reality have evolved. Anything that is somehow related to the real world and involves some computing, such as standing in front of a shop and watching 3D models wear the latest fashions, has become AR. Augmented Reality emerged from research labs a few decades ago and different definitions of AR have been produced. As more and more research fields (for example, computer vision, computer graphics, human-computer interaction, medicine, humanities, and art) have investigated AR as a technology, application, or concept, multiple overlapping definitions now exist for AR. Rather than providing you with an exhaustive list of definitions, we will present some major concepts present in any AR application.

Sensory augmentation

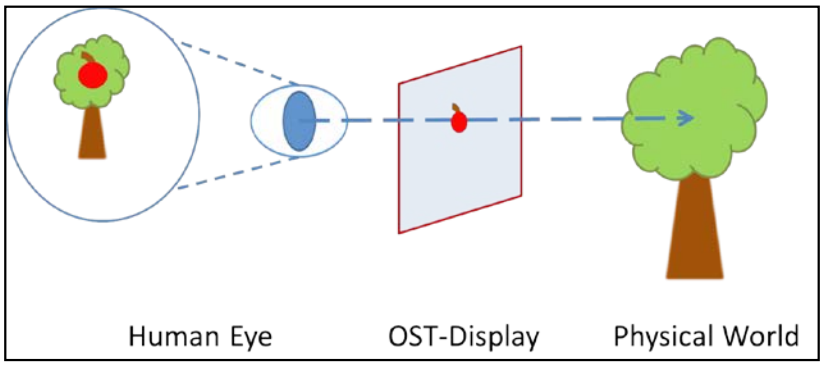
The term Augmented Reality itself contains the notion of reality. Augmenting generally refers to the aspect of influencing one of your human sensory systems, such as vision or hearing, with additional information. This information is generally defined as digital or virtual and will be produced by a computer. The technology currently uses **displays** to overlay and merge the physical information with the digital information. To augment your hearing, modified headphones or earphones equipped with microphones are able to mix sound from your surroundings in real-time with sound generated by your computer. In this book, we will mainly look at visual augmentation.

Displays

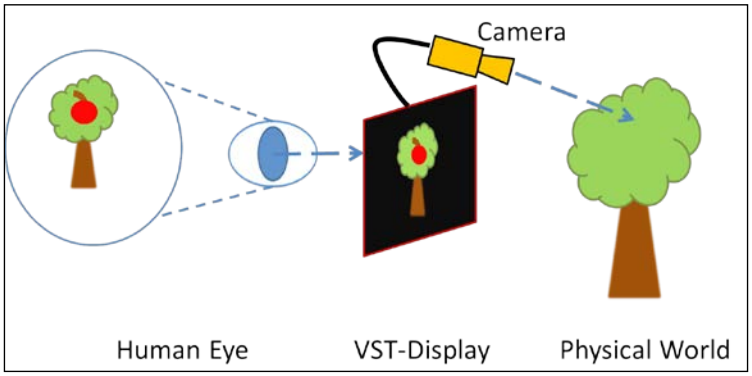
The TV screen at home is the ideal device to perceive virtual content, streamed from broadcasts or played from your DVD. Unfortunately, most common TV screens are not able to capture the real world and augment it. An Augmented Reality display needs to simultaneously show the real and virtual worlds.

One of the first display technologies for AR was produced by *Ivan Sutherland* in 1964 (named "The Sword of Damocles"). The system was rigidly mounted on the ceiling and used some CRT screens and a transparent display to be able to create the sensation of visually merging the real and virtual.

Since then, we have seen different trends in AR display, going from static to wearable and handheld displays. One of the major trends is the usage of **optical see-through** (**OST**) technology. The idea is to still see the real world through a semi-transparent screen and project some virtual content on the screen. The merging of the real and virtual worlds does not happen on the computer screen, but directly on the retina of your eye, as depicted in the following figure:



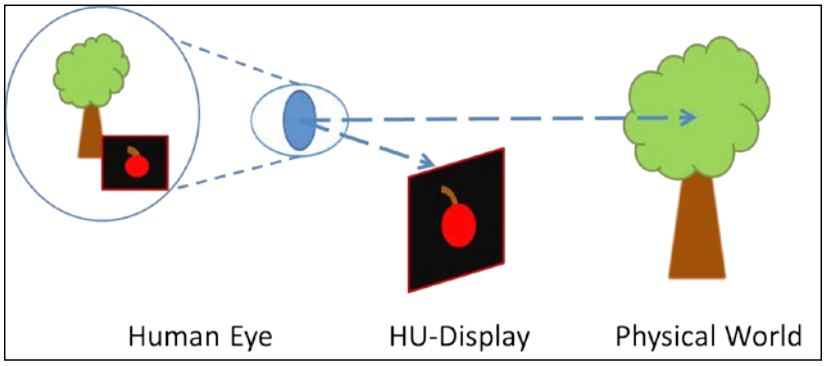
The other major trend in AR display is what we call **video see-through** (**VST**) technology. You can imagine perceiving the world not directly, but through a video on a monitor. The video image is mixed with some virtual content (as you will see in a movie) and sent back to some standard display, such as your desktop screen, your mobile phone, or the upcoming generation of head-mounted displays as shown in the following figure:



In this book, we will work on Android-driven mobile phones and, therefore, discuss only VST systems; the video camera used will be the one on the back of your phone.

Registration in 3D

With a display (OST or VST) in your hands, you are already able to superimpose things from your real world, as you will see in TV advertisements with text banners at the bottom of the screen. However, any virtual content (such as text or images) will remain fixed in its position on the screen. The superposition being really static, your AR display will act as a **head-up display** (**HUD**), but won't really be an AR as shown in the following figure:



Google Glass is an example of an HUD. While it uses a semitransparent screen like an OST, the digital content remains in a static position.

AR needs to know more about real and virtual content. It needs to know where things are in space (**registration**) and follow where they are moving (**tracking**).

Registration is basically the idea of aligning virtual and real content in the same space. If you are into movies or sports, you will notice that 2D or 3D graphics are superimposed onto scenes of the physical world quite often. In ice hockey, the puck is often highlighted with a colored trail. In movies such as *Walt Disney's* TRON (1982 version), the real and virtual elements are seamlessly blended. However, AR differs from those effects as it is based on all of the following aspects (proposed by *Ronald T. Azuma* in 1997):

- **It's in 3D**: In the olden days, some of the movies were edited manually to merge virtual visual effects with real content. A well-known example is Star Wars, where all the lightsaber effects have been painted by hand by hundreds of artists and, thus, frame by frame. Nowadays, more complex techniques support merging digital 3D content (such as characters or cars) with the video image (and is called match moving). AR is inherently always doing that in a 3D space.
- **The registration happens in real time**: In a movie, everything is pre-recorded and generated in a studio; you just play the media. In AR, everything is in real time, so your application needs to merge, at each instance, reality and virtuality.
- **It's interactive**: In a movie, you only look passively at the scene from where it has been shot. In AR, you can actively move around, forward, and backward and turn your AR display—you will still see an alignment between both worlds.

Interaction with the environment

Building a rich AR application needs interaction between environments; otherwise you end up with pretty, 3D graphics that can turn boring quite fast. AR interaction refers to selecting and manipulating digital and physical objects and navigating in the augmented scene. Rich AR applications allow you to use objects which can be on your table, to move some virtual characters, use your hands to select some floating virtual objects while walking on the street, or speak to a virtual agent appearing on your watch to arrange a meeting later in the day. In *Chapter 6*, *Make It Interactive – Create the User Experience*, we will discuss mobile-AR interaction. We will look at how some of the standard mobile interaction techniques can also be applied to AR. We will also dig into specific techniques involving the manipulation of the real world.

Choose your style – sensor-based and computer vision-based AR

Previously in this chapter, we discussed what AR is and elaborated on display, registration, and interaction. As some of the notions in this book can also be applied to any AR development, we will specifically look at **mobile AR**.

Mobile AR sometimes refers to any transportable, wearable AR system that can be used indoors and outdoors. In this book, we will look at mobile AR with the most popular connotation used today—using handheld mobile devices, such as smartphones or tablets. With the current generation of smartphones, two major approaches to the AR system can be realized. These systems are characterized by their specific registration techniques and, also, their interaction range. They both enable a different range of applications. The systems, sensor-based AR and computer vision-based AR, are using the video see-through display, relying on the camera and screen of the mobile phone.

Sensor-based AR

The first type of system is called sensor-based AR and generally referred to as a GPS plus inertial AR (or, sometimes, outdoor AR system). Sensor-based AR uses the location sensor from a mobile as well as the orientation sensor. Combining both the location and orientation sensors delivers the global position of the user in the physical world.

The location sensor is mainly supported with a **GNSS** (**Global Navigation Satellite System**) receiver. One of the most popular GNSS receivers is the GPS (maintained by the USA), which is present on most smartphones.

Other systems are currently (or will soon be) deployed, such as GLONASS (Russia), Galileo (Europe, 2020), or Compass (China, 2020).

There are several possible orientation sensors available on handheld devices, such as accelerometers, magnetometers, and gyroscopes. The measured position and orientation of your handheld device provides tracking information, which is used for registering virtual objects on the physical scene. The position reported by the GPS module can be both inaccurate and updated slower than you move around. This can result in a **lag,** that is, when you do a fast movement, virtual elements seem to float behind. One of the most popular types of AR applications with sensor-based systems are AR browsers, which visualize **Points of Interests** (**POIs**), that is, simple graphical information about things around you. If you try some of the most popular products such as Junaio, Layar, or Wikitude, you will probably observe this effect of lag.

The advantage of this technique is that the sensor-based ARs are working on a general scale around the world, in practically any physical outdoor position (such as if you are in the middle of the desert or in a city). One of the limitations of such systems is their inability to work inside (or work poorly) or in any occluded area (no line-of-sight with the sky, such as in forests or on streets with high buildings all around). We will discuss more about this type of mobile AR system in *Chapter 4*, *Locating in the World*.

Computer vision-based AR

The other popular type of AR system is computer vision-based AR. The idea here is to leverage the power of the inbuilt camera for more than capturing and displaying the physical world (as done in sensor-based AR). This technology generally operates with image processing and computer vision algorithms that analyze the image to detect any object visible from the camera. This analysis can provide information about the position of different objects and, therefore, the user (more about that in *Chapter 5*, *Same as Hollywood – Virtual on Physical Objects*).

The advantage is that things seem to be perfectly aligned. The current technology allows you to recognize different types of planar pictorial content, such as a specifically designed marker (**marker-based tracking**) or more natural content (**markerless tracking**). One of the disadvantages is that vision-based AR is heavy in processing and can drain the battery really rapidly. Recent generations of smartphones are more adapted to handle this type of problem, being that they are optimized for energy consumption.

AR architecture concepts

So let's explore how we can support the development of the previously described concepts and the two general AR systems. As in the development of any other application, some well-known concepts of software engineering can be applied in developing an AR application. We will look at the structural aspect of an AR application (software components) followed by the behavioral aspect (control flow).

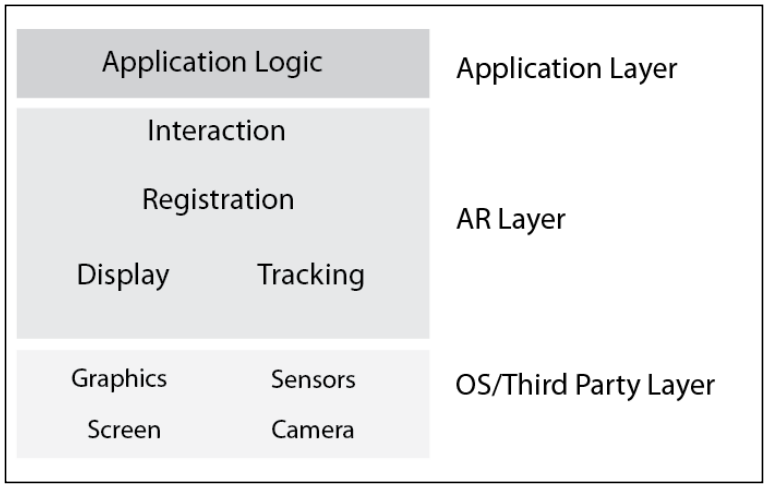
AR software components

An AR application can be structured in three layers: the application layer, the AR layer, and the OS/Third Party layer.

The **application layer** corresponds to the domain logic of your application. If you want to develop an AR game, anything related to managing the game assets (characters, scenes, objects) or the game logic will be implemented in this specific layer. The AR layer corresponds to the instantiation of the concepts we've previously described. Each of the AR notions and concepts that we've presented (display, registration, and interaction) can be seen, in terms of software, as a modular element, a component, or a service of the AR layer.

You can note that we have separated tracking from registration in the figure, making tracking one major software component for an AR application. Tracking, which provides spatial information to the registration service, is a complex and computationally intensive process in any AR application. The OS/Third Party layer corresponds to existing tools and libraries which don't provide any AR functionalities, but will enable the AR layer. For example, the **Display** module for a mobile application will communicate with the OS layer to access the camera to create a view of the physical world. On Android, the Google Android API is part of this layer. Some additional libraries, such as JMonkeyEngine, which handle the graphics, are also part of this layer.

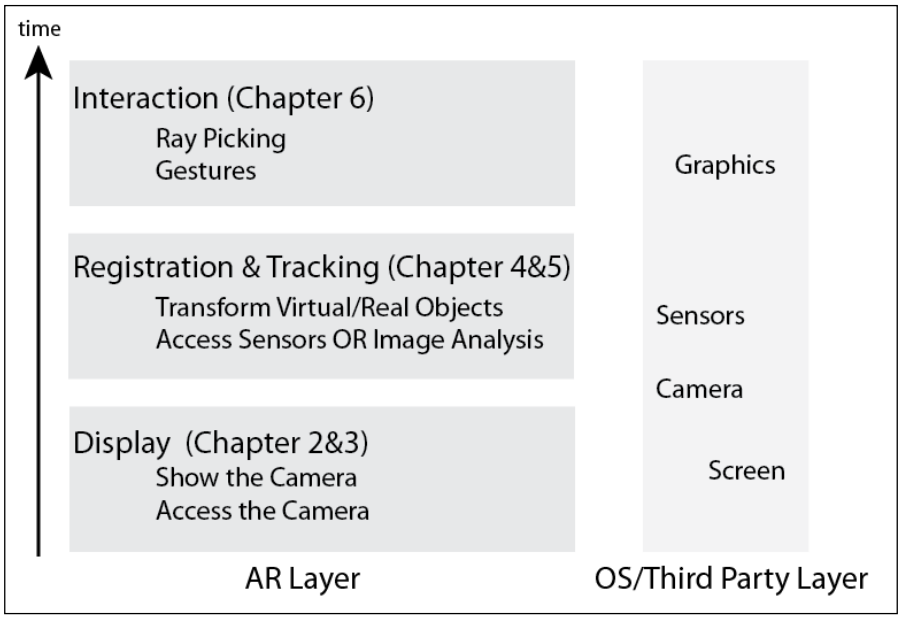
In the rest of the book, we will show you how to implement the different modules of the AR layer, which also involves communication with the OS/Third Party layer. The major layers of an AR application (see the right-hand side of the following figure), with their application modules (the left-hand side of the following figure), are depicted in the following figure:



AR control flow

With the concept of software layers and components in mind, we can now look at how information will flow in a typical AR application. We will focus here on describing how each of the components of the AR layer relate to each other over time and what their connections with the OS/Third Party layer are.

Over the last decade, AR researchers and developers have converged toward a well-used method of combining these components using a similar order of execution—the AR control flow. We present here the general AR control flow used by the community and summarized in the following figure:



The preceding figure, read from the bottom up, shows the main activities of an AR application. This sequence is repeated indefinitely in an AR application; it can be seen as the typical **AR main loop** (please note that we've excluded the domain logic here as well as the OS activities). Each activity corresponds to the same module we've presented before. The structure of the AR layer and AR control flow is, therefore, quite symmetric.

Understand that this control flow is the key to developing an AR application, so we will come back to it and use it in the rest of the book. We will get into more details of each of the components and steps in the next chapter.

So, looking at the preceding figure, the main activities and steps in your application are as follows:

- **Manage the display first**: For mobile AR, this means accessing the video camera and showing a captured image on the screen (a view of your physical world). We will discuss that in *Chapter 2*, *Viewing the World*. This also involves matching camera parameters between the physical camera and the virtual one that renders your digital objects (*Chapter 3*, *Superimposing the World*).

- **Register and track your objects**: Analyze the sensors on your mobile (approach 1) or analyze the video image (approach 2) and detect the position of each element of your world (such as camera or objects). We will discuss this aspect in *Chapter 4*, *Locating in the World* and *Chapter 5*, *Same as Hollywood – Virtual on Physical Objects*.
- **Interact**: Once your content is correctly registered, you can start to interact with it, as we will discuss in *Chapter 6*, *Make It Interactive – Create the User Experience*.

System requirements for development and deployment

If you want to develop Augmented Reality applications for Android, you can share the majority of tools with regular Android developers. Specifically, you can leverage the power of the widely supported **Google Android Developer Tools Bundle** (**ADT Bundle**). This includes the following:

- The Eclipse **Integrated Development Environment** (**IDE**)
- The **Google Android Developer Tools** (**ADT**) plugin for Eclipse
- The Android platform for your targeted devices (further platforms can be downloaded)
- The Android Emulator with the latest system image

Besides this standard package common to many Android development environments, you will need the following:

- A snapshot of **JMonkeyEngine** (**JME**), version 3 or higher
- **Qualcomm® Vuforia™ SDK** (**Vuforia™**), version 2.6 or higher
- **Android Native Development Kit** (**Android NDK**), version r9 or higher

The JME Java OpenGL® game engine is a free toolkit that brings the 3D graphics in your programs to life. It provides 3D graphics and gaming middleware that frees you from exclusively coding in low-level **OpenGL® ES** (**OpenGL® for Embedded Systems**), for example, by providing an asset system for importing models, predefined lighting, and physics and special effects components.

The Qualcomm® Vuforia™ SDK brings state-of-the art computer vision algorithms targeted at recognizing and tracking a wide variety of objects, including fiducials (frame markers), image targets, and even 3D objects. While it is not needed for sensor-based AR, it allows you to conveniently implement computer vision-based AR applications.

The Google Android NDK is a toolset for performance-critical applications. It allows parts of the application to be written in native-code languages (C/C++). While you don't need to code in C or C++, this toolset is required by Vuforia™ SDK.

Of course, you are not bound to a specific IDE and can work with command-line tools as well. The code snippets themselves, which we present in this book, do not rely on the use of a specific IDE. However, within this book, we will give you setup instructions specifically for the popular Eclipse IDE. Furthermore, all development tools can be used on Windows (XP or later), Linux, and Mac OS X (10.7 or later).

On the next pages, we will guide you through the installation processes of the Android Developer Tools Bundle, NDK, JME, and Vuforia™ SDK. While the development tools can be spread throughout the system, we recommend that you use a common base directory for both the development tools and the sample code; let's call it `AR4Android` (for example, `C:/AR4Android` under Windows or `/opt/AR4Android` under Linux or Mac OS X).

Installing the Android Developer Tools Bundle and the Android NDK

You can install the ADT Bundle in two easy steps as follows:

1. Download the ADT Bundle from `http://developer.android.com/sdk/index.html`.
2. After downloading, unzip `adt-bundle-<os_platform>.zip` into the `AR4Android` base directory.

You can then start the Eclipse IDE by launching `AR4Android/adt-bundle-<os_platform>/eclipse/eclipse(.exe)`.

Please note that you might need to install additional system images, depending on the devices you use (for example, version 2.3.5, or 4.0.1). You can follow the instructions given at the following website: `http://developer.android.com/tools/help/sdk-manager.html`.

For the Android NDK (version r9 or higher), you follow a similar procedure as follows:

1. Download it from `http://developer.android.com/tools/sdk/ndk/index.html`.
2. After downloading, unzip `android-ndk-r<version>Y-<os_platform>.(zip|bz2)` into the `AR4Android` base directory.

Installing JMonkeyEngine

JME is a powerful Java-based 3D game engine. It comes with its own development environment (JME IDE based on NetBeans) which is targeted towards the development of desktop games. While the JME IDE also supports the deployment of Android devices, it (at the time this book is being written) lacks the integration of convenient Android SDK tools like the **Android Debug Bridge** (**adb**), **Dalvik Debug Monitor Server view** (**DDMS**) or integration of the Android Emulator found in the ADT Bundle. So, instead of using the JME IDE, we will integrate the base libraries into our AR projects in Eclipse. The easiest way to obtain the JME libraries is to download the SDK for your operating system from `http://jmonkeyengine.org/downloads` and install it into the `AR4Android` base directory (or your own developer directory; just make sure you can easily access it later in your projects). At the time this book is being published, there are three packages: Windows, GNU/Linux, and Mac OS X.

You can also obtain most recent versions from `http://updates.jmonkeyengine.org/nightly/3.0/engine/`

You need only the Java libraries of JME (`.jar`) for the AR development, using the ADT Bundle. If you work on Windows or Linux, you can include them in any existing Eclipse project by performing the following steps:

1. Right-click on your AR project (which we will create in the next chapter) or any other project in the Eclipse explorer and go to **Build Path** | **Add External Archives**.
2. In the **JAR selection** dialog, browse to `AR4Android/jmonkeyplatform/jmonkeyplatform/libs`.
3. You can select **all JARs** in the lib directory and click on **Open**.

If you work on Mac OS X, you should extract the libraries from `jmonkeyplatform.app` before applying the same procedure as for Windows or Linux described in the preceding section. To extract the libraries, you need to right-click on your `jmonkeyplatform.app` app and select **Show Package contents** and you will find the libraries in `/Applications/jmonkeyplatform.app/Contents/Resources/`.

Please note that, in the context of this book, we only use a few of them. In the Eclipse projects accompanying the source code of the book, you will find the necessary JARs already in the local lib directories containing the subset of Java libraries necessary for running the examples. You can also reference them in your build path.

The reference documentation for using JME with Android can be found at `http://hub.jmonkeyengine.org/wiki/doku.php/jme3:android`.

Installing Vuforia™

Vuforia™ is a state-of-the-art library for computer vision recognition and natural feature tracking.

In order to download and install Vuforia™, you have to initially register at `https://developer.vuforia.com/user/register`. Afterwards, you can download the SDK (for Windows, Linux, or Mac OS X) from `https://developer.vuforia.com/resources/sdk/android`. Create a folder named `AR4Android/ThirdParty`. Now create an Eclipse project by going to **File | New | Project ...** named `ThirdParty` and choose as location the folder `AR4Android/ThirdParty` (see also the section *Creating an Eclipse project* in *Chapter 2*, *Viewing the World*). Then install the Vuforia™ SDK in `AR4Android/ThirdParty/vuforia-sdk-android-<VERSION>`. For the examples in *Chapter 5*, *Same as Hollywood – Virtual on Physical Objects* and *Chapter 6*, *Make It Interactive – Create the User Experience*, you will need to reference this `ThirdParty Eclipse` project.

Which Android devices should you use?

The Augmented Reality applications which you will learn to build will run on a wide variety of Android-powered smartphone and tablet devices. However, depending on the specific algorithms, we will introduce certain hardware requirements that should be met. Specifically, the Android device needs to have the following features:

- A back-facing camera for all examples in this book
- A GPS module for the sensor-based AR examples
- A gyroscope or linear accelerometers for the sensor-based AR examples

Augmented Reality on mobile phones can be challenging as many integrated sensors have to be active during the running of applications and computationally demanding algorithms are executed. Therefore, we recommend deploying them on a dual-core processor (or more cores) for the best AR experience. The earliest Android version to deploy should be 2.3.3 (API 10, Gingerbread). This gives potential outreach to your AR app across approximately 95 percent of all Android devices.

Visit `http://developer.android.com/about/dashboards/index.html` for up-to-date numbers.

Please make sure to set up your device for development as described at `http://developer.android.com/tools/device.html`.

In addition, most AR applications, specifically the computer-vision based applications (using Vuforia™), require enough processing power.

Summary

In this chapter, we introduced the foundational background of AR. We've presented some of the main concepts of AR, such as sensory augmentation, dedicated display technology, real-time spatial registration of physical and digital information, and interaction with the content.

We've also presented computer vision-based and sensor-based AR systems, the two major trends of architecture on mobile devices. The basic software architecture blocks of an AR application have also been described and will be used as a guide for the remaining presentation of this book. By now, you should have installed the third-party tools used in the coming chapters. In the next chapter, you will get started with viewing the virtual world and implementing camera access with JME.

2
Viewing the World

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.

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

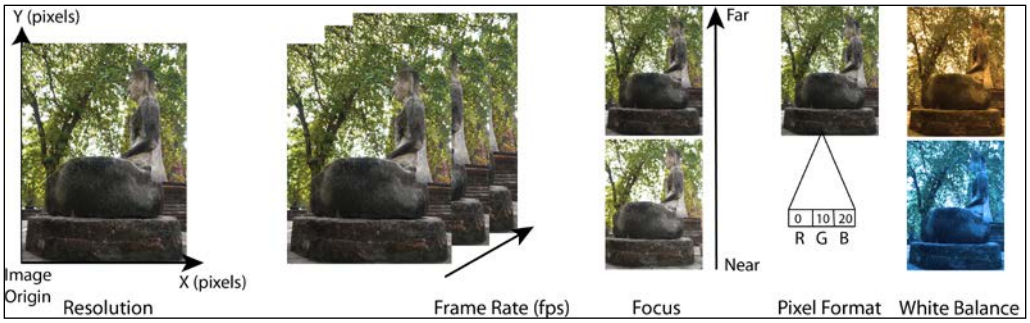
Cameras on smartphones nowadays share many characteristics with digital point-and-shoot cameras. They generally support two operative modes: the still image mode (which is an instantaneous, singular capture of an image), or the video mode (which is a continuous, real-time capture of images).

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).

- **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.

- 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).

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 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.

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

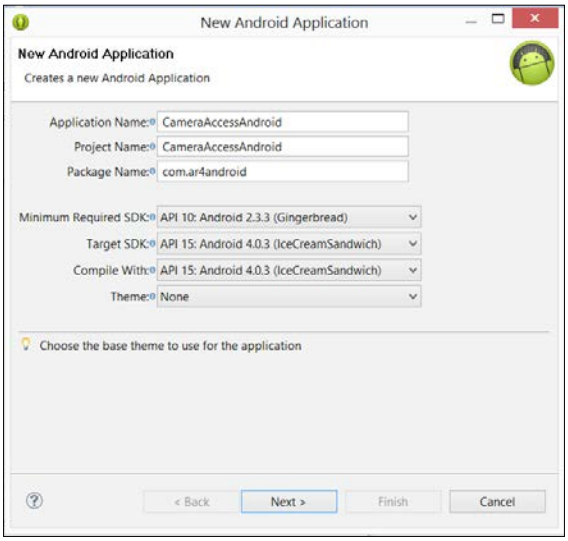
Downloading the example code

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:

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

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

```
<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:

```
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:

```
public class CameraAccessAndroidActivity extends Activity {
private Camera mCamera;

}
```

The next step is to open the camera. To do that, we define a `getCameraInstance()` method:

```
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:

```
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:

```
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).

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:

```
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 (lets 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:

```
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:

```
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.

Having our `CameraPreview` class defined, we can declare it in the `Activity` class:

```
private CameraPreview mPreview;
```

Then, instantiate it in the `onResume()` method:

```
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 | 1 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:

```
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:

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

```
      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.

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:

```
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:

```
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:

```
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;
}
```

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.

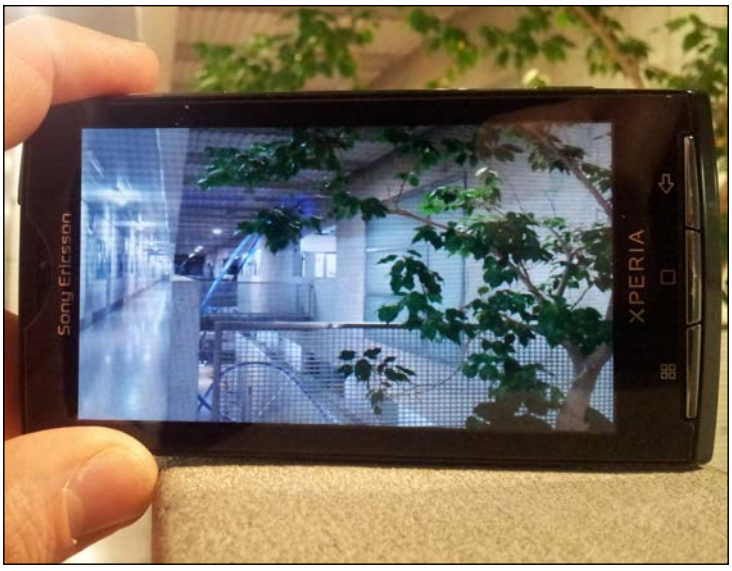
Now, we have our camera looking at the textured quad rendered fullscreen. All that is left to do is update the texture of the quad each time a new video frame is available from the camera. We will do this in the `simpleUpdate()` method, which is called regularly by the JME rendering engine:

```
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:

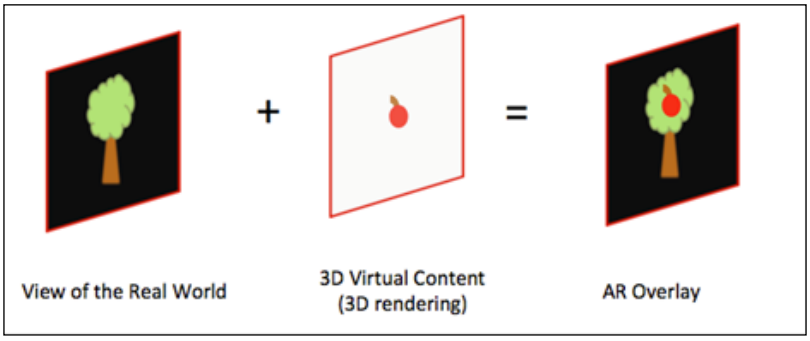


Summary

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.

3
Superimposing the World

Now that you have a view of the physical world on your screen, our next goal is to overlay digital 3D models on top of it. Overlay in 3D as used in Augmented Reality, is different from basic 2D overlays possible with Adobe Photoshop or similar drawing applications (in which we only adjust the position of two 2D layers). The notion of 3D overlay involves the management and rendering of content with six degrees of freedom (translation and rotation in three dimensions) as shown in the following figure:



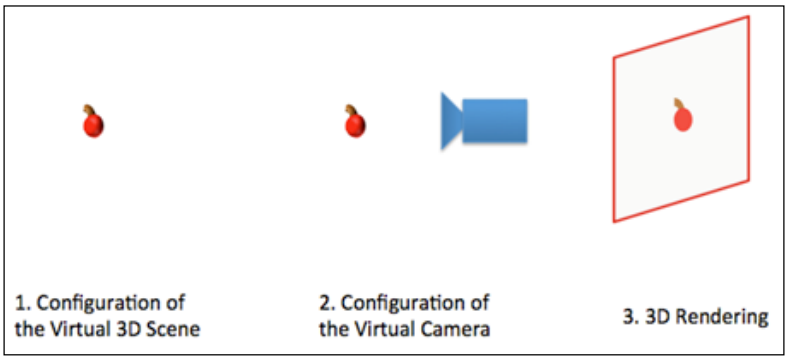
In this chapter, we will guide you through the different concepts and present you with the best way to superimpose real and virtual content. We will successively describe the concept of real and virtual cameras, how to perform superimposition with our scene graph engine, and create high quality superimposition. First, let's discuss the 3D world and the virtual camera.

The building blocks of 3D rendering

Representing and rendering virtual 3D content operates in the same way as when you click a picture with a digital camera in the physical world. If you take a picture of your friend or a landscape, you will first check your subject with the naked eye and after that will look at it through the viewfinder of the camera; only then will you take the picture. These three different steps are the same with virtual 3D content. You do not have a physical camera taking pictures, but you will use a **virtual camera** to render your scene. Your virtual camera can be seen as a digital representation of a real camera and can be configured in a similar way; you can position your camera, change its field of view, and so on. With virtual 3D content, you manipulate a digital representation of a geometrical 3D scene, which we simply call your virtual 3D scene or virtual world.

The three basic steps for rendering a scene using 3D computer graphics are shown in the following figure and consist of:

- Configuring your virtual 3D scene (objects position and appearance)
- Configuring your virtual camera
- Rendering the 3D scene with the virtual camera

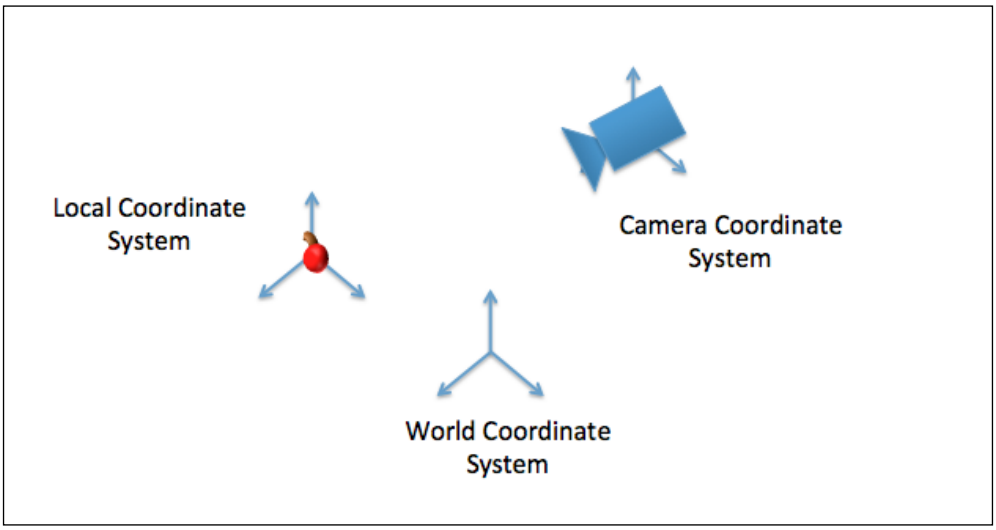


As we do real-time rendering for AR, you will repeat these steps in a loop; objects or cameras can be moved at each time frame (typically at 20-30 FPS).

While positioning objects in a scene, or the camera in a scene, we need a way of representing the location (and also the orientation) of objects as functions of each other. To do so, we generally use some spatial representation of the scene based on geometric mathematical models. The most common approach is to use **Euclidian geometry** and **coordinate systems**. A coordinate system defines a method of referencing an object (or point) in a space using a numerical representation to define this position (**coordinates**). Everything in your scene can be defined in a coordinate system, and coordinate systems can be related to each other using **transformations**.

The most common coordinate systems are shown in the following figure and are:

- **World Coordinate System**: It is the ground where you reference everything.
- **Camera Coordinate System**: It is placed in the world coordinate system and used to render your scene seen from this specific viewpoint. It is sometimes also referenced as the Eye Coordinate System.
- **Local Coordinate System(s)**: It is, for example, an object coordinate system, used to represent the 3D points of an object. Traditionally, you use the (geometric) center of your object to define your local coordinate system.



There are two conventions for the orientation of the coordinate systems: left-handed and right-handed. In both the conventions, X goes on the right-hand side and Y goes upwards. Z goes towards you in the right-handed convention and away from you in the left-handed convention.

Another common coordinate system, not illustrated here, is the image coordinate system. You are probably familiar with this one if you edit your pictures. It defines the position of each pixel of your image from a referenced origin (commonly the top-left corner or the bottom-left corner of an image). When you perform 3D graphics rendering, it's the same concept. Now we will focus on the virtual camera characteristics.

Real camera and virtual camera

A virtual camera for 3D graphics rendering is generally represented by two main sets of parameters: the **extrinsic** and **intrinsic** parameters. The extrinsic parameters define the location of the camera in the virtual world (the transformation from the world coordinate system to the camera coordinate system and vice versa). The intrinsic parameters define the projective properties of the camera, including its field of view (focal length), image center, and skew. Both the parameters can be represented with different data structures, with the most common being a matrix.

If you develop a 3D mobile game, you are generally free to configure the cameras the way you want; you can put the camera above a 3D character running on a terrain (extrinsic) or set up a large field of view to have a large view of the character and the terrain (intrinsic). However, when you do Augmented Reality, the choice is constrained by the properties of the real camera in your mobile phone. In AR, we want properties of the virtual camera to match those of the real camera: the field of view and the camera position. This is an important element of AR, and we will explain how to realize it further in this chapter.

Camera parameters (intrinsic orientation)

The extrinsic parameters of the virtual camera will be explored in subsequent chapters; they are used for 3D registration in Augmented Reality. For our 3D overlay, we will now explore the intrinsic camera parameters.

There are different computational models for representing a virtual camera (and its parameters) and we will use the most popular one: the pinhole camera model. The pinhole camera model is a simplified model of a physical camera, where you consider that there is only a single point (pinhole) where light enters your camera image. With this assumption, computer vision researchers simplify the description of the intrinsic parameters as:

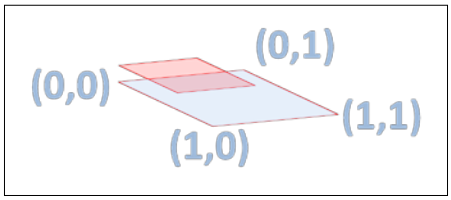
- **Focal length of your (physical or virtual) lens**: This together with the size of the camera center determines the **field of view** (**FOV**)—also called the angle of view—of your camera. The FOV is the extent of the object space your camera can see and is represented in radians (or degrees). It can be determined for the horizontal, vertical, and diagonal direction of your camera sensor.
- **Image center (principal point)**: This accommodates any displacement of the sensor from the center position.
- **Skew factor**: This is used for non-square pixels.

On non-mobile cameras you should also consider the lens distortion, such as the radial and the tangential distortions. They can be modeled and corrected with advanced software algorithms. Lens distortions on mobile phone cameras are usually corrected in hardware.

With all these concepts in mind, let's do a bit of practice now.

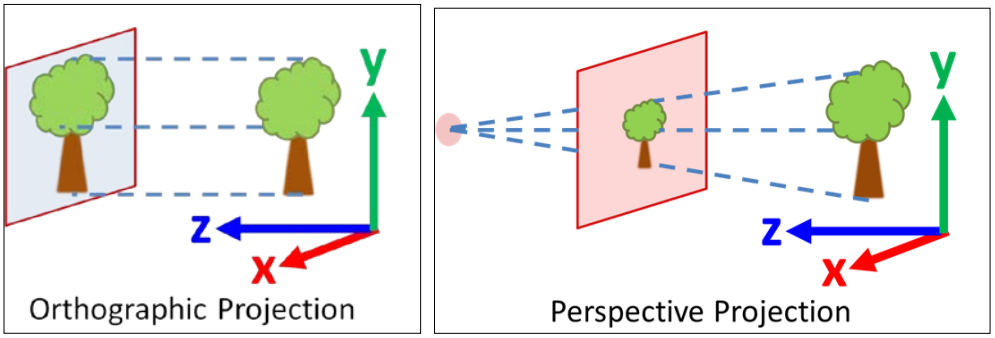
Using the scenegraph to overlay a 3D model onto the camera view

In the previous chapter you learned how to set up a single viewport and camera to render the video background. While the virtual camera determines how your 3D graphics are projected on a 2D image plane, the viewport defines the mapping of this image plane to a part of the actual window in which your application runs (or the whole screen of the smartphone if the app runs in fullscreen mode). It determines the portion of the application window in which graphics are rendered. Multiple viewports can be stacked and can cover the same or different screen areas as shown in the following figure. For a basic AR application, you typically have two viewports. One is associated with the camera rendering the background video and one is used with a camera rendering the 3D objects. Typically, these viewports cover the whole screen.



The viewport size is not defined in pixels but is unitless and is defined from 0 to 1 for the width and height to be able to easily adapt to changing window sizes. One camera is associated with one viewport at a time.

Remember that for the video background we used an orthographic camera to avoid perspective foreshortening of the video image. However, this perspective is crucial for getting a proper visual impression of your 3D objects. Orthographic (parallel) projection (on the left-hand side of the following figure) and perspective projection (on the right-hand side of the following figure) determine how the 3D volume is projected on a 2D image plane as shown in the following figure:



JME uses a right-handed coordinate system (OpenGL® convention, x on the right-hand side, y upwards, and z towards you). You certainly want 3D objects to appear bigger as the camera moves closer to them and smaller as it moves away. So how do we go along? Right, you just add a second camera—this time a perspective one—and an associated viewport that also covers the whole application window.

In the `SuperimposeJME` project associated with this chapter, we again have Android activity (`SuperimposeJMEActivity.java`) and a JME application class (`SuperimposeJME.java`). The application needs no major change from our previous project; you only have to extend the JME `SimpleApplication` class. In its `simpleInitApp()` startup method, we now explicitly differentiate between the initialization of the scene geometry (video background: `initVideoBackground();` 3D foreground scene: `initForegroundScene()`) and the associated cameras and viewports:

```
private float mForegroundCamFOVY = 30;
…
public void simpleInitApp() {
…
initVideoBackground(settings.getWidth(), settings.getHeight());
initForegroundScene();
initBackgroundCamera();
initForegroundCamera(mForegroundCamFOVY);
…
}
```

Note that the order in which the camera and viewports are initialized is important. Only when we first add the camera and viewport for the video background (`initBackgroundCamera()`) and later add the foreground camera and viewport (`initForegroundCamera()`), can we ensure that our 3D objects are rendered on top of the video background; otherwise, you would only see the video background.

We will now add your first 3D model into the scene using `initForegroundScene()`. A convenient feature of JME is that it supports the loading of external assets—for example, Wavefront files (`.obj`) or Ogre3D files (`.mesh.xml`/`.scene`)—including animations. We will load and animate a green ninja, a default asset that ships with JME.

```
private AnimControl mAniControl;
private AnimChannel mAniChannel;
…
public void initForegroundScene() {
Spatial ninja = assetManager.loadModel("Models/Ninja/Ninja.mesh.xml");
ninja.scale(0.025f, 0.025f, 0.025f);
ninja.rotate(0.0f, -3.0f, 0.0f);
ninja.setLocalTranslation(0.0f, -2.5f, 0.0f);
rootNode.attachChild(ninja);

DirectionalLight sun = new DirectionalLight();
sun.setDirection(new Vector3f(-0.1f, -0.7f, -1.0f));
rootNode.addLight(sun);

mAniControl = ninja.getControl(AnimControl.class);
mAniControl.addListener(this);
mAniChannel = mAniControl.createChannel();
mAniChannel.setAnim("Walk");
mAniChannel.setLoopMode(LoopMode.Loop);
mAniChannel.setSpeed(1f);
}
```

So in this method you load a model relative to your project's `root/asset` folder. If you want to load other models, you also place them in this `asset` folder. You scale, translate, and orient it and then add it to the root scenegraph node. To make the model visible, you also add a directional light shining from the top front onto the model (you can try not adding the light and see the result). For the animation, access the "Walk" animation sequence stored in the model. In order to do this, your class needs to implement the `AnimEventListener` interface and you need to use an `AnimControl` instance to access that animation sequence in the model. Finally, you will assign the "Walk" sequence to an `AnimChannel` instance, tell it to loop the animation, and set the animation speed.

Great, you have now loaded your first 3D model, but you still need to display it on the screen.

This is what you do next in `initForegroundCamera(fovY)`. It takes care of setting up the perspective camera and the associated viewport for your 3D model. As the perspective camera is characterized by the spatial extent of the object space it can see (the FOV), we pass the vertical angle of view stored in `mForegroundCamFOVY` to the method. It then attaches the root node of our scene containing the 3D model to the foreground viewport.

```
public void initForegroundCamera(float fovY) {
  Camera fgCam = new Camera(settings.getWidth(),
  settings.getHeight());
  fgCam.setLocation(new Vector3f(0f, 0f, 10f));
  fgCam.setAxes(new Vector3f(-1f,0f,0f),
  new Vector3f(0f,1f,0f), new Vector3f(0f,0f,-1f));
  fgCam.setFrustumPerspective(fovY,
  settings.getWidth()/settings.getHeight(), 1, 1000);
  ViewPort fgVP = renderManager.createMainView("ForegroundView",
  fgCam);
  fgVP.attachScene(rootNode);
  fgVP.setBackgroundColor(ColorRGBA.Blue);
  fgVP.setClearFlags(false, true, false);
}
```

While you could just copy some standard parameters from the default camera (similar to what we did with the video background camera), it is good to know which steps you actually have to do to initialize a new camera. After creating a perspective camera initialized with the window width and height, you set both the location (`setLocation()`) and the rotation (`setAxes()`) of the camera. JME uses a right-handed coordinate system, and our camera is configured to look along the negative z axis into the origin just as depicted in the previous figure. In addition, we set the vertical angle of the view passed to `setFrustumPerspective()` to 30 degrees, which corresponds approximately with a field of view that appears natural to a human (as opposed to a very wide or very narrow field of view).

Afterwards, we set up the viewport as we did for the video background camera. In addition, we tell the viewport to delete its depth buffer but retain the color and stencil buffers with `setClearFlags(false, true, false)`. We do this to ensure that our 3D models are always rendered in front of the quadrilateral holding the video texture, no matter if they are actually before or behind that quad in object space (beware that all our graphical objects are referenced in the same world coordinate system). We do not clear the color buffer as, otherwise, the color values of the video background, which are previously rendered into the color buffer will be deleted and we will only see the background color of this viewport (blue). If you run your application now, you should be able to see a walking ninja in front of your video background, as shown in the following pretty cool screenshot:

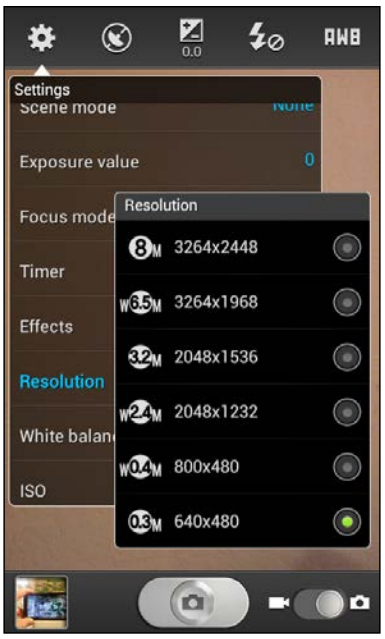
Improving the overlay

In the previous section you created a perspective camera, which renders your model with a vertical field of view of 30 degrees. However, to increase the realism of your scene, you actually want to match the field of view of your virtual and physical cameras as well as possible. This field of view in a general imaging system such as your phone's camera is dependent both on the size of the camera sensor and the focal length of the optics used. The focal length is a measure of how strongly the camera lens bends incoming parallel light rays until they come into focus (on the sensor plane), it is basically the distance between the sensor plane and the optical elements of your lens.

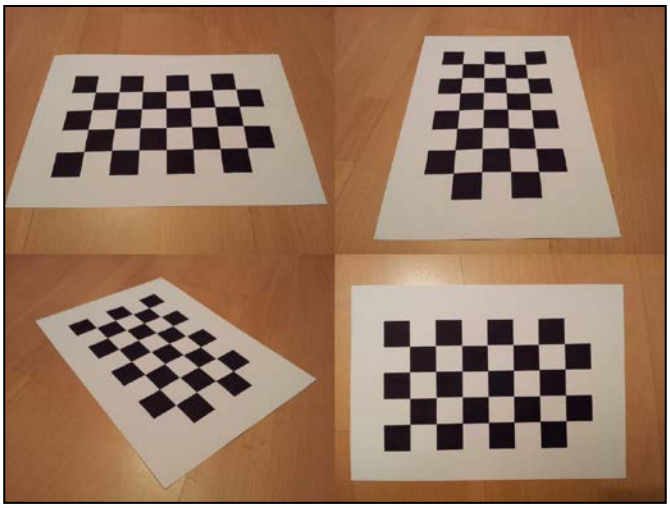
The FOV can be computed from the formula $a = 2 \arctan d/2f$, where d is the (vertical, horizontal, or diagonal) extent of the camera sensor and 2 is the focal length. Sounds easy, right? There is only a small challenge. You most often do not know the (physical) sensor size or the focal length of the phone camera. The good thing about the preceding formula is that you do not need to know the physical extent of your sensor or its focal length but can calculate it in arbitrary coordinates such as pixels. And for the sensor size, we can easily use the resolution of the camera image, which you already learned to query in *Chapter 2*, *Viewing the World*.

The trickiest part is to estimate the focal length of your camera. There are some tools that help you to do just this using a set of pictures taken from a known object; they are called camera resectioning tools (or geometric camera calibration tools). We will show you how to achieve this with a tool called GML C++ Camera Calibration Toolbox, which you can download from `http://graphics.cs.msu.ru/en/node/909`.

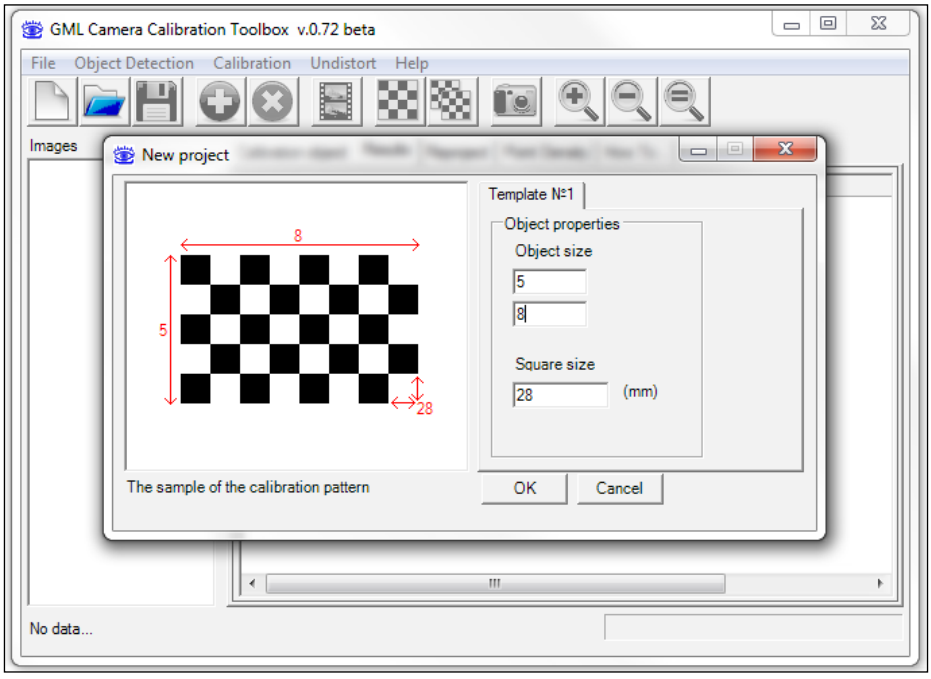
After installing the tool, open the standard camera app on your Android phone. Under the still image settings select the camera resolution that you also use in your JME application, for example, **640 x 480**, as shown in the following screenshot:



Take an A4 size printout of the `checkerboard_8x5_A4.pdf` file in the GML Calibration pattern subdirectory. Take at least four pictures with your camera app from different viewpoints (6 to 8 pictures will be better). Try to avoid very acute angles and try to maximize the checkerboards in the image. Example images are depicted in the following figure:



When you are done, transfer the images to a folder on your computer (for example, `AR4Android\calibration-images`). Afterwards, start the GML Camera Calibration app on your computer and create a new project. Type into the **New project** dialog box the correct number of black and white squares (for example, `5` and `8`), as shown in the following screenshot:

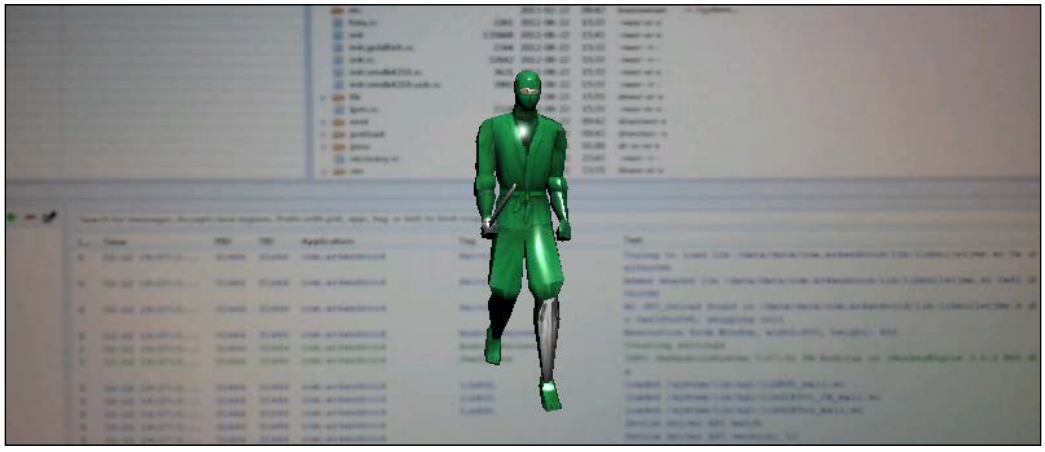


It is also crucial to actually measure the square size as your printer might scale the PDF to its paper size. Then, click on **OK** and start adding the pictures you have just taken (navigate to **Object detection** | **Add image**). When you have added all the images, navigate to **Object detection** | **Detect All** and then **Calibration** | **Calibrate**. If the calibration was successful, you should see camera parameters in the result tab. We are mostly interested in the **Focal length** section. While there are two different focal lengths for the x and y axes, it is fine to just use the first one. In the sample case of the images, which were taken with a Samsung Galaxy SII, the resulting focal length is 522 pixels.

You can then plug this number together with your vertical image resolution into the preceding formula and retrieve the vertical angle of the view in radians. As JME needs the angle in degrees, you simply convert it by applying this factor: *180/PI*. If you are also using a Samsung Galaxy SII, a vertical angle of view of approximately 50 degrees should result, which equals a focal length of approximately 28 mm in 35 mm film format (wide angle lens). If you plug this into the `mForegroundCamFOVY` variable and upload the application, the walking ninja should appear smaller as shown in the following figure. Of course, you can increase its size again by adjusting the camera position.

Note that you cannot model all parameters of the physical camera in JME. For example, you cannot easily set the principal point of your physical camera with your JME camera.

JME also doesn't support direct lens distortion correction. You can account for these artifacts via advanced lens correction techniques covered, for example, here: `http://paulbourke.net/miscellaneous/lenscorrection/`.



Summary

In this chapter, we introduced you to the concept of 3D rendering, the 3D virtual camera, and the notion of 3D overlay for Augmented Reality. We presented what a virtual camera is and its characteristics and described the importance of intrinsic camera parameters for accurate Augmented Reality. You also got a chance to develop your first 3D overlay and calibrate your mobile camera for improved realism. However, as you move your phone along, the video background changes, while the 3D models stay in place. In the next chapter, we will tackle one of the fundamental bricks of an Augmented Reality application: the registration.

4
Locating in the World

In the last chapter you learned how to overlay digital content on the view of the physical world. However, if you move around with your device, point it somewhere else, the virtual content will always stay at the same place on your screen. This is not exactly what happens in AR. The virtual content should stay at the same place relative to the physical world (and you can move around it), not remaining fixed on your screen.

In this chapter we will look at how to achieve **dynamic registration** between digital content and the physical space. If at every time step, we update the position of moving objects in our application, we will create the feeling that digital content sticks to the physical world. Following the position of moving elements in our scene can be defined as **tracking**, and this is what we will use and implement in this chapter. We will use sensor-based AR to update the registration between digital content and physical space. As some of these sensors are commonly of poor quality, we will show you how to improve the measurement you get from them using a technique named **sensor fusion**. To make it more practical, we will show you how to develop the basic building blocks for a simple prototype of one of the most common AR applications using global tracking: an AR Browser (such as Junaio, Layar, or Wikitude).

Knowing where you are – handling GPS

In this section, we will look at one of the major approaches for mobile AR and sensor-based AR (see *Chapter 1*, *Augmented Reality Concepts and Tools*), which uses **global tracking**. Global tracking refers to tracking in a global reference frame (world coordinate system), which can encompass the whole earth. We will first look at the position aspect, and then the location sensor built on your phone that will be used for AR. We will learn how to retrieve information from it using the Android API and will integrate its position information into JME.

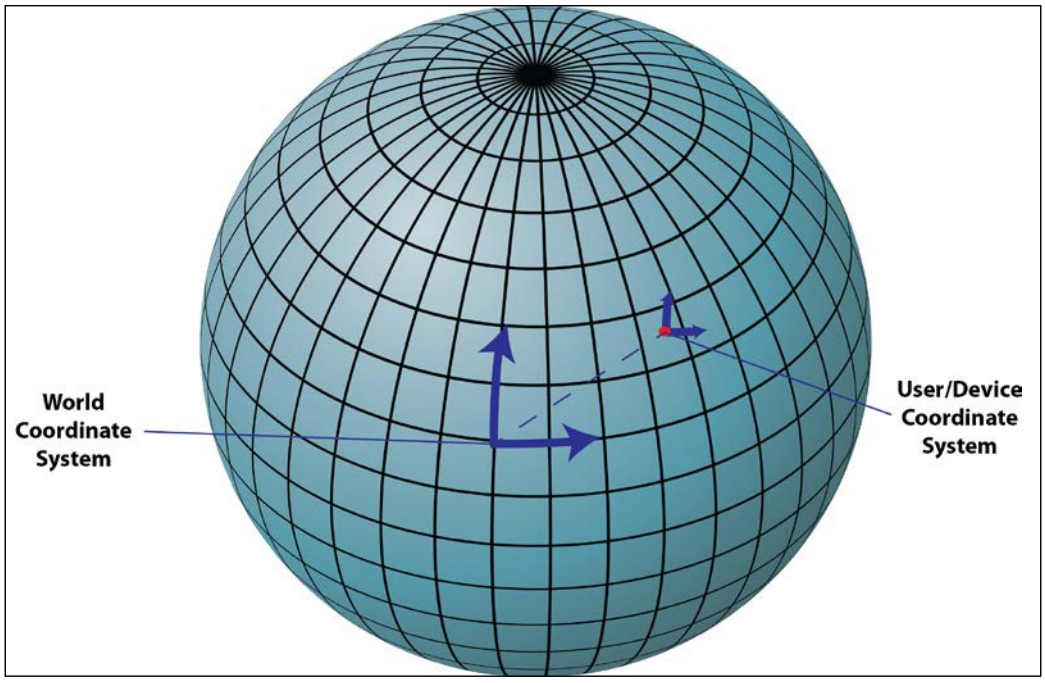
GPS and GNSS

So we need to track the position of the user to know where he/she is located in the real world. While we say we track the user, handheld AR applications actually track the position of the device.

User tracking versus device tracking

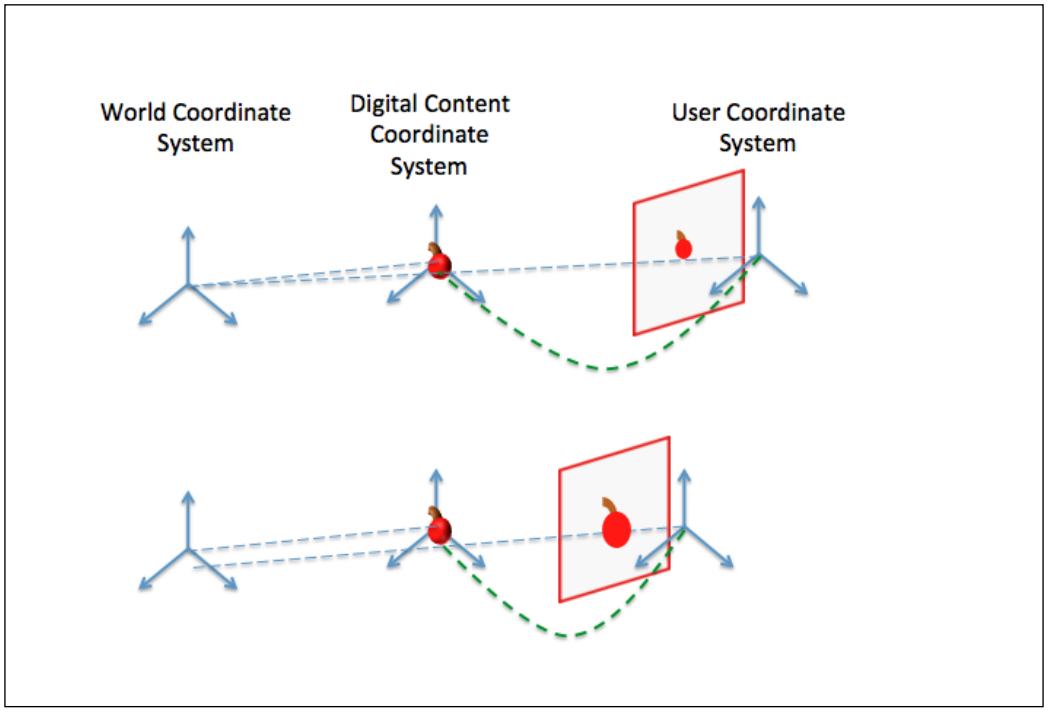
To create a fully-immersive AR application, you ideally need to know where the device is, where the body of the user in reference to the device is, and where the eyes of the user in reference of the body are. This approach has been explored in the past, especially with Head Mounted Displays. For that, you need to track the head of the user, the body of the user, and have all the static transformations between them (calibration). With mobile AR, we are still far from that; maybe in the future, users will wear glasses or clothes equipped with sensors which will allow creating more precise registration and tracking.

So how do we track the position of the device in a global coordinate system? Certainly you, or maybe some of your friends, have used a GPS for car navigation or for going running or hiking. GPS is one example of a common technology used for global tracking, in reference to an earth coordinate system, as shown in the following figure:



Most mobile phones are now equipped with GPS, so it seems an ideal technology for global tracking in AR. A GPS is the American version of a **global navigation satellite system** (**GNSS**). The technology relies on a constellation of geo-referenced satellites, which can give your position anywhere around the planet using geographic coordinates. GPS is not the only GNSS out there; a Russian version (**GLONASS**) is currently also operational, and a European version (**Galileo**) will be effective around 2020. However, GPS is currently the most supported GNSS on mobile devices, so we will use this term for the rest of the book when we talk about tracking with GNSS.

For common AR applications relying on GPS, you have two things to consider: the digital content location and the device location. If both of them are defined in the same coordinate system, in reference to earth, you will be able to know how they are in reference to each other (see the elliptical pattern in the following figure). With that knowledge, you can model the position of the 3D content in the user coordinate system and update it with each location update from your GPS sensor. As a result, if you move closer to an object (bottom to top), the object will appear closer (and bigger in the image), reproducing the behavior you have in the normal world.



A small issue we have with this technology is related to the coordinate system used in GPS. Using latitude and longitude coordinates (what a basic GPS delivers) is not the most adapted representation for using AR. When we do 3D graphics, we are used to a Euclidian coordinate system to position digital content; position using the Cartesian coordinate system, defined in terms of X, Y, and Z coordinates. So we need to address this problem by transforming these GPS coordinates to something more adapted.

JME and GPS – tracking the location of your device

The Google Android API offers access to GPS through the Location Manager service. The Location Manager can provide you GPS data, but it can also use the network (for example, Wi-Fi and cellphone network) to pinpoint your location and give you a rough estimation of it. In Android terminology, this is named Location Provider. To use the Location Manager, you need to apply the standard Android mechanism for notifications in Android based on a listener class; `LocationListener` in this case.

So open the `LocationAccessJME` project associated with this chapter, which is a modified version of the `SuperimposeJME` project (*Chapter 3*, *Superimposing the World*).

First, we need to modify our Android manifest to allow access to the GPS sensor. They are different quality modes regarding GPS (quality of estimated location), we will authorize all of them. So add these two permissions to your `AndroidManifest.xml` file:

```
  <uses-permission android:name="android.permission.ACCESS_COARSE_
LOCATION"/>
  <uses-permission android:name="android.permission.ACCESS_FINE_
LOCATION"/>
```

The project has, same as before, a JME class (`LocationAccessJME`), an activity class (`LocationAccessJMEActivity`), as well as `CameraPreview`. What we need to do is create a `LocationListener` class and a `LocationManager` class that we add to our `LocationAccessJMEActivity` class:

```
private LocationManager locationManager;
```

Inside the `LocationListener` class, we need to override different callback functions:

```
private LocationListener locListener= new LocationListener() {
  …
  @Override
```

```
    public void onLocationChanged(Location location) {
      Log.d(TAG, "onLocation: " + location.toString());
      if ((com.ar4android.LocationAccessJME) app != null) {
        ((com.ar4android.LocationAccessJME) app)
          .setUserLocation(xyzposition);
      }
    }
    …
  }
```

The `onLocationChanged` callback is the one which is the call for any changes in a user's location; the location parameter contains both the measured latitude and longitude (in degrees). To pass the converted data to our JME, we will use the same principle as before: call a method in our JME class using the location as argument. So `setUserLocation` will be called each time there is an update of the location of the user, and the new value will be available to the JME class.

Next, we need to access the location manager service and register our location listener to it, using the `requestLocationUpdates` function:

```
public void onResume() {
  super.onResume();
  …
  locationManager =
  (LocationManager)getSystemService(LOCATION_SERVICE);
  locationManager.requestLocationUpdates
  (LocationManager.GPS_PROVIDER, 500, 0, locListener);
}
```

The parameters of `requestLocationUpdates` are the types of provider we want to use (GPS versus network), update frequency (in milliseconds), and change of position threshold (in meters) as our listener.

On the JME side, we need to define two new variables to our `LocationAccessJME` class:

```
//the User position which serves as intermediate storage place
  for the Android
//Location listener position update
private Vector3f mUserPosition;

//A flag indicating if a new Location is available
private boolean mNewUserPositionAvailable =false;
```

We also need to define our `setUserLocation` function, which is called from the callback in `LocationListener`:

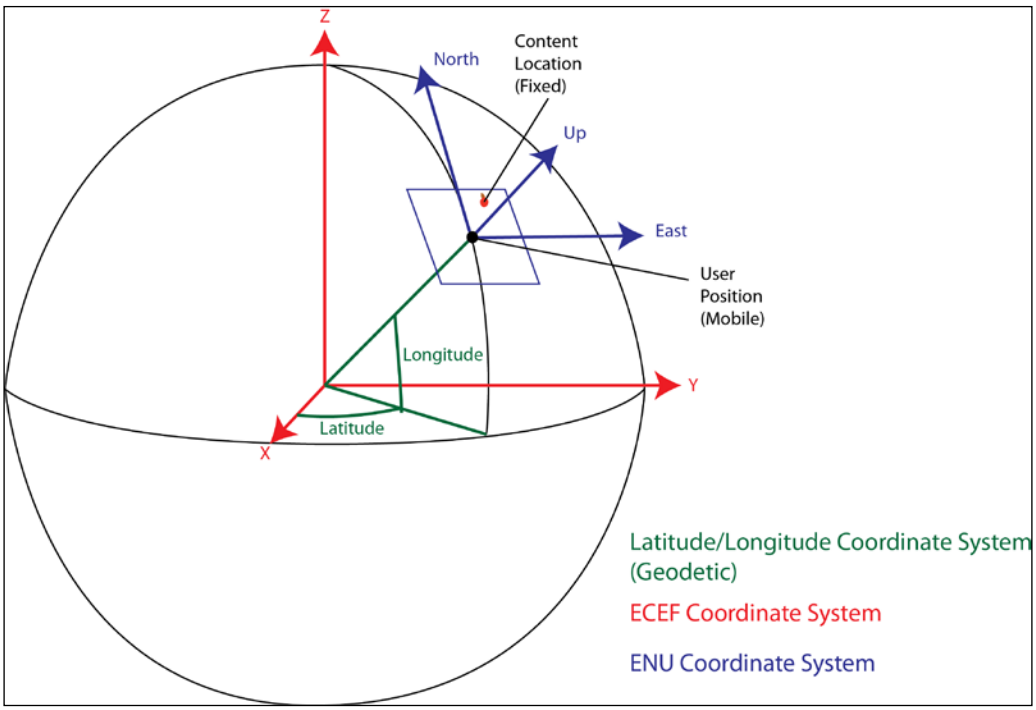
```
public void setUserLocation(Vector3f location) {
if (!mSceneInitialized) {
  return;
}
WSG84toECEF(location,mUserPosition);
//update your POI location in reference to the user position
….
mNewUserPositionAvailable =true;
}
```

Inside this function we need to transform the position of the camera from latitude/longitude format to a Cartesian coordinate system. There are different techniques to do so; we will use the conversion algorithm from the SatSleuth website (`http://www.satsleuth.com/GPS_ECEF_Datum_transformation.htm`), converting our data to an **ECEF** (**Earth-Centered, Earth-Fixed**) format. Now we have `mUserPosition` available in ECEF format in our JME class. Each time a user's location will change, the `onLocationChange` method and `setUserLocation` will be called and we will get an updated value of `mUserPosition`. The question now is how we use this variable in our scenegraph and in relation with geo-referenced digital content (for example, POI)?

The method to use is to reference your content locally from your current position. For doing that, we need to use an additional coordinate system: the **ENU** (**East-North-Up**) coordinate system. For each data you have (for example, a certain number of POIs at 5 km radius from your position), you compute the location from your current position. Let's have a look at how we can do that on our ninja model, as shown in the following code:

```
Vector3f ECEFNinja=new Vector3f();
Vector3f ENUNinja=new Vector3f();
WSG84toECEF(locationNinja,ECEFNinja);
ECEFtoENU(location,mUserPosition,ECEFNinja,ENUNinja);
mNinjaPosition.set(ENUNinja.x,0,ENUNinja.y);
```

The position of the ninja in latitude-longitude format (`locationNinja`) is also converted to the ECEF format (`ECEFNinja`). From there, using the current GPS location (in latitude-longitude format and ECEF format, location, mUserPosition), we compute the position of the ninja in a local coordinate system (`ENUNinja`). Each time the user moves, his or her GPS position will be updated, transformed to ECEF format, and the local position of the content will be updated, which will trigger a different rendering. That's it! We have implemented GPS-based tracking. An illustration of the relation of the different coordinate systems is represented in the following figure:



The only remaining part is to update the position of the model using the new local position. We can implement that from the `simpleUpdate` function by adding the following code:

```
if (mNewUserPositionAvailable) {
  Log.d(TAG,"update user location");
  ninja.setLocalTranslation
  (mNinjaPosition.x+0.0f,mNinjaPosition.
  y-2.5f,mNinjaPosition.z+0.0f);
  mNewUserPositionAvailable=false;
}
```

In a real AR application, you may have some 3D content positioned around your current position in a GPS coordinate system, such as having a virtual ninja positioned in Fifth street in New York, or in front of the Eiffel Tower in Paris.

Since we want to be sure, you can run this sample independently of where you are currently testing and reading the book (from New York to Timbuktu). We will modify this demo slightly for educational purposes. What we will do is add the ninja model at 10 meters from your initial GPS location (that is, first time the GPS updates), by adding the following call in `setUserLocation`:

```
if (firstTimeLocation) {
  //put it at 10 meters
  locationNinja.setLatitude(location.getLatitude()+0.0001);
  locationNinja.setLongitude(location.getLongitude());
  firstTimeLocation=false;
}
```

Time for testing: deploy the application on your mobile and go outside to a location where you should get a nice GPS reception (you should be able to see the sky and avoid a really cloudy day). Don't forget to activate the GPS on your device. Start the application, move around and you should see the ninja shifting positions. Congratulations, you developed your first instance of tracking for an AR application!

Knowing where you look – handling inertial sensors

With the previous example and access to GPS location, we can now update a user's location and be able to do a basic tracking in Augmented Reality. However, this tracking is only considering position of the user and not his or her orientation. If, for example, the user rotates the phone, nothing will happen, with changes being effective only if he is moving. For that we need to be able to detect changes in rotation for the user; this is where inertial sensors come in. The inertial sensors can be used to detect changes in orientation.

Understanding sensors

In the current generation of mobile phones, three types of sensors are available and useful for orientation:

- **Accelerometers**: These sensors detect the proper acceleration of your phone, also called **g-force** acceleration. Your phone is generally equipped with multi-axis model to deliver you acceleration in the 3 axes: pitch, roll, and tilt of your phone. They were the first sensors made available on mobile phones and are used for sensor-based games, being cheap to produce. With accelerometers, and a bit of elementary physics, you are able to compute the orientation of the phone. They are, however, rather inaccurate and the measured data is really noisy (which can result in getting jitters in your AR application).
- **Magnetometers**: They can detect the earth's magnetic field and act like a compass. Ideally, you can get the north direction with them by measuring the magnetic field in three dimensions and know where your phone points. The challenge with magnetometers is that they can easily be distracted by metallic objects around them, such as a watch on the user's wrist, and then indicate a wrong north direction.
- **Gyroscopes**: They measure angular velocity using the **Coriolis Effect**. The ones used in your phone are **multi-axis miniature mechanical system** (**MEMS**) using a vibrating mechanism. They are more accurate than the previous sensors, but their main issue is the drift: the accuracy of measurement decreases over time; after a short period your measure starts getting really inaccurate.

You can combine measurements of each of them to address their limitations, as we will see later in this chapter. Inertial sensors have been used intensively before coming to mobile phones, the most famous usage being in planes for measuring their orientation or velocity, used as an **inertial measurement unit** (**IMU**).
As manufacturers always try to cut down costs, quality of the sensors varies considerably between mobile devices. The effect of noise, drift, and inaccuracy will induce your AR content to jump or move without you displacing the phone or it may lead to positioning the content in the wrong orientation. Be sure you test a range of them if you want to deploy your application commercially.

Sensors in JME

Sensor access on Google Android API is made through `SensorManager`, and uses `SensorListener` to retrieve measurements. `SensorManager` doesn't give you access only to the inertial sensors, but to all the sensors present on your phone. Sensors are divided in three categories in the Android API: motion sensors, environmental sensors, and position sensors. The accelerometers and the gyroscope are defined as motion sensors; the magnetometer is defined as a position sensor. The Android API also implements some software sensors, which combine the values of these different sensors (which may include position sensor too) to provide you with motion and orientation information. The five motion sensors available are:

- `TYPE_ACCELEROMETER`
- `TYPE_GRAVITY`
- `TYPE_GYROSCOPE`
- `TYPE_LINEAR_ACCELERATION`
- `TYPE_ROTATION_VECTOR`

Please refer to the Google Developer Android website `http://developer.android.com/guide/topics/sensors/sensors_overview.html`, for more information about the characteristics of each of them. So let's open the `SensorAccessJME` project. As we did before, we define a `SensorManager` class and we will add a `Sensor` class for each of these motion sensors:

```
private SensorManager sensorManager;
Sensor rotationVectorSensor;
Sensor gyroscopeSensor;
Sensor magneticFieldSensor;
Sensor accelSensor;
Sensor linearAccelSensor;
```

We also need to define `SensorListener`, which will handle any sensor changes from the motion sensors:

```
private SensorEventListener sensorListener = new SensorEventListener()
{
    …
@Override
public void onSensorChanged(SensorEvent event) {
  switch(event.sensor.getType()) {
      …
      case Sensor.TYPE_ROTATION_VECTOR:
  float[] rotationVector =
  {event.values[0],event.values[1], event.values[2]};
```

```
    float[] quaternion = {0.f,0.f,0.f,0.f};
    sensorManager.getQuaternionFromVector
    (quaternion,rotationVector);
    float qw = quaternion[0]; float qx = quaternion[1];
    float qy = quaternion[2];float qz = quaternion[3];
      double headingQ = Math.atan2(2*qy*qw-2*qx*qz ,
      1 - 2*qy*qy - 2*qz*qz);
    double pitchQ = Math.asin(2*qx*qy + 2*qz*qw);
    double rollQ = Math.atan2(2*qx*qw-2*qy*qz ,
      1 - 2*qx*qx - 2*qz*qz);
    if ((com.ar4android.SensorAccessJME) app != null) {
    ((com.ar4android.SensorAccessJME) app).
    setRotation((float)pitchQ, (float)rollQ, (float)headingQ);
    }
    }
  }
  };
```

The rotation changes could also solely be handled with Quaternions, but we explicitly used Euler angles for a more intuitive understanding. Privilege quaternions as composing rotations is easier and they don't suffer from "gimbal lock".

Our listener overrides two callbacks: the `onAccuracyChanged` and `onSensorChanged` callbacks. The `onSensorChanged` channel will be called for any changes in the sensors we registered to `SensorManager`. Here we identify which type of sensor changed by querying the type of event with `event.sensor.getType()`. For each type of sensor, you can use the generated measurement to compute the new orientation of the device. In this example we will only show you how to use the value of the `TYPE_ROTATION_VECTOR` sensor (software sensor). The orientation delivered by this sensor needs to be mapped to match the coordinate frame of the virtual camera. We pass Euler angles (heading, pitch, and roll) to the JME application to achieve this in the JME application's `setRotation` function (the Euler angle is just another representation of orientation and can be calculated from Quaternions and axis-angle representations delivered in the sensor event).

Now, having our `SensorListener`, we need to query `SensorManager` to get the sensor service and initialize our sensors. In your `onCreate` method add:

```
    // sensor setup
    sensorManager = (SensorManager)getSystemService(SENSOR_SERVICE);
    List<Sensor> deviceSensors = sensorManager.getSensorList
                                 (Sensor.TYPE_ALL);
    Log.d(TAG, "Integrated sensors:");
```

```
    for(int i = 0; i < deviceSensors.size(); ++i ) {
      Sensor curSensor = deviceSensors.get(i);
      Log.d(TAG, curSensor.getName() + "\t" + curSensor.getType()
      + "\t" + curSensor.getMinDelay() / 1000.0f);
    }
  initSensors();
```

After getting access to the sensor service, we query the list of all available sensors and display the results on our logcat. For initializing the sensors, we call our `initSensors` method, and define it as:

```
    protected void initSensors(){
      //look specifically for the gyroscope first and then for the
     rotation_vector_sensor (underlying sensors vary from platform
     to platform)
    gyroscopeSensor = initSingleSensor(Sensor.TYPE_GYROSCOPE,
    "TYPE_GYROSCOPE");
    rotationVectorSensor =
    initSingleSensor(Sensor.TYPE_ROTATION_VECTOR,
    "TYPE_ROTATION_VECTOR");
    accelSensor = initSingleSensor(Sensor.TYPE_ACCELEROMETER,
    "TYPE_ACCELEROMETER");
      linearAccelSensor =
      initSingleSensor(Sensor.TYPE_LINEAR_ACCELERATION,
      "TYPE_LINEAR_ACCELERATION");
    magneticFieldSensor =
    initSingleSensor(Sensor.TYPE_MAGNETIC_FIELD,
    "TYPE_MAGNETIC_FIELD");
    }
```

The function `initSingleSensor` will create an instance of `Sensor` and register our previously created listener with a specific type of sensor passed in argument:

```
    protected Sensor initSingleSensor( int type, String name ){
    Sensor newSensor = sensorManager.getDefaultSensor(type);
    if(newSensor != null){
      if(sensorManager.registerListener(sensorListener, newSensor,
      SensorManager.SENSOR_DELAY_GAME)) {
        Log.i(TAG, name + " successfully registered default");
      } else {
        Log.e(TAG, name + " not registered default");
      }
    } …
    return newSensor;
    }
```

We shouldn't forget to unregister the listener when we quit the application, so modify your `onStop` method as follows:

```
public void onStop() {
  super.onStop();
  sensorManager.unregisterListener(sensorListener);
}
```

So, we are now set in our `Activity`. In our `SensorAccessJME` class we add following variables:

```
private Quaternion mRotXYZQ;
private Quaternion mInitialCamRotation;
private Quaternion mCurrentCamRotation;
```

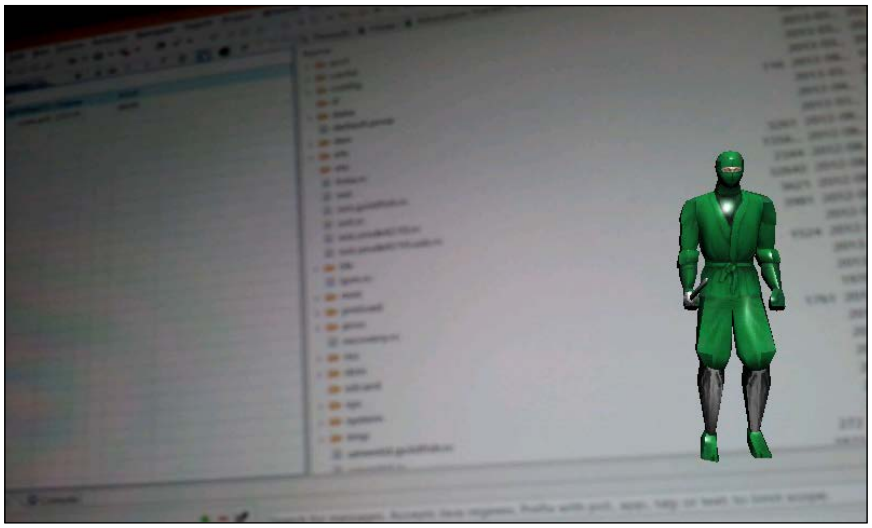
The variable `mInitialCamRotation` holds the initial camera orientation, `mRotXYZQ` holds the sensor orientation mapped to the camera coordinate system, and `mCurrentCamRotation` stores the final camera rotation which is composed from multiplying `mInitialCamRotation` with `mRotXYZQ`. The `setRotation` function takes the sensor values from the Android activity and maps them to the camera coordinate system. Finally, it multiplies the current rotation values with the initial camera orientation:

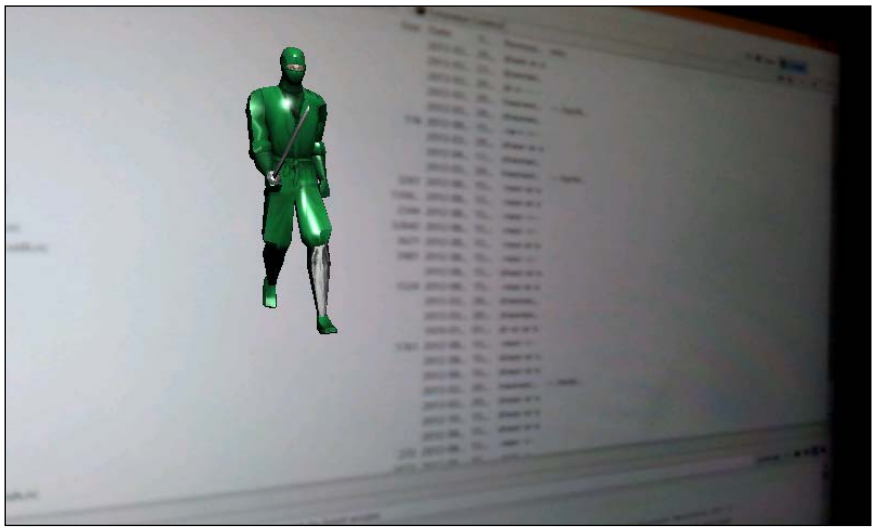
```
public void setRotation(float pitch, float roll, float heading)
{
  if (!mSceneInitialized) {
    return;
  }
  mRotXYZQ.fromAngles(pitch , roll - FastMath.HALF_PI, 0);
  mCurrentCamRotation = mInitialCamRotation.mult(mRotXYZQ);
  mNewCamRotationAvailable = true;
```

As a last step, we need to use this rotation value for our virtual camera, the same as we did for our GPS example. In `simpleUpdate` you now add:

```
if (mNewCamRotationAvailable) {
  fgCam.setAxes(mCurrentCamRotation);
  mNewCamRotationAvailable = false;
}
```

So, we are now ready to run the application. It's important to consider that the natural orientation of the device, which defines the coordinate system for motion sensors, is not the same for all devices. If your device is, by default, in the portrait mode and you change it to landscape mode , the coordinate system will be rotated. In our examples we explicitly set the device orientation to landscape. Deploy your application on your device using this default orientation mode. You may need to rotate your device around to see the ninja moving on your screen, as shown in the following screenshots:



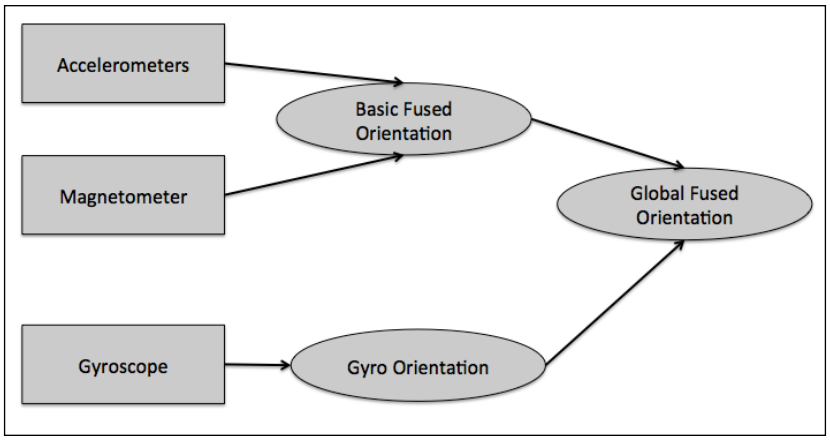


Improving orientation tracking – handling sensor fusion

One of the limitations with sensor-based tracking is the sensors. As we introduced before, some of the sensors are inaccurate, noisy, or have drift. A technique to compensate their individual issue is to combine their values to improve the overall rotation you can get with them. This technique is called sensor fusion. There are different methods for fusing the sensors, we will use the method presented by *Paul Lawitzki* with a source code under MIT License available at `http://www.thousand-thoughts.com/2012/03/android-sensor-fusion-tutorial/`. In this section, we will briefly explain how the technique works and how to integrate sensor fusion to our JME AR application.

Sensor fusion in a nutshell

The fusion algorithm proposed by *Paul Lawitzki* merges accelerometers, magnetometers, and gyroscope sensor data. Similar to what is done with software sensor of an Android API, accelerometers and magnetometers are first merged to get an absolute orientation (magnetometer, acting as a compass, gives you the true north). To compensate for the noise and inaccuracy of both of them, the gyroscope is used. The gyroscope, being precise but drifting over time, is used at high frequency in the system; the accelerometers and magnetometers are considered over longer periods. Here is an overview of the algorithm:



You can find more information about the details of the algorithm (complimentary filter) on *Paul Lawitzki's* webpage.

Sensor fusion in JME

Open the `SensorFusionJME` project. The sensor fusion uses a certain number of internal variables that you declare at the beginning of `SensorFusionJMEActivity`:

```
// angular speeds from gyro
private float[] gyro = new float[3]; …
```

Also add the code of different subroutines used by the algorithm:

- `calculateAccMagOrientation`: Calculates the orientation angles from the accelerometer and magnetometer measurement
- `getRotationVectorFromGyro`: Calculates a rotation vector from the gyroscope angular speed measurement
- `gyroFunction`: Writes the gyroscope-based orientation into `gyroOrientation`
- **Two matrix transformation functions**: `getRotationMatrixFromOrientation` and `matrixMultiplication`

The main part of the processing is done in the `calculatedFusedOrientationTask` function. This function generates new fused orientation as part of `TimerTask`, a task that can be scheduled at a specific time. At the end of this function, we will pass the generated data to our JME class:

```
if ((com.ar4android.SensorFusionJME) app != null) {
      ((com.ar4android.SensorFusionJME)
      app).setRotationFused((float)(fusedOrientation[2]),
      (float)(-fusedOrientation[0]),
      (float)(fusedOrientation[1]));
    }
}
```

The argument passed to our JME activity bridge function (`setRotationFused`) is the fused orientation defined in the Euler angles format.

We also need to modify our `onSensorChanged` callback to call the subroutines used by `calculatedFusedOrientationTask`:

```
public void onSensorChanged(SensorEvent event) {
  switch(event.sensor.getType()) {
  case Sensor.TYPE_ACCELEROMETER:
    System.arraycopy(event.values, 0, accel, 0, 3);
    calculateAccMagOrientation();
```

```
      break;
    case Sensor.TYPE_MAGNETIC_FIELD:
      System.arraycopy(event.values, 0, magnet, 0, 3);
      break;
    case Sensor.TYPE_GYROSCOPE:
      gyroFunction(event)
      break;
  }
```

For our activity class, the last change is to specify a task for our timer, specify the schedule rate, and the delay before the first execution. We add that to our `onCreate` method after the call to `initSensors`:

```
fuseTimer.scheduleAtFixedRate(new calculateFusedOrientationTask(),
1000, TIME_CONSTANT);
```

On the JME side, we define a new bridge function for updating the rotation (and again converting the sensor orientation into an appropriate orientation of the virtual camera):

```
public void setRotationFused(float pitch, float roll, float heading) {
  if (!mSceneInitialized) {
    return;
  } // pitch: cams x axis roll: cams y axisheading: cams z axis
  mRotXYZQ.fromAngles(pitch + FastMath.HALF_PI , roll -
  FastMath.HALF_PI, 0);
  mCurrentCamRotationFused = mInitialCamRotation.mult(mRotXYZQ);
  mNewUserRotationFusedAvailable = true;
}
```

We finally use this function in the same way as for `setRotation` in `simpleUpdate`, updating camera orientation with `fgCam.setAxes(mCurrentCamRotationFused)`. You can now deploy the application and see the results on your device.

If you combine the `LocationAccessJME` and `SensorAccessJME` examples, you will now get full 6 degrees of freedom (6DOF) tracking, which is the foundation for a classical sensor-based AR application.

Getting content for your AR browser – the Google Places API

After knowing how to obtain your GPS position and the orientation of the phone, you are now ready to integrate great content into the live view of the camera. Would it not be cool to physically explore points of interests, such as landmarks and shops around you? We will now show you how to integrate popular location-based services such as the Google Places API to achieve exactly this. For a successful integration into your application, you will need to perform the following steps:

- Query for point of interests (POIs) around your current location
- Parse the results and extract information belonging to the POIs
- Visualize the information in your AR view

Before we start, you have to make sure that you have a valid API key for your application. For that you also need a Google account. You can obtain it by logging in with your Google account under `https://code.google.com/apis/console`.

For testing your application you can either use the default project `API Project` or create a new one. To create a new API key you need to:

1. Click on the link **Services** in the menu on the left-hand side.
2. Activate the Places API status switch.
3. Access your key by clicking on the **API access** menu entry on the left-hand side menu and looking at the **Simple API Access** area.

You can store the key in the `String mPlacesKey = "<YOUR API KEY HERE>"` variable in the `LocationAccessJME` project.

Next, we will show you how to query for POIs around the devices location, and getting some basic information such as their name and position. The integration of this information into the AR view follows the same principles as described in the *JME and GPS – tracking the location of your device* section.

Querying for POIs around your current location

Previously in this chapter, you learned how to obtain your current location in the world (latitude and longitude). You can now use this information to obtain the location of POIs around you. The Google Places API allows you to query for landmarks and businesses in the vicinity of the user via HTTP requests and returns the results as JSON or XML strings. All queries will be addressed towards URLs starting with `https://maps.googleapis.com/maps/api/place/`.

While you could easily make the queries in your web browser, you would want to have both the request sent and the response processed inside your Android application. As calling a URL and waiting for the response can take up several seconds, you would want to implement this request-response processing in a way that does not block the execution of your main program. Here we show you how to do that with threads.

In your `LocationAccessJME` project, you define some new member variables, which take care of the interaction with the Google Places API. Specifically, you create a `HttpClient` for sending your request and a list `List<POI> mPOIs`, for storing the most important information about POIs. The `POI` class is a simple helper class to store the Google Places reference string (a unique identifier in the Google Places database, the POI name, its latitude, and longitude):

```
private class POI {
  public String placesReference;
  public String name;
  public Location location;
…
}
```

Of course, you can easily extend this class to hold additional information such as street address or image URLs. To query for POIs you make a call to the `sendPlacesQuery` function. We do the call at program startup, but you can easily do it in regular intervals (for example, when the user moves a certain distance) or explicitly on a button click.

```
public void sendPlacesQuery(final Location location,  final Handler
guiHandler) throws Exception  {
Thread t = new Thread() {
public void run() {
  Looper.prepare();
  BufferedReader in = null;
  try {
    String url =
    "https://maps.googleapis.com/maps/api/place/nearbysearch/json?
    location=" + location.getLatitude() + "," +
    location.getLongitude() + "&radius=" +  mPlacesRadius +
    "&sensor=true&key=" + mPlacesKey;
    HttpConnectionParams.setConnectionTimeout
    (mHttpClient.getParams(), 10000);
    HttpResponse response;
    HttpGet get = new HttpGet(url);
    response = mHttpClient.execute(get);
```

```
        Message toGUI = guiHandler.obtainMessage();
…
guiHandler.sendMessage(toGUI);
…
```

In this method, we create a new thread for each query to the Google Places service. This is very important for not blocking the execution of the main program. The response of the Places API should be a JSON string, which we pass to a `Handler` instance in the main thread to parse the JSON results, which we will discuss next.

Parsing the Google Places APIs results

Google Places returns its result in the lightweight JSON format (with XML being another option). You can use the `org.json` library delivered as a standard Android package to conveniently parse those results.

A typical JSON result for your query will look like:

```
{
   …
   "results" : [
      {
         "geometry" : {
            "location" : {
               "lat" : 47.07010720,
               "lng" : 15.45455070
            },
   …
         },
         "name" : "Sankt Leonhard",
         "reference" :
         "CpQBiQAAADXt6JM47sunYZ8vZvt0GViZDLICZi2JLRdfhHGbtK-
         ekFMjkaceN6GmECaynOnR69buuDZ6t-PKow-
         J98l2tFyg3T50P0Fr39DRV3YQMpqW6YGhu5sAzArNzipS2
         tUY0ocoMNHoNSGPbuuYIDX5QURVgncFQ5K8eQL8OkPST78
         A_lKTN7icaKQV7HvvHkEQJBIQrx2r8IxIYuaVhL1mOZOsK
         BoUQjlsuuhqa1k7OCtxThYqVgfGUGw",
         …
      },
…
}
```

In `handleMessage` of our handler `placesPOIQueryHandler`, we will parse this JSON string into a list of POIs, which then can be visualized in your AR view:

```
public void handleMessage(Message msg) {
  try {
    JSONObject response = new JSONObject(msg.obj.toString());
    JSONArray results = response.getJSONArray("results");
    for(int i = 0; i < results.length(); ++i) {
      JSONObject curResult = results.getJSONObject(i);
      String poiName = curResult.getString("name");
      String poiReference = curResult.getString("reference");
      double lat =
      curResult.getJSONObject("geometry").
      getJSONObject("location").getDouble("lat");
      double lng =
      curResult.getJSONObject("geometry").
      getJSONObject("location").getDouble("lng");
      Location refLoc = new
      Location(LocationManager.GPS_PROVIDER);
      refLoc.setLatitude(lat);
      refLoc.setLongitude(lng);
      mPOIs.add(new POI(poiReference, poiName, refLoc));
      …
    }
  …
  }
}
```

So that is it. You now have your basic POI information and with the latitude, longitude information you can easily instantiate new 3D objects in JME and position them correctly relative to your camera position, just as you did with the ninja. You can also query for more details about the POIs or filter them by various criteria. For more information on the Google Places API please visit `https://developers.google.com/places/documentation/`.

If you want to include text in the 3D scene, we recommend avoiding the use of 3D text objects as they result in a high number of additional polygons to render. Use bitmap text instead, which you render as a texture on a mesh that can be generated.

Summary

In this chapter we introduced you to the first popular methods of mobile AR: GPS and sensor-based Augmented Reality. We introduced the basic building blocks of tracking the device location in a global reference frame, dynamically determining the device orientation, improving the robustness of orientation tracking, and finally using the popular Google Places API to retrieve information about POIs around the user which can then be integrated into the AR view.

In the next chapter we will introduce you to the second popular way of realizing mobile AR: computer vision-based Augmented Reality.

5

Same as Hollywood – Virtual on Physical Objects

In the previous chapter you learned about the basic building blocks for implementing GPS and sensor-based AR applications. If you tried the different examples we presented, you might have noticed that the feeling of getting digital objects in real space (*registration*) works but can become coarse and unstable. This is mainly due to the accuracy problems of the used sensors (GPS, accelerometer, and so on) found in smartphones or tablet devices, and the characteristics of these technologies (for example, gyroscope drifting, GPS reliance on satellite visibility, and other such technologies). In this chapter, we will introduce you to a more robust solution, with it being the second major approach for supporting mobile AR: **Computer vision-based AR**.

Computer vision-based AR doesn't rely on any external sensors but uses the content of the camera image to support tracking, which is analysis through a flow of different algorithms. With computer vision-based AR, you get a better registration between the digital and physical worlds albeit at a little higher cost in terms of processing.

Probably, without even knowing it, you have already seen computer vision-based registration. If you go to see a blockbuster action movie with lots of cinematic effects, you will sometimes notice that some digital content has been overlaid over the physical recording set (for example, fake explosions, fake background, and fake characters running). In the same way as AR, the movie industry has to deal with the registration between digital and physical content, relying on analyzing the recorded image to recover tracking and camera information (using, for example, the match matchmoving technique). However, compared to Augmented Reality, it's done offline, and not in real time, generally relying on heavy workstations for registration and visual integration.

In this chapter, we will introduce you to the different types of computer vision-based tracking for AR. We will also describe to you the integration of a well-used and high-quality tracking library for mobile AR, **Vuforia™** by Qualcomm® Inc. With this library, we will be able to implement our first computer vision-based AR application.

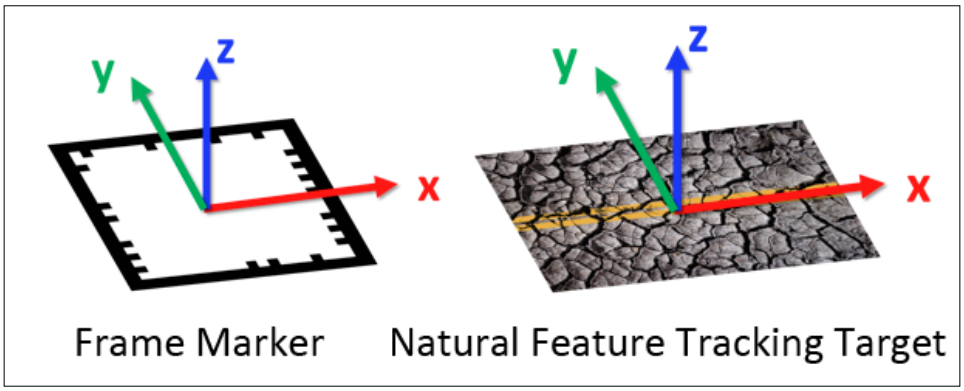
Introduction to computer vision-based tracking and Vuforia™

So far, you have used the camera of the mobile phone exclusively for rendering the view of the real world as the background for your models. Computer vision-based AR goes a step further and processes each image frame to look for familiar *patterns* (or image features) in the camera image.

In a typical computer vision-based AR application, planar objects such as *frame markers* or *natural feature tracking targets* are used to position the camera in a *local coordinate system* (see *Chapter 3*, *Superimposing the World*, *Figure showing the three most common coordinate systems*). This is in contrast to the global coordinate system (the earth) used in sensor-based AR but allows for more precise and stable overlay of virtual content in this local coordinate frame. Similar to before, obtaining the tracking information allows the updating of information about the virtual camera in our 3D graphics rendering engine and automatically provides us with registration.

Choosing physical objects

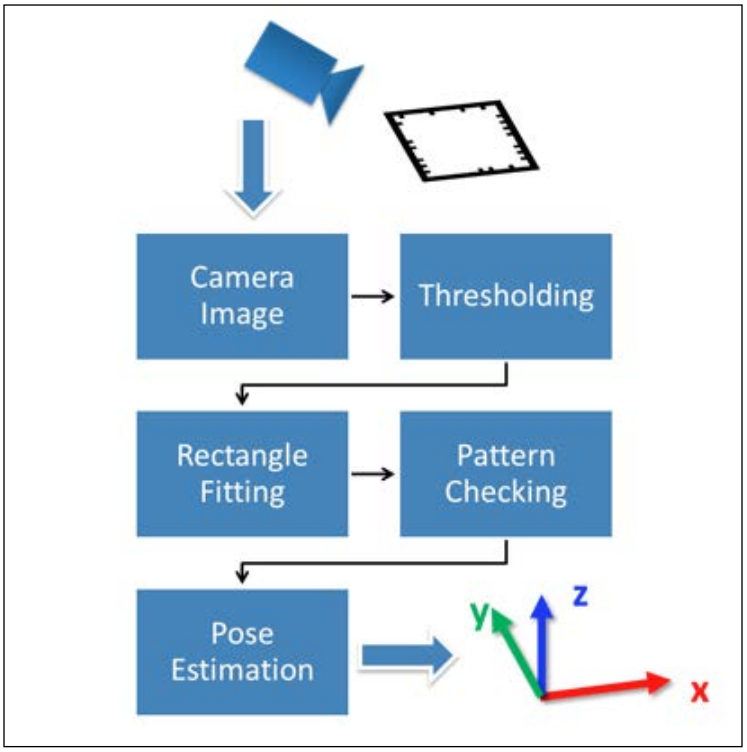
In order to successfully implement computer vision-based AR, you need to understand which physical objects you can use to track the camera. Currently there are two major approaches to do this: Frame markers (**Fiducials**) and natural feature tracking targets (planar textured objects), as shown in the following figure. We will discuss both of them in the following section.



Understanding frame markers

In the early days of mobile Augmented Reality, it was of paramount importance to use computationally efficient algorithms. Computer vision algorithms are traditionally demanding as they generally rely on image analysis, complex geometric algorithms, and mathematical transformation, summing to a large number of operations that should take place at every time frame (to keep a constant frame rate at 30 Hz, you only have 33 ms for all that). Therefore, one of the first approaches to computer vision-based AR was to use relatively simple types of objects, which could be detected with computationally low-demanding algorithms, such as Fiducial markers. These markers are generally only defined at a grayscale level, simplifying their analysis and recognition in a traditional physical world (think about QR code but in 3D).

A typical algorithmic workflow for detecting these kinds of markers is depicted in the following figure and will be briefly explained next:



After an acquired camera image being converted to a grayscale image, the **threshold** is applied, that is, the grayscale level gets converted to a purely black and white image. The next step, **rectangle detection**, searches for edges in this simplified image, which is then followed by a process of detecting closed-contour, and potentially parallelogram shapes. Further steps are taken to ensure that the detected contour is really a parallelogram (that is, it has exactly four points and a couple of parallel lines). Once the shape is confirmed, the content of the marker is analyzed. A (binary) pattern within the border of the marker is extracted in the **pattern checking** step to *identify* the marker. This is important to be able to overlay different virtual content on different markers. For frame markers a simple bit code is used that supports 512 different combinations (and hence markers).

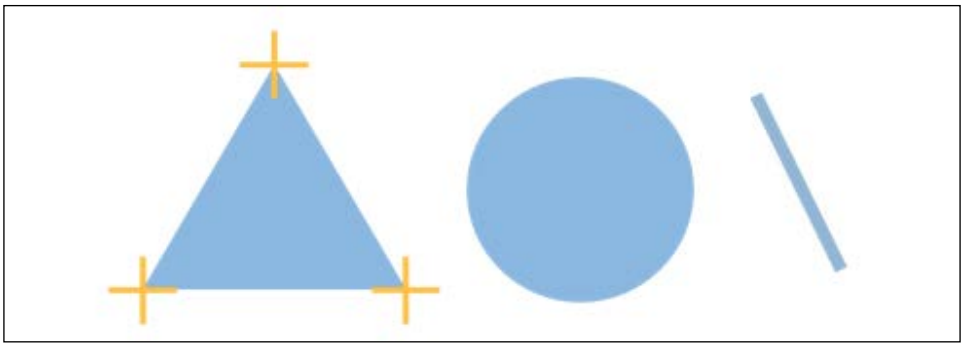
In the last step, the pose (that is the translation and rotation of the camera in the local coordinate system of the marker or reversely) is computed in the **pose estimation** step.

Pose computation, in its simplest form a *homography* (a mapping between points on two planes), can be used together with the intrinsic parameters to recover the translation and rotation of the camera.

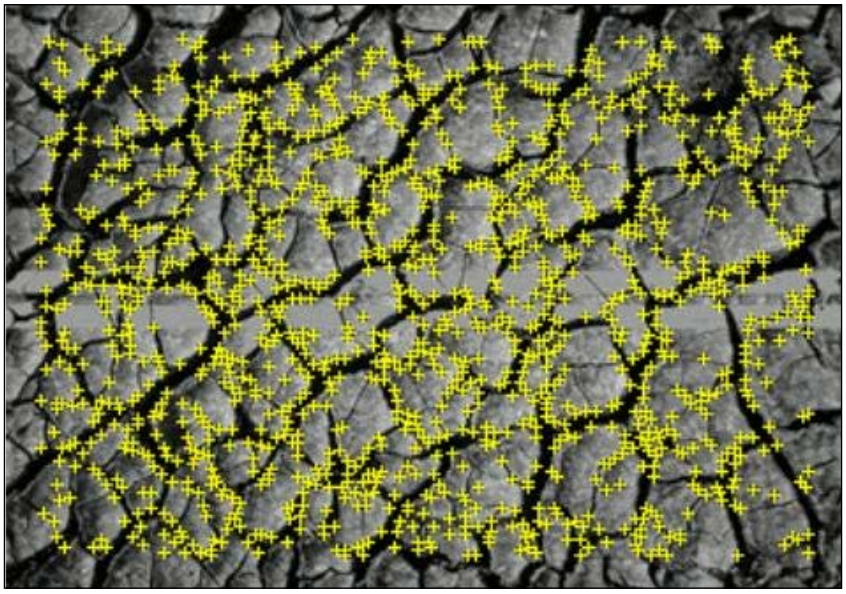
In practice, this is not a one-time computation, but rather, an iterative process in which the initial pose gets refined several times to obtain more accurate results. In order to reliably estimate the camera pose, the length of at least one side (the width or height) of the marker has to be known to the system; this is typically done through a configuration step when a marker description is loaded. Otherwise, the system could not tell reliably whether a small marker is near or a large marker is far away (due to the effects of perspective projection).

Understanding natural feature tracking targets

While the frame markers can be used to efficiently track the camera pose for many applications, you will want less obtrusive objects to track. You can achieve this by employing more advanced, but also computationally expensive, algorithms. The general idea of natural feature tracking is to use a number (in theory only three, and in practice several dozens or hundreds) of local points on a target to compute the camera pose. The challenge is that these points have to be reliable, robustly detected, and tracked. This is achieved with advanced computer vision algorithms to detect and describe the local neighborhood of an **interest point** (or feature point). Interest points have sharp, crisp details (such as corners), for example, using gradient orientations, which are suitable for feature points indicated by yellow crosses in the following figure. A circle or a straight line does not have sharp features and is not suitable for interest points:



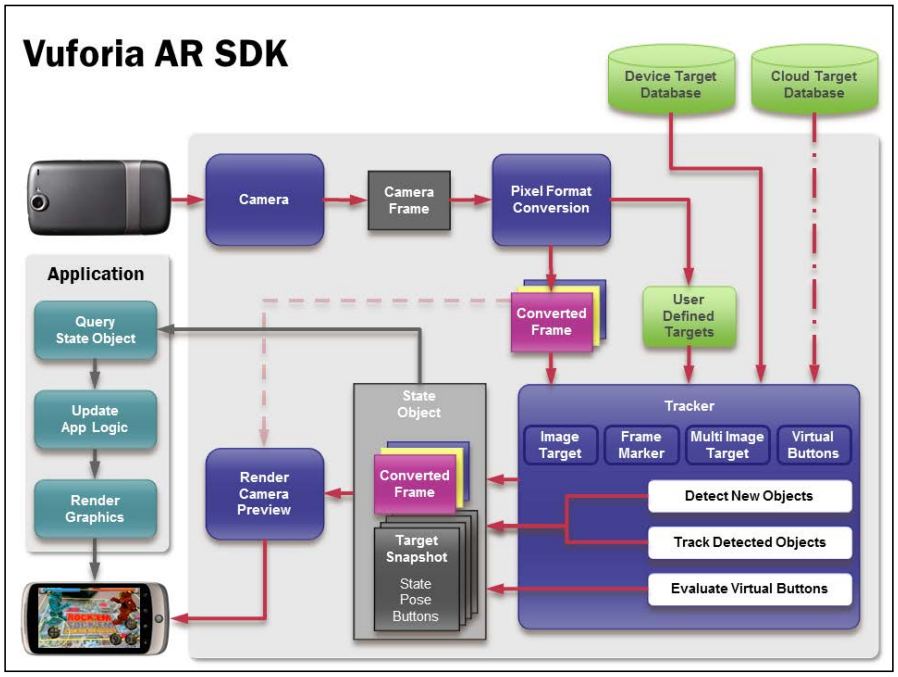
Many feature points can be found on well-textured images (such as the image of the street used throughout this chapter):



Beware that feature points cannot be well identified on images with homogenous color regions or soft edges (such as a blue sky or some computer graphics-rendered pictures).

Vuforia™ architecture

Vuforia™ is an Augmented Reality library distributed by Qualcomm® Inc. The library is free for use in non-commercial or commercial projects. The library supports frame marker and natural feature target tracking as well as multi-target, which are combinations of multiple targets. The library also features basic rendering functions (video background and OpenGL® 3D rendering), linear algebra (matrix/vector transformation), and interaction capabilities (virtual buttons). The library is actually available on both iOS and Android platforms, and the performance is improved on mobile devices equipped with Qualcomm® chipsets. An overview of the library architecture is presented in the following figure:



The architecture, from a client viewpoint (application box on the left of the preceding figure), offers a state object to the developer, which contains information about recognized targets as well as the camera content. We won't get into too much of details here as a list of samples is available on their website, along with full documentation and an active forum, at `http://developer.vuforia.com/`. What you need to know is that the library uses the **Android NDK** for its integration as it's being developed in C++.

This is mainly due to the gains of high-performance computation for image analysis or computer vision with C++ rather than doing it in Java (concurrent technologies also use the same approach). It's a drawback for us (as we are using JME and Java only) but a gain for you in terms of getting performances in your application.

To use the library, you generally need to follow these three steps:

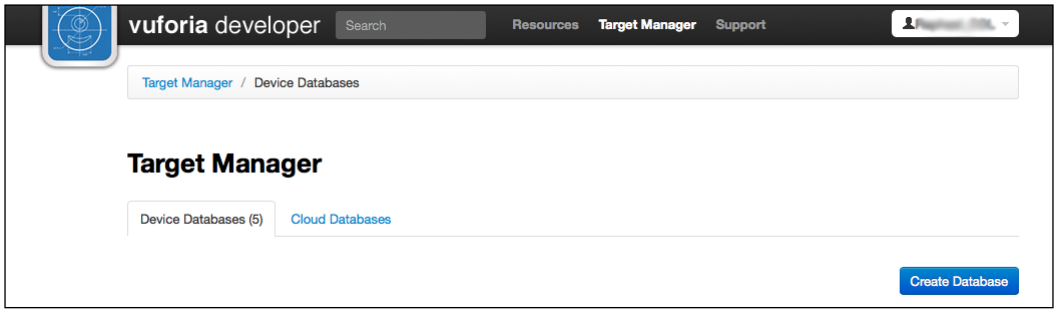
- Train and create your target or markers
- Integrate the library in your application
- Deploy your application

In the next section, we will introduce you to creating and training your targets.

Configuring Vuforia™ to recognize objects

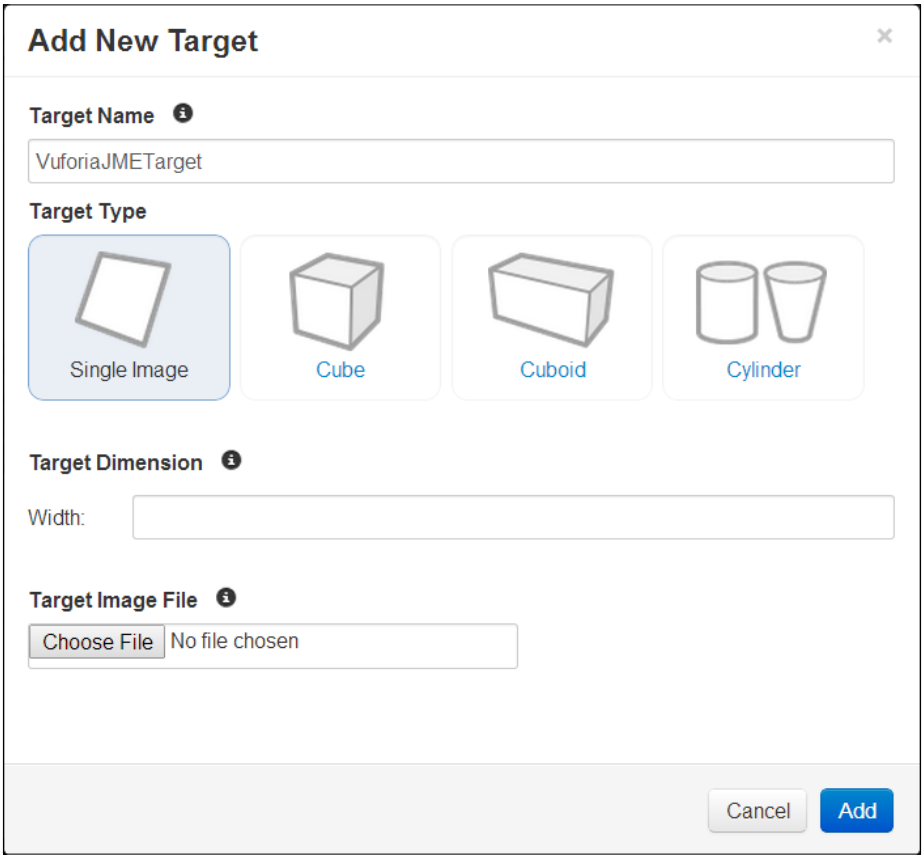
To use the Vuforia™ toolkit with natural feature tracking targets, first you need to create them. In the recent version of the library (2.0), you can automatically create your target when the application is running (online) or predefine them before deploying your application (offline). We will show you how to proceed for offline creation. First go to the Vuforia™ developer website `https://developer.vuforia.com`.

The first thing you need to do is to log in to the website to access the tool for creating your target. Click on the upper-right corner and register if you have not done it before. After login, you can click on **Target Manager**, the training program to create targets. The target manager is organized in a database (which can correspond to your project), and for database, you can create a list of targets, as shown in the following screenshot:



So let's create our first database. Click on **Create Database**, and enter `VuforiaJME`. Your database should appear in your **Device Databases** list. Select it to get onto the following page:

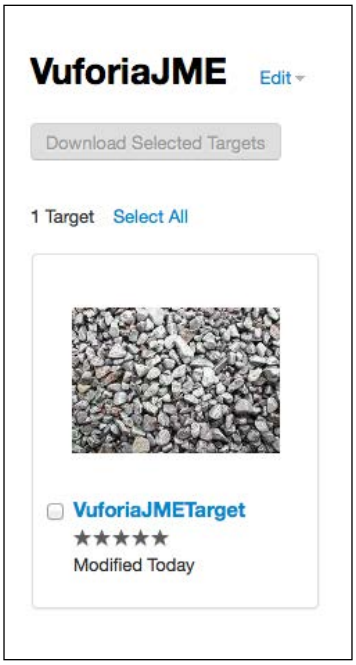
Click on **Add New Target** to create the first target. A dialog box will appear with different text fields to complete, as shown in the following screenshot:



First you need to pick up a name for your target; in our case, we will call it `VuforiaJMETarget`. Vuforia™ allows you to create different types of targets as follows:

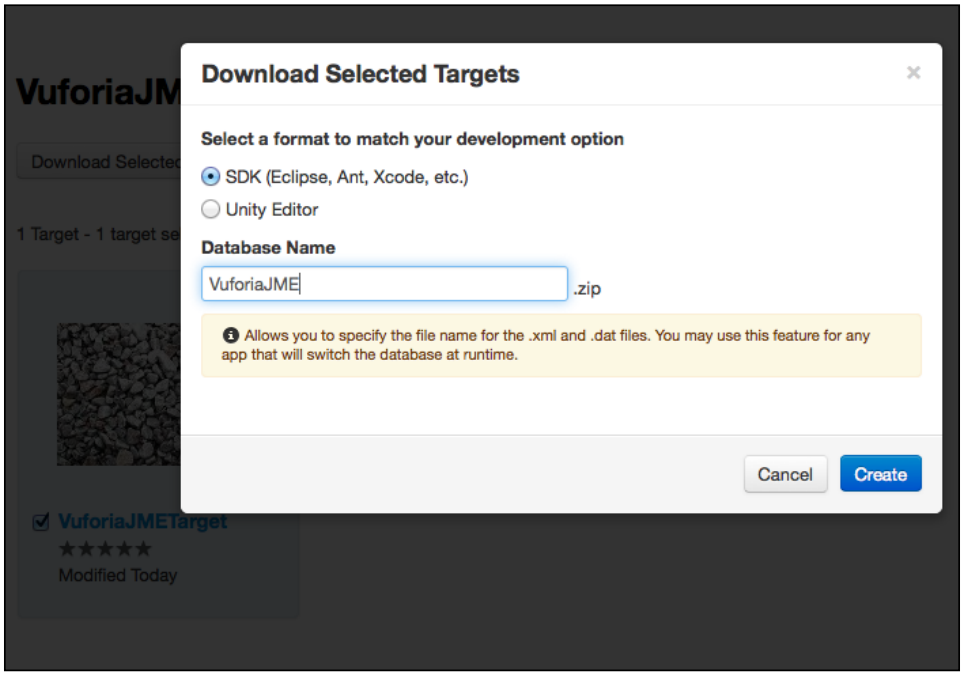
- **Single Image**: You create only one planar surface and use only one image. The target is generally used for printing on a page, part of a magazine, and so on.
- **Cube**: You define multiple surfaces (with multiple pictures), which will be used to track a 3D cube. This can be used for games, packaging, and so on.
- **Cuboid**: It's a variation of the cube type, as a parallelepiped with non-square faces.

Select **Single Image** target type. The target dimension defines a relative scale for your marker. The unit is not defined as it corresponds to the size of your virtual object. A good tip is to consider that everything is in centimeters or millimeters, which is generally the size of your physical marker (for example, print on an A4 or letter page). In our case, we enter the dimension in centimeters. Finally, you need to select an image which will be used for the target. As an example, you can select the `stones.jpg` image, which is available with the Vuforia™ sample distribution (Media directory in the *ImageTargets* example on the Vuforia™ website). To validate your configuration, click on **Add**, and wait as the image is being processed. When the processing is over, you should get a screen like the following:



The stars notify you of the quality of the target for tracking. This example has five stars, which means it will work really well. You can get more information on the Vuforia™ website on how to create a good image for a target: `https://developer.vuforia.com/resources/dev-guide/natural-features-and-rating`.

Our last step is now to export the created target. So select the target (tick the box next to **VuforiaJMETarget**), and click on **Download Selected Targets**. On the dialog box that appears, choose **SDK** for export and **VuforiaJME** for our database name, and save.



Unzip your compressed file. You will see two files: a `.dat` file and a `.xml` file. Both files are used for operating the Vuforia™ tracking at runtime. The `.dat` file specifies the feature points from your image and the `.xml` file is a configuration file. Sometimes you may want to change the size of your marker or do some basic editing without having to restart or do the training; you can modify it directly on your XML file. So now we are ready with our target for implementing our first Vuforia™ project!

Putting it together – Vuforia™ with JME

In this section we will show you how to integrate Vuforia™ with JME. We will use a natural feature-tracking target for this purpose. So open the **VuforiaJME** project in your Eclipse to start. As you can already observe, there are two main changes compared to our previous projects:

- The camera preview class is gone
- There is a new directory in the project root named `jni`

The first change is due to the way Vuforia™ manages the camera. Vuforia™ uses its own camera handle and camera preview integrated in the library. Therefore, we'll need to query the video image through the Vuforia™ library to display it on our scene graph (using the same principle as seen in *Chapter 2*, *Viewing the World*).

The `jni` folder contains C++ source code, which is required for Vuforia™. To integrate Vuforia™ with JME, we need to interoperate Vuforia's low-level part (C++) with the high-level part (Java). It means we will need to compile C++ and Java code and transfer data between them. If you have done it, you'll need to download and install the Android NDK before going further (as explained in *Chapter 1*, *Augmented Reality Concepts and Tools*).

The C++ integration

The C++ layer is based on a modified version of the **ImageTargets** example available on the Vuforia™ website. The `jni` folder contains the following files:

- `MathUtils.cpp` and `MathUtils.h`: Utilities functions for mathematical computation
- `VuforiaNative.cpp`: This is the main C++ class that interacts with our Java layer
- `Android.mk` and `Application.mk`: These contains configuration files for compilation

Open the `Android.mk` file, and check if the path to your Vuforia™ installation is correct in the `QCAR_DIR` directory. Use only a relative path to make it cross-platform (on MacOS with the android ndk r9 or higher, an absolute path will be concatenated with the current directory and result in an incorrect directory path).

Now open the `VuforiNative.cpp` file. A lot of functions are defined in the files but only three are relevant to us:

- `Java_com_ar4android_VuforiaJMEActivity_loadTrackerData(JNIEnv *, jobject)`: This is the function for loading our specific target (created in the previous section)
- `virtual void QCAR_onUpdate(QCAR::State& state)`: This is the function to query the camera image and transfer it to the Java layer
- `Java_com_ar4android_VuforiaJME_updateTracking(JNIEnv *env, jobject obj)`: This function is used to query the position of the targets and transfer it to the Java layer

The first step will be to use our specific target in our application and the first function. So copy and paste the `VuforiaJME.dat` and `VuforiaJME.xml` files to your assets directory (there should already be two target configurations). Vuforia™ configures the target that will be used based on the XMLconfiguration file. `loadTrackerData` gets first access to `TrackerManager` and `imageTracker` (which is a tracker for non-natural features):

```
JNIEXPORT int JNICALL
Java_com_ar4android_VuforiaJMEActivity_loadTrackerData(JNIEnv *,
jobject)
{
    LOG("Java_com_ar4android_VuforiaJMEActivity_ImageTargets_
      loadTrackerData");

    // Get the image tracker:
    QCAR::TrackerManager& trackerManager = QCAR::TrackerManager::
                                           getInstance();
    QCAR::ImageTracker* imageTracker = static_
      cast<QCAR::ImageTracker*>(trackerManager.
      getTracker(QCAR::Tracker::IMAGE_TRACKER));
    if (imageTracker == NULL)
    {
        LOG("Failed to load tracking data set because the
          ImageTracker has not been initialized.");
        return 0;
    }
```

The next step is to create a specific target, such as instancing a dataset. In this example, one dataset is created, named `dataSetStonesAndChips`:

```
    // Create the data sets:
    dataSetStonesAndChips = imageTracker->createDataSet();
```

```
    if (dataSetStonesAndChips == 0)
    {
        LOG("Failed to create a new tracking data.");
        return 0;
    }
```

After we load the configuration of the targets in the created instance, this is where we set up our VuforiaJME target:

```
    // Load the data sets:
    if (!dataSetStonesAndChips->load("VuforiaJME.xml",
      QCAR::DataSet::STORAGE_APPRESOURCE))
    {
        LOG("Failed to load data set.");
        return 0;
    }
```

Finally we can activate the dataset by calling the `activateDataSet` function. If you don't activate the dataset, the target will be loaded and initialized in the tracker but won't be tracked until activation:

```
    // Activate the data set:
    if (!imageTracker->activateDataSet(dataSetStonesAndChips))
    {
        LOG("Failed to activate data set.");
        return 0;
    }

    LOG("Successfully loaded and activated data set.");
    return 1;
}
```

Once we have our target initialized, we need to get the real view of the world with Vuforia™. The concept is the same as we have seen before: using a video background camera in the JME class and updating it with an image. However, here, the image is not coming from a Java `Camera.PreviewCallback` but from Vuforia™. In Vuforia™ the best place to get the video image is in the `QCAR_onUpdate` function. This function is called just after the tracker gets updated. An image can be retrieved by querying a frame on the State object of Vuforia™ with `getFrame()`. A frame can contain multiple images, as the camera image is in different formats (for example, YUV, RGB888, GREYSCALE, RGB565, and so on). In the previous example, we used the RGB565 format in our JME class. We will do the same here. So our class will start as:

```
class ImageTargets_UpdateCallback : public QCAR::UpdateCallback
{
    virtual void QCAR_onUpdate(QCAR::State& state)
    {
```

```
        //inspired from:
        //https://developer.vuforia.com/forum/faq/android-how-can-i-
          access-camera-image

QCAR::Image *imageRGB565 = NULL;
        QCAR::Frame frame = state.getFrame();
        for (int i = 0; i < frame.getNumImages(); ++i) {
              const QCAR::Image *image = frame.getImage(i);
              if (image->getFormat() == QCAR::RGB565) {
                  imageRGB565 = (QCAR::Image*)image;
                  break;
              }
        }
```

The function parses a list of images in the frame and retrieves the `RGB565` image. Once we get this image, we need to transfer it to the **Java Layer**. For doing that you can use a JNI function:

```
        if (imageRGB565) {
            JNIEnv* env = 0;

            if ((javaVM != 0) && (activityObj != 0) && (javaVM-
              >GetEnv((void**)&env, JNI_VERSION_1_4) == JNI_OK)) {

                const short* pixels = (const short*) imageRGB565-
                  >getPixels();
                int width = imageRGB565->getWidth();
                int height = imageRGB565->getHeight();
                int numPixels = width * height;

                jbyteArray pixelArray = env->NewByteArray
                  (numPixels * 2);
                env->SetByteArrayRegion(pixelArray, 0, numPixels * 2,
                  (const jbyte*) pixels);
                jclass javaClass = env->GetObjectClass(activityObj);
                jmethodID method = env-> GetMethodID(javaClass,
                  "setRGB565CameraImage", "([BII)V");
                env->CallVoidMethod(activityObj, method, pixelArray,
                  width, height);
                env->DeleteLocalRef(pixelArray);
            }
        }

};
```

In this example, we get information about the size of the image and a pointer on the raw data of the image. We use a JNI function named `setRGB565CameraImage`, which is defined in our `Java Activity` class. We call this function from C++ and pass in argument the content of the image (`pixelArray`) as `width` and `height` of the image. So each time the tracker updates, we retrieve a new camera image and send it to the Java layer by calling the `setRGB565CameraImage` function. The JNI mechanism is really useful and you can use it for passing any data, from a sophisticated computation process back to your Java class (for example, physics, numerical simulation, and so on).

The next step is to retrieve the location of the targets from the tracking. We will do that from the `updateTracking` function. As before, we get an instance of the State object from Vuforia™. The State object contains `TrackableResults`, which is a list of the identified targets in the video image (identified here as being recognized as a target and their position identified):

```
JNIEXPORT void JNICALL
Java_com_ar4android_VuforiaJME_updateTracking(JNIEnv *env,
  jobject obj)
{
    //LOG("Java_com_ar4android_VuforiaJMEActivity_GLRenderer_
      renderFrame");

    //Get the state from QCAR and mark the beginning of a rendering
      section
    QCAR::State state = QCAR::Renderer::getInstance().begin();

    // Did we find any trackables this frame?
    for(int tIdx = 0; tIdx < state.getNumTrackableResults(); tIdx++)
    {
        // Get the trackable:
        const QCAR::TrackableResult* result = state.
          getTrackableResult(tIdx);
```

In our example, we have only one target activated, so if we get a result, it will obviously be our marker. We can then directly query the position information from it. If you had multiple activated markers, you will need to identify which one is which, by getting information from the result by calling `result->getTrackable()`.

The position of `trackable` is queried by calling `result->getPose()`, which returns a matrix defining a linear transformation. This transformation gives you the position of the marker relative to the camera position. Vuforia™ uses a computer-vision coordinate system (x on the left, y down, and z away from you), which is different from JME, so we will have to do some conversion later on. For now, what we will do first is inverse the transformation, to get the position of the camera relative to the marker; this will make the marker the reference coordinate system for our virtual content. So you will do some basic mathematical operations as follows:

```
QCAR::Matrix44F modelViewMatrix = QCAR::Tool::
  convertPose2GLMatrix(result->getPose());

QCAR::Matrix44F inverseMV = MathUtil::
  Matrix44FInverse(modelViewMatrix);
QCAR::Matrix44F invTranspMV = MathUtil::
  Matrix44FTranspose(inverseMV);

float cam_x = invTranspMV.data[12];
float cam_y = invTranspMV.data[13];
float cam_z = invTranspMV.data[14];

float cam_right_x = invTranspMV.data[0];
float cam_right_y = invTranspMV.data[1];
float cam_right_z = invTranspMV.data[2];
float cam_up_x = invTranspMV.data[4];
float cam_up_y = invTranspMV.data[5];
float cam_up_z = invTranspMV.data[6];
float cam_dir_x = invTranspMV.data[8];
float cam_dir_y = invTranspMV.data[9];
float cam_dir_z = invTranspMV.data[10];
```

Now we have the location (`cam_x,y,z`) as well as the orientation of our camera (`cam_right_/cam_up_/cam_dir_x,y,z`).

The last step is to transfer this information to the Java layer. We will use JNI again for this operation. What we also need is information about the internal parameters of our camera. This is similar to what was discussed in *Chapter 3*, *Superimposing the World*, but now it has been done here with Vuforia™. For that, you can access the `CameraCalibration` object from `CameraDevice`:

```
float nearPlane = 1.0f;
float farPlane = 1000.0f;
const QCAR::CameraCalibration& cameraCalibration = QCAR::
  CameraDevice::getInstance().getCameraCalibration();
```

```
QCAR::Matrix44F projectionMatrix = QCAR::Tool::
  getProjectionGL(cameraCalibration, nearPlane, farPlane);
```

We can easily transform the projection transformation to a more readable format for the camera configuration, such as its field of view (`fovDegrees`), which we also have to adapt to allow for differences in the aspect ratios of the camera sensor and the screen:

```
QCAR::Vec2F size = cameraCalibration.getSize();
QCAR::Vec2F focalLength = cameraCalibration.getFocalLength();
float fovRadians = 2 * atan(0.5f * size.data[1] /
  focalLength.data[1]);
float fovDegrees = fovRadians * 180.0f / M_PI;
float aspectRatio=(size.data[0]/size.data[1]);

float viewportDistort=1.0;
if (viewportWidth != screenWidth)     {
  viewportDistort = viewportWidth / (float) screenWidth;
    fovDegrees=fovDegrees*viewportDistort;
    aspectRatio=aspectRatio/viewportDistort;
}
if (viewportHeight != screenHeight)  {
  viewportDistort = viewportHeight / (float) screenHeight;
    fovDegrees=fovDegrees/viewportDistort;
    aspectRatio=aspectRatio*viewportDistort;
}
```

We then call three JNI functions to transfer the field of view (`setCameraPerspectiveNative`), camera position (`setCameraPoseNative`) and camera orientation (`setCameraOrientationNative`) to our Java layer. These three functions are time defined in the `VuforiaJME` class, which allows us to quickly modify our virtual camera:

```
jclass activityClass = env->GetObjectClass(obj);
        jmethodID setCameraPerspectiveMethod = env->GetMethodID
          (activityClass,"setCameraPerspectiveNative", "(FF)V");
        env->CallVoidMethod(obj,setCameraPerspectiveMethod,
          fovDegrees,aspectRatio);
        jmethodID setCameraViewportMethod = env->GetMethodID
          (activityClass,"setCameraViewportNative", "(FFFF)V");
        env->CallVoidMethod(obj,setCameraViewportMethod,viewportWidth,
          viewportHeight,cameraCalibration.getSize().
          data[0],cameraCalibration.getSize().data[1]);
```

```
        // jclass activityClass = env->GetObjectClass(obj);
         jmethodID setCameraPoseMethod = env->GetMethodID
           (activityClass,"setCameraPoseNative", "(FFF)V");
         env->CallVoidMethod(obj,setCameraPoseMethod,cam_x,cam_y,
           cam_z);

         //jclass activityClass = env->GetObjectClass(obj);
         jmethodID setCameraOrientationMethod = env->GetMethodID
           (activityClass,"setCameraOrientationNative", "(FFFFFFFFF)V");
         env->CallVoidMethod(obj,setCameraOrientationMethod,
           cam_right_x,cam_right_y,cam_right_z,
         cam_up_x,cam_up_y,cam_up_z,cam_dir_x,cam_dir_y,cam_dir_z);

      }

      QCAR::Renderer::getInstance().end();
}
```

The last step will be to compile the program. So run a command shell, and go the `jni` directory containing the files. From there you need to call the `ndk-build` function. The function is defined in your `android-ndk-r9d` directory, so be sure it's accessible from your path. If everything goes well, you should see the following:

```
Install        : libQCAR.so => libs/armeabi-v7a/libQCAR.so
Compile++ arm  : VuforiaNative <= VuforiaNative.cpp
SharedLibrary  : libVuforiaNative.so
Install        : libVuforiaNative.so => libs/armeabi-v7a/
                 libVuforiaNative.so
```

Time to go back to Java!

The Java integration

The Java layer defines the function previously called using similar classes from our *Superimpose* example. The first function is the `setRGB565CameraImage` function which handles the video image as in the previous examples.

The other JNI functions will modify the characteristics of our foreground camera. Specifically, we modify the left axis of the JME camera to match the coordinate system used by Vuforia™ (as depicted in the figure in the *Choosing physical objects* section).

```
  public void setCameraPerspectiveNative(float fovY,float aspectRatio)
{
            fgCam.setFrustumPerspective(fovY,aspectRatio, 1, 1000);
  }
```

```
public void setCameraPoseNative(float cam_x,float cam_y,float cam_z)
{
         fgCam.setLocation(new Vector3f(cam_x,cam_y,cam_z));
}

public void setCameraOrientationNative(float cam_right_x,float
  cam_right_y,float cam_right_z,
float cam_up_x,float cam_up_y,float cam_up_z,float cam_dir_x,
  float cam_dir_y,float cam_dir_z) {
     //left,up,direction
     fgCam.setAxes(new Vector3f(-cam_right_x,-cam_right_y,
       -cam_right_z),
       new Vector3f(-cam_up_x,-cam_up_y,-cam_up_z),
       new Vector3f(cam_dir_x,cam_dir_y,cam_dir_z));
}
```

Finally, we have to adjust the viewport of the background camera, which shows the camera image, to prevent 3D objects from floating above the physical target:

```
public void setCameraViewportNative(float viewport_w,float
viewport_h,float size_x,float size_y) {
      float newWidth = 1.f;
      float newHeight = 1.f;

      if (viewport_h != settings.getHeight())
      {
        newWidth=viewport_w/viewport_h;
        newHeight=1.0f;
        videoBGCam.resize((int)viewport_w,(int)viewport_h,true);
        videoBGCam.setParallelProjection(true);
      }
      float viewportPosition_x =  (((int)(settings.getWidth()  -
        viewport_w)) / (int) 2);//+0
      float viewportPosition_y =  (((int)(settings.getHeight() -
        viewport_h)) / (int) 2);//+0
      float viewportSize_x = viewport_w;//2560
      float viewportSize_y = viewport_h;//1920

      //transform in normalized coordinate
      viewportPosition_x =  (float)viewportPosition_x/(float)
        viewport_w;
      viewportPosition_y =  (float)viewportPosition_y/(float)
        viewport_h;
```

```
        viewportSize_x = viewportSize_x/viewport_w;
        viewportSize_y = viewportSize_y/viewport_h;

      //adjust for viewport start (modify video quad)
          mVideoBGGeom.setLocalTranslation(-0.5f*newWidth+
            viewportPosition_x,-0.5f*newHeight+viewportPosition_y,0.f);
      //adust for viewport size (modify video quad)
      mVideoBGGeom.setLocalScale(newWidth, newHeight, 1.f);
    }
```

And that's it. What we want to outline again here is the concept behind it:

- The camera model used in your tracker is matched with your virtual camera (in this example `CameraCalibration` from Vuforia™ to our JME Virtual Camera). This will guarantee us a correct registration.
- You track a target in your camera coordinate system (in this example, a natural feature target from Vuforia™). This tracking replaces our GPS as seen previously, and uses a local coordinate system.
- The position of this target is used to modify the pose of your virtual camera (in this example, transferring the detected position from C++ to Java with JNI, and updating our JME Virtual Camera). As we repeat the process for each frame, we have a full 6DOF registration between physical (the target) and virtual (our JME scene).

Your results should look similar to the one in the following figure:

Summary

In this chapter, we introduced you to computer vision-based AR. We developed an application with the Vuforia™ library and showed you how to integrate it with JME. You are now ready to create natural feature tracking-based AR applications. In this demo, you can move your device around the marker and see the virtual content from every direction. In the next chapter, we will learn how we can do more in terms of interaction. How about being able to select the model and play with it?

6
Make It Interactive – Create the User Experience

Over the course of the previous chapters, we've learned the essentials of creating augmentations using the two most common AR approaches: sensor-based and computer vision-based AR. We are now able to overlay digital content over a view of the physical world, support AR tracking, and handle account registration (on a target or outdoors).

However, we can merely navigate the augmented world around them. Wouldn't it be cool to allow the users to also interact with the virtual content in an intuitive way? User interaction is a major component in the development of any application. As we are focusing here on the user interaction with 3D content (3D interaction), the following are three main categories of interaction techniques that can be developed:

- **Navigation**: Moving around a scene and selecting a specific viewpoint. In AR, this navigation is done by physical motion (such as, walking on the street or turning a table) and can be complemented with additional virtual functions (for example, map view, navigation path, freeze mode, and so on).
- **Manipulation**: Selecting, moving, and modifying objects. In AR, this can be done on physical and virtual elements, through a range of traditional methods (for example, ray picking), and novel interaction paradigms (for example, tangible user interfaces).
- **System control**: Adapting the parameters of your application, including rendering, polling processes, and application dependent content. In AR, it can correspond to adjusting tracking parameters or visualization techniques (for example, presenting the distance to your POI in an AR Browser).

In this chapter we will show you a subset of some of the commonly used AR interaction techniques. We will show you how to develop three interaction techniques, including ray picking, proximity-based interaction, and 3D motion gesture-based interaction. This is the next step in designing an AR Application and a fundamental brick in our AR layer (See *Chapter 1*, *Augmented Reality Concepts and Tools*).

Pick the stick – 3D selection using ray picking

3D interaction on desktop computers made use of a limited set of devices, including the keyboard, mouse, or joystick (for games). On a smartphone (or tablet), interaction is mainly driven by touch or sensor input. From an interaction input (the sensor data, such as x and y coordinates on the screen, or the event type, such as click or dwell), you can develop different interaction techniques, such as ray picking, steering navigation, and so on. For mobile AR, a large set of interaction techniques can be used for 2D or 3D interactions. In this section, we will look at using touch input combined with a technique named **ray picking**.

The concept of ray picking is to use a virtual ray going from your device to your environment (which is the target) and detect what it hits along the way. When you get a hit on some object (for example, the ray intersects with one of your virtual characters), you can consider this object picked (selected) and start to manipulate it. Here, we will only look at how we can pick an object in JME. In the sample code, you can extend the object to support further manipulation, for example, when an object is hit and picked, you can detect sliding touch motion and translate the object, make it explode, or use the hit as a shooting ray for some game, and so on.

So let's start. In JME, you can use either an Android-specific function for the input (via `AndroidInput`) or the same one you would use in a desktop application (`MouseInput`). JME on Android, by default, maps any touch event as a mouse event that allows us to have (almost) the same code working on Android and your desktop. We will choose the following solution for this project; as an exercise, you can try to use `AndroidInput` (look into `AndroidTouchInputListener` to use `AndroidInput`).

Open the `RayPickingJME` example. It's using the same base code as `VuforiaJME` and our picking method is based on a JME example, for this picking method please visit the following link: `http://jmonkeyengine.org/wiki/doku.php/jme3:beginner:hello_picking`.

The first thing to do is add the different packages necessary for ray picking in our `RayPickingJME` class:

```
import com.jme3.math.Ray;
import com.jme3.collision.CollisionResult;
import com.jme3.collision.CollisionResults;
import com.jme3.input.MouseInput;
import com.jme3.input.controls.ActionListener;
import com.jme3.input.controls.KeyTrigger;
import com.jme3.input.controls.MouseButtonTrigger;
```

To be able to pick an object, we need to declare some global variables in the scope of our `RayPicking` class:

- `Node shootables`
- `Geometry geom`

The next step is to add a listener to our class. If you've never done Android or JME programming, you may not know what a listener is. A **listener** is an event handler technique that can listen to any of the activities happening in a class and provide specific methods to handle any event. For example, if you have a mouse button click event, you can create a listener for it that has an `onPushEvent()` method where you can install your own code. In JME, event management and listeners are organized into two components, controlled by using the `InputManager` class:

- **Trigger mapping**: Using this you can associate the device input can with a trigger name, for example, clicking on the mouse can be associated with `Press` or `Shoot` or `MoveEntity`, and so on.
- **Listener**: Using this you can associate the trigger name with a specific listener; either `ActionListener` (used for discrete events, such as "button pressed") or `AnalogListener` (used for continuous events, such as the amplitude of a joystick movement).

So, in your `simpleInitApp` procedure, add the following code:

```
inputManager.addMapping("Shoot",      // Declare...
  newKeyTrigger(KeyInput.KEY_SPACE), // trigger 1: spacebar, or
  newMouseButtonTrigger(0));         // trigger 2: left-button
    click
inputManager.addListener(actionListener, "Shoot");
```

So, here, we map the occasions when the spacebar is pressed (even when using a virtual keyboard) and mouse click (which is a touch action on our mobile) to the trigger name `Shoot`. This trigger name is associated with the `ActionListener` event listener that we've named `actionListener`. The action listener will be where we do the ray picking; so, on a touchscreen device, by touching the screen, you can activate `actionListener` (using the trigger `Shoot`).

Our next step is to define the list of objects that can potentially be hit by our ray picking. A good technique for that is to regroup them under a specific group node. In the following code, we will create a box object and place it under a group node named `shootables`:

```
Box b = new Box(7, 4, 6); // create cube shape at the origin
geom = new Geometry("Box", b);  // create cube geometry from the shape
Material mat = new Material(assetManager,
"Common/MatDefs/Misc/Unshaded.j3md");  // create a simple material
mat.setColor("Color", ColorRGBA.Red);   // set color of material to
blue
geom.setMaterial(mat);        // set the cube's material
geom.setLocalTranslation(new Vector3f(0.0f,0.0f,6.0f));

shootables = new Node("Shootables");
shootables.attachChild(geom);
rootNode.attachChild(shootables);
```

Now we have our touch mapping and our objects that can be hit. We only need to implement our listener. The way to ray cast in JME is similar to that in many other libraries; we use the hit coordinates (defined in the screen coordinates), transform them using our camera, create a ray, and run a hit. In our AR example, we will use the AR camera, which is updated by our computer vision-based tracker `fgCam`. So, the code is the same in AR as for another virtual game, except that here, our camera position is updated by the tracker.

We create a `Ray` object and run a picking test (hitting test) by calling `collideWith` for our list object that can be hit (`shootables`). Results of the collision will be stored in a `CollisionResults` object. So, our listener's code looks like the following code:

```
privateActionListeneractionListener = new ActionListener() {

public void onAction(String name, booleankeyPressed, float tpf)
  {
    Log.d(TAG,"Shooting.");
```

```
        if (name.equals("Shoot") && !keyPressed) {

          // 1. Reset results list.
          CollisionResults results = new CollisionResults();

          // 2. Mode 1: user touch location.
          Vector2f click2d = inputManager.getCursorPosition();
          Vector3f click3d = fgCam.getWorldCoordinates(
          new Vector2f(click2d.x, click2d.y), 0f).clone();
          Vector3f dir = fgCam.getWorldCoordinates(
          new Vector2f(click2d.x, click2d.y),
            1f).subtractLocal(click3d).normalizeLocal();
          Ray ray = new Ray(click3d, dir);

          // 2. Mode 2: using screen center
          //Aim the ray from fgcamloc to fgcam direction.
          //Ray ray = new Ray(fgCam.getLocation(),
            fgCam.getDirection());

          // 3. Collect intersections between Ray and Shootables in
            results list.
          shootables.collideWith(ray, results);
…
```

So, what do we do with the result? As explained earlier in the book, you can manipulate it in a different way. We will do something simple here; we will detect whether or not our box is selected, and if it is, change its color to red for no intersection and green if there is an intersection. We first print the results for debugging, where you can use the `getCollision()` function to detect which object has been hit (`getGeometry()`), at what distance (`getDistance()`), and the point of contact (`getContactPoint()`):

```
    for (int i = 0; i<results.size(); i++) {
      // For each hit, we know distance, impact point, name of
        geometry.
      floatdist = results.getCollision(i).getDistance();
      Vector3f pt = results.getCollision(i).getContactPoint();
      String hit = results.getCollision(i).getGeometry().getName();

      Log.d(TAG,"* Collision #" + i + hit);
      //         Log.d(TAG,"  You shot " + hit + " at " + pt + ", "
        + dist + "wu away.");
    }
```

So, by using the preceding code we can detect whether or not we have any result, and since we only have one object in our scene, we consider that if we got a hit, it's our object, so we change the color of the object to green. If we don't get any hit, since there is only our object, we turn it red:

```
if (results.size() > 0) {
  // The closest collision point is what was truly hit:
CollisionResult closest = results.getClosestCollision();
closest.getGeometry().getMaterial().setColor("Color",
  ColorRGBA.Green);
} else {
  geom.getMaterial().setColor("Color", ColorRGBA.Red);
}
```

You should get a result similar to that shown in the following screenshot (hit: left, miss: right):

You can now deploy and run the example; touch the object on the screen and see our box changing color!

Proximity-based interaction

Another type of interaction in AR is using the relation between the camera and a physical object. If you have a target placed on a table and you move around with your device to look at a virtual object from different angles, you can also use that to create interaction. The idea is simple: you can detect any change in spatial transformation between your camera (on your moving device) and your target (placed on a table), and trigger some events. For example, you can detect if the camera is under a specific angle, if it's looking at the target from above, and so on.

In this example, we will implement a **proximity** technique that can be used to create creating some cool animation and effects. The proximity technique uses the distance between the AR camera and a computer vision-based target.

So, open the `ProximityBasedJME` project in your Eclipse. Again, this project is also based on the `VuforiaJME` example.

First, we create three objects—a box, a sphere, and a torus—using three different colors—red, green and blue—as follows:

```
Box b = new Box(7, 4, 6); // create cube shape at the origin
geom1 = new Geometry("Box", b);  // create cube geometry from
  the shape
Material mat = new Material(assetManager,
  "Common/MatDefs/Misc/Unshaded.j3md");  // create a simple
  material
mat.setColor("Color", ColorRGBA.Red);   // set color of
  material to red
geom1.setMaterial(mat);                   // set the cube's
  material

geom1.setLocalTranslation(new Vector3f(0.0f,0.0f,6.0f));

rootNode.attachChild(geom1);              // make the cube
  appear in the scene

Sphere s = new Sphere(12,12,6);
geom2 = new Geometry("Sphere", s);  // create sphere geometry
  from the shape
Material mat2 = new Material(assetManager,
  "Common/MatDefs/Misc/Unshaded.j3md");  // create a simple
  material
mat2.setColor("Color", ColorRGBA.Green);   // set color of
  material to green
geom2.setMaterial(mat2);                   // set the sphere's
  material

geom2.setLocalTranslation(new Vector3f(0.0f,0.0f,6.0f));

rootNode.attachChild(geom2);              // make the sphere
  appear in the scene

Torus= new Torus(12, 12, 2, 6); // create torus shape at
  the origin
geom3 = new Geometry("Torus", t);  // create torus geometry
  from the shape
Material mat3 = new Material(assetManager,
  "Common/MatDefs/Misc/Unshaded.j3md");  // create a simple
  material
mat3.setColor("Color", ColorRGBA.Blue);   // set color of
  material to blue
geom3.setMaterial(mat3);                   // set the
  torus material
```

```
geom3.setLocalTranslation(new Vector3f(0.0f,0.0f,6.0f));

rootNode.attachChild(geom3);              // make the torus
  appear in the scene
```

In a large number of scene graph libraries, you will often find a switch node that allows the representation of an object based on some parameters to be switched, such as the distance from the object to the camera. JME doesn't have a switch node, so we will simulate its behavior. We will change which object is displayed (box, sphere, or torus) as a function of its distance from the camera. The simple way to do that is to add or remove objects that shouldn't be displayed at a certain distance.

To implement the proximity technique, we query the location of our AR camera (`fgCam.getLocation()`). From that location, you can compute the distance to some objects or just the target. The distance to the target is, by definition, similar to the distance of the location (expressed as a vector with three dimensions) of the camera. So, what we do is define three ranges for our object as follows:

- **Camera distance 50 and more**: Shows the cube
- **Camera distance 40-50**: Shows the sphere
- **Camera distance under 40**: Shows the torus

The resulting code in the `simpleUpdate` method is rather simple:

```
Vector3f pos=new Vector3f();

pos=fgCam.getLocation();

if (pos.length()>50.)
{
  rootNode.attachChild(geom1);
  rootNode.detachChild(geom2);
  rootNode.detachChild(geom3);
}
else
  if (pos.length()>40.)
  {
    rootNode.detachChild(geom1);
```

```
    rootNode.attachChild(geom2);
    rootNode.detachChild(geom3);
  },
  else
  {
    rootNode.detachChild(geom1);
    rootNode.detachChild(geom2);
    rootNode.attachChild(geom3);
  }
```

Run your example and change the distance of the device to that of the tracking target. This will affect the object which is presented. A cube will appear when you are far away (as shown on the left-hand side of the following figure), torus when you are close (as shown on the right-hand side of the following figure), and a sphere in between (as shown in the center of the following figure):

Simple gesture recognition using accelerometers

In *Chapter 4, Locating in the World,* you were introduced to the various sensors that are built into the typical Android device. You learned how to use them to derive the orientation of your device. However, there is much more you can do with those sensors, specifically accelerometers. If you have ever played Wii games, you were surely fascinated by the natural interaction you could achieve by waving the Wiimote around (for example, when playing a tennis or golf Wii game). Interestingly, the Wiimote uses similar accelerometers to many Android smartphones, so you can actually implement similar interaction methods as with the Wiimote. For complex 3D-motion gestures (such as drawing a figure eight in the air), you will need either some machine learning background or access to use libraries such as the one at the following link: `http://www.dfki.de/~rnessel/tools/gesture_recognition/gesture_recognition.html`. However, if you only want to recognize simple gestures, you can do that easily in a few lines of code. Next, we will show you how to recognize simple gestures such as a shake gesture, that is, quickly waving your phone back and forth several times.

If you have a look at the sample project `ShakeItJME`, you will notice that it is, to a large degree, identical to the `SensorFusionJME` project from *Chapter 4*, *Locating in the World*. Indeed, we only need to perform a few simple steps to extend any application that already uses accelerometers. In `ShakeItJMEActivity`, you first add some variables that are relevant for the shake detection, including mainly variables for storing timestamps of accelerometer events (`mTimeOfLastShake`, `mTimeOfLastForce`, and `mLastTime`), ones for storing past accelerometer forces (`mLastAccelValX`, `mLastAccelValY`, and `mLastAccelValZ`), and a number of thresholds for shake durations, timeouts (`SHAKE_DURATION_THRESHOLD`, `TIME_BETWEEN_ACCEL_EVENTS_THRESHOLD`, and `SHAKE_TIMEOUT`), and a minimum number of accelerometer forces and sensor samples (`ACCEL_FORCE_THRESHOLD` and `ACCEL_EVENT_COUNT_THRESHOLD`).

Next, you simply add a call to the `detectShake()` method in your `SensorEventListener::onSensorChanged` method in the `Sensor.TYPE_ACCELEROMETER` section of code.

The `detectShake()` method is at the core of your shake detection:

```
public void detectShake(SensorEvent event) {
…
  floatcurAccForce = Math.abs(event.values[2] - mLastAccelValZ) /
    timeDiff;
  if (curAccForce> ACCEL_FORCE_THRESHOLD) {
    mShakeCount++;
    if ((mShakeCount>= ACCEL_EVENT_COUNT_THRESHOLD) && (now -
      mTimeOfLastShake> SHAKE_DURATION_THRESHOLD)) {
      mTimeOfLastShake = now;mShakeCount = 0;
      if ((com.ar4android.ShakeItJME) app != null) {
        ((com.ar4android.ShakeItJME) app).onShake();
      }
    }
…
  }
}
```

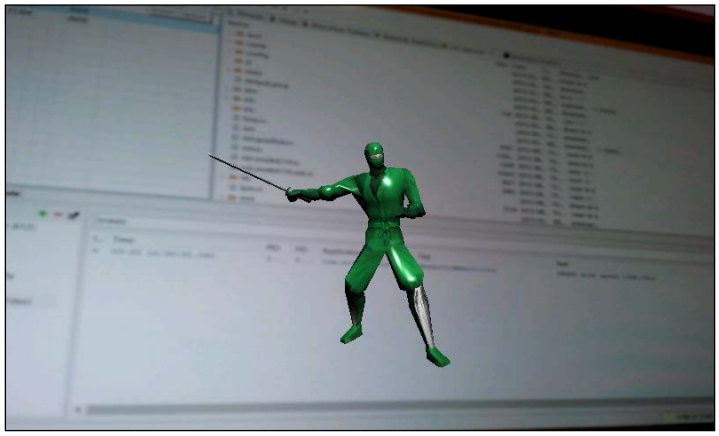
In this method, you basically check whether or not accelerometer values in a certain time frame are greater than the threshold value. If they are, you call the `onShake()` method of your JME app and integrate the event into your application `logic.Note` that, in this example, we only use the accelerometer values in the z direction, that is, parallel to the direction in which the camera is pointing. You can easily extend this to also include sideways shake movements by incorporating the x and y values of the accelerometer in the computation of `curAccForce`. As an example of how to trigger events using shake detection, in the `onShake()` method of your JME application, we trigger a new animation of our walking ninja:

```
public void onShake() {
  mAniChannel.setAnim("Spin");
}
```

To avoid that the ninja now spins all the time; we will switch to the walking animation after the spin animation has ended:

```
public void onAnimCycleDone(AnimControl control, AnimChannel
  channel, String animName) {
if(animName.contains("Spin")) {
    mAniChannel.setAnim("Walk");
  }
}
```

If you start your app now and shake the device along the viewing direction, you should see how the ninja stops walking and makes a gentle spin, just as shown in the following figure:



Summary

In this chapter, we've introduced you to three interaction techniques, suitable for a wide variety of AR applications. Picking allows you to select 3D objects by touching the screen, just like you would in 2D selection. Proximity-based camera techniques allow you to experiment with the distance and orientation of your device to trigger application events. Finally, we've showed you a simple example of a 3D gesture detection method to add even more interaction possibilities into your application. These techniques should serve as building blocks for you to create your own interaction methods, targeted to your specific application scenario. In the final chapter, we will introduce some advanced techniques and further reading to help you get the best out of your Augmented Reality applications.

7
Further Reading and Tips

In this final chapter, we will present you with tips and links to more advanced techniques to improve any AR application's development. We will introduce content management techniques such as multi-targets and cloud recognition, as well as advanced interaction techniques.

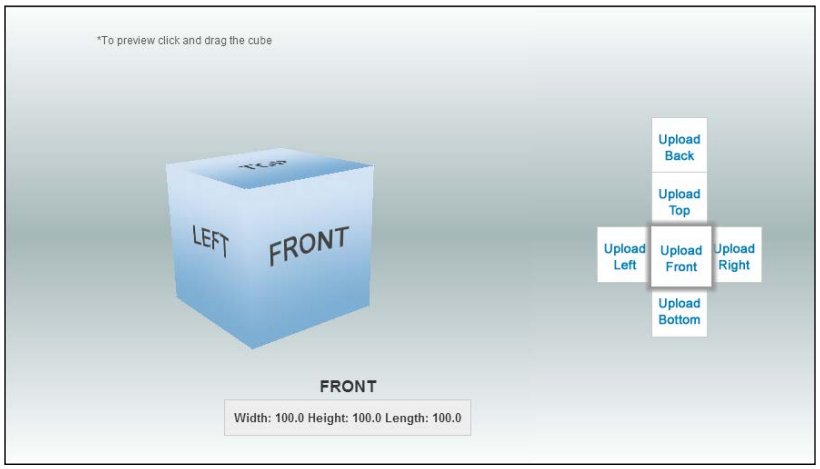
Managing your content

For computer-vision-based AR, we showed you how to build applications using a single target. However, there might be scenarios in which you need to use several markers at once. Just think of augmenting a room for which you would need at least one target on each wall, or you may want your application to be able to recognize and augment hundreds of different product packages. The former case can be achieved by tracking multiple targets that have a common coordinate frame, and the latter use case can be achieved by using the power of cloud recognition. We will briefly discuss both of them in the following sections.

Multi-targets

Multi-targets are more than a collection of several individual images. They realize a single and consistent coordinate system where a handheld device can be tracked. This allows for continuous augmentation of the scene as long as even a single target is visible. The main challenges of creating multi-targets lie in defining the common coordinate system (which you will do only once) and maintaining the relative poses of those targets during the operation of the device.

To create a common coordinate system, you have to specify the translation and orientation of all image targets with respect to a common origin. Vuforia™ gives you an option to even build commonly used multi-targets such as cubes or cuboids without getting into the details of specifying the entire target transforms. In the Vuforia™ Target Manager, you can simply add a cube (equal length, height, and width) or cuboids (different length, height, and width) to a target that has its coordinate origin at the (invisible) center of the cuboids. All you have to do is to specify one extend to three extends of the cuboids and add individual images for all the sides of your targets, as shown in the following figure:



If you want to create more complex multi-targets, for example, for tracking an entire room, you have to take a slightly different approach. You first upload all the images you want to use for the multi-target into a single device database inside the Vuforia™ Target Manager. After, you have downloaded the device database to your development machine, you can then modify the downloaded `<database >.xml` file to add the names of the individual image targets and their translations and orientations relative to the coordinate origin. A sample XML file can be found in the Vuforia™ knowledge base under `https://developer.vuforia.com/resources/dev-guide/creating-multi-target-xml-file`.

Note that you can only have a maximum of 100 targets in your device database, and hence your multi-target can maximally consist of only that number of image targets. Also note that changing the position of image targets during the runtime (for example, opening a product packaging) will inhibit consistent tracking of your coordinate system, that is, the defined spatial relationships between the individual target elements would not be valid anymore. This can even lead to complete failure of tracking. If you want to use individual moving elements as part of your application, you have to define them in addition to the multi-target as separate image targets.

Cloud recognition

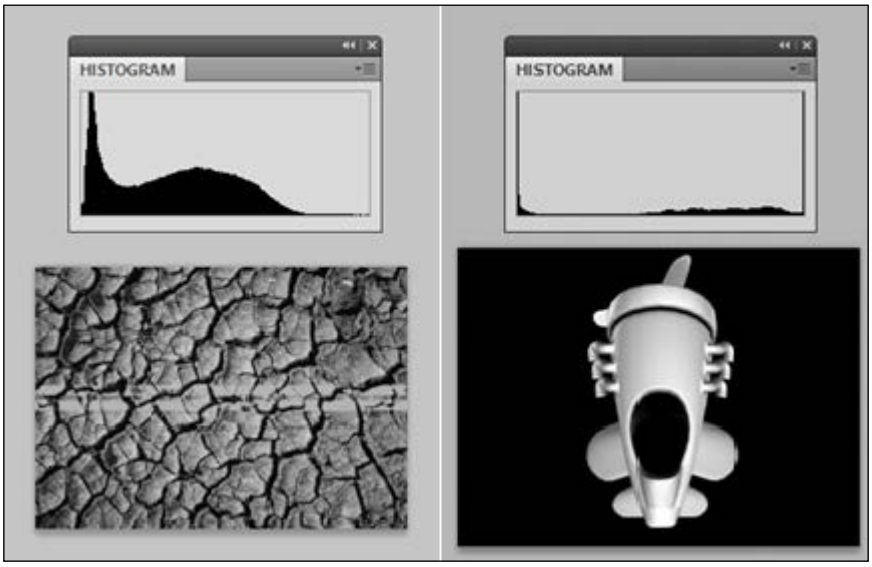
As mentioned in the preceding section, you can only use up to 100 images simultaneously in your Vuforia™ application. This limitation can be overcome by using cloud databases. The basic idea here is that you query a cloud service with a camera image, and (if the target is recognized in the cloud), handle the tracking of the recognized target locally on your device. The major benefit of this approach is that you can recognize up to one million images that should be sufficient for most application scenarios. However, this benefit does not come for free. As the recognition happens in the cloud, your client has to be connected to the Internet, and the response time can take up to several seconds (typically around two to three seconds).

Unlike, in the case of recognition, image databases stored on the device typically only take about 60 to 100 milliseconds. To make it easier to upload many images for the cloud recognition, you do not even have to use the Vuforia™ online target manager website but can use a specific web API—the Vuforia™ Web Services API—that can be found under the following URL: `https://developer.vuforia.com/resources/dev-guide/managing-targets-cloud-database-using-developer-api`. You can find further information about using cloud recognition in the Vuforia™ knowledge base by visiting `https://developer.vuforia.com/resources/dev-guide/cloud-targets`.

Improving recognition and tracking

If you want to create your own natural feature-tracking targets, it is important to design them in a way that they can be well recognized and tracked by the AR system. The basics of natural feature targets were explained in the *Understanding natural feature tracking targets* section of *Chapter 5*, *Same as Hollywood – Virtual on Physical Objects*. The basic requirement for well-traceable targets is that they possess a high number of local features. But how do you go along if your target is not well recognized? To a certain extent, you can improve the tracking by using the forthcoming tips.

First, you want to make sure that your images have enough local contrast. A good indicator for the overall contrast in your target is to have a look at the histogram of its greyscale representation in any photo editing software such as GIMP or Photoshop. You generally want a widely distributed histogram instead of one with few spikes, as shown in the following figure:



To increase the local contrast in your images, you can use the photo editor of your choice and apply unsharpening mask filters or clarity filters, such as in Adobe Lightroom.

In addition, to avoid resampling artifacts in the Vuforia™ target creation process, make sure to upload your individual images with an exact image width of 320 px. This will avoid aliasing effects and lowering the local feature count due to automatic server-side resizing of your images. By improving the rendering, Vuforia™ will rescale your images to have a maximum extend of 320 px for the longest image side.

During the course of this book, we used different types of 3D models in our sample applications, including basic primitives (such as our colored cube or sphere) or more advanced 3D models (such as the ninja model). For all of them, we didn't really consider the realistic aspect, including the light condition. Any desktop or mobile 3D application will always consider how the rendering looks realistic. This photorealistic quest always passes through the quality of the geometry of the model, the definition of their appearance (material reflectance properties), and how they interact with light (shading and illumination).

Photorealistic rendering will expose properties such as occlusion (what is in front of, behind something), shadows (from the illumination), support for a range of realistic material (developed with shader technology), or more advanced properties such as supporting global illumination.

When you develop your AR applications, you should also consider photorealistic rendering. However, things are a bit more complicated because in AR, you not only consider the virtual aspect (for example, a desktop 3D game) but also the real aspect. Supporting photorealism in AR will imply that you consider **how real (R**) **environments and virtual** (**V**) **environments** also interact during the rendering that can be simplified as follows through four different cases:

1. V→V
2. V→R
3. R→V
4. R→R

The easiest thing you can do is support V→V, which means that you enable any of the advanced rendering techniques in your 3D rendering engine. For computer-vision-based applications, it will mean that everything looks realistic on your target. For sensor-based applications, it will mean that your virtual object seems realistic between each other.

A second easy step, especially for computer-vision-based applications, is to support V→R using a plane technique. If you have a target, you can create a semi-transparent version of it and add it to your virtual scene. If you have shadows enabled, it will seem that the shadow is projecting on to your target, creating a simple illusion of V→R. You can refer to the following paper which will provide you with some technical solutions to this problem:

- Refer to *A real-time shadow approach for an augmented reality application using shadow volumes. VRST 2003: 56-65* by *Michael Haller*, *Stephan Drab*, and *Werner Hartmann*.

Handling R→V is a bit more complicated and still a difficult research topic. For example, support illumination of virtual objects by physical light sources requires a lot of effort.

Instead, occlusion is easy to implement for R→V. Occlusion in the case of R→V can happen if, for example, a physical object (such as a **can**) is placed in front of your virtual object. In standard AR, you always render the virtual content in front of the video, so your **can** will appear to be behind even though it can be in front of your target.

A simple technique to reproduce this effect is sometimes referred to as **phantom object**. You need to create a virtual counterpart of your physical object, such as a cylinder, to represent your can. Place this virtual counterpart at the same position as the physical one and do a **depth-only rendering**. Depth-only rendering is available in a large range of libraries, and it's related to the color mask where, when you render anything, you can decide which channel to render. Commonly, you have the combination of red, green, blue, and depth. So, you need to deactivate the first three channels and only activate depth. It will render some sort of phantom object (no color but only depth), and via the standard rendering pipeline, the video will not be occluded anymore where you have your real object, and occlusion will look realistic; see, for example, `http://hal.inria.fr/docs/00/53/75/15/PDF/occlusionCollaborative.pdf`.

This is the simple case; when you have a dynamic object, things are way more complicated, and you need to be able to track your objects, to update their phantom models, and to be able to get a photorealistic rendering.

Advanced interaction techniques

In the preceding chapter, we looked at some simple interaction techniques, that included ray picking (via touch interaction), sensor interaction, or camera to target proximity. There are a large number of other interaction techniques that can be used in Augmented Reality.

One standard technique that we will also find on other mobile user interfaces, is a **virtual control pad**. As a mobile phone limits access to additional control devices, such as a joypad or joystick, you can emulate their behavior via a touch interface. With this technique, you can display a virtual controller on your screen and analyze the touch in this area as being equivalent to controlling a control pad. It's easy to implement and enhance the basic ray-casting technique. Control pads are generally displayed near the border of the screen, adapting to the form factor and grasping the gesture you make when you hold the device, so you can hold the device with your hand and naturally move your finger on the screen.

Another technique that is really popular in Augmented Reality is **Tangible User Interface** (**TUI**). When we created the sample using the concept of a camera to target proximity, we practically implemented a Tangible User Interface. The idea of a TUI is to use a physical object for supporting interaction. The concept was largely developed and enriched by *Iroshi Ishii* from the Tangible Media Group at MIT—the website to refer to is `http://tangible.media.mit.edu/`. *Mark Billinghurst* during his Ph.D. applied this concept to Augmented Reality and demonstrated a range of dedicated interaction techniques with it.

The first type of TUI AR is **local interaction**, where you can, for example, use two targets for interaction. Similar to the way we detected the distance between the camera and target in our `ProximityBasedJME` project, you can replicate the same idea with two targets. You can detect whether two targets are close to each other, aligned in the same direction, and trigger some actions with it. You can use this type of interaction for card-based games when you want cards to interact with each other, or games that include puzzles where users need to combine different cards together, and so on.

A second type of TUI AR is **global interaction** where you will also use two or more targets, but one of the targets will become *special*. What you do in this case is define a target as being a base target, and all the other targets refer to it. To implement it, you just compute the local transformation of the other targets to the base target, with the base target behind and defined as your origin. With this, it's really easy to place targets on the main target, somehow defining some kind of ground plane and performing a range of different types of interaction with it. *Mark Billinghurst* introduced a famous derivate version of it, for performing paddle-based interaction. In this case, one of the targets is used as a paddle and can be used to interact on the ground plane—you can touch the ground plane, have the paddle at a specific position on the ground plane, or even detect a simple gesture with it (shake the paddle, tilt the paddle, and so on). To set up mobile AR, you need to consider the fact that end users hold a device and can't perform complex gestures, but with a mobile phone, interaction with one hand is still possible. Refer to the following technical papers:

- *Tangible augmented reality. ACM SIGGRAPH ASIA (2008): 1-10* by *Mark Billinghurst*, *Hirokazu Kato*, and *Ivan Poupyrev*.
- *Designing augmented reality interfaces. ACM Siggraph Computer Graphics 39.1 (2005): 17-22* by *Mark Billinghurst*, *Raphael Grasset*, and *Julian Looser*.

Global interaction with a TUI, in a sense, can be defined as interaction *behind the screen*, while virtual control pad can be seen as interaction *in front of the screen*. This is another way to classify interaction with a mobile, which brings us to the third category of interaction techniques: **touch interaction on the target**. The Vuforia™ library implements, for example, the concept of virtual buttons. A specific area on your target can be used to place the controller (for example, buttons, sliders, and dial), and users can place their finger on this area and control these elements. The concept behind this uses a time-based approach; if you keep your finger placed on this area for a long time, it simulates a click that you can have on a computer, or a tap you can do on a touch screen. Refer to `https://developer.vuforia.com/resources/sample-apps/virtual-button-sample-app`, for example.

There are other techniques that are investigated in research laboratories, and they will soon become available to the future generation of mobile AR, so you should already think about them also when will be available. One trend is towards 3D gesture interaction or also called **mid-air interaction**. Rather than touching your screen or touching your target, you can imagine making gestures between the device and the target. Having a mobile AR for 3D modeling would be an appropriate technique. 3D gestures have a lot of challenges such as recognizing the hand, the fingers, the gesture, physical engagement that can result in fatigue, and so on. In the near future, this type of interaction, which is already popular on smart home devices (such as Microsoft Kinect), will be available on devices (equipped with 3D sensors).

Summary

In this chapter, we showed you how to go beyond the standard AR applications by using multi-targets or cloud recognition for computer-vision-based AR. We also showed you how you can improve the tracking performance for your image targets. In addition, we introduced you to some advanced rendering techniques for your AR applications. Finally, we also showed you some novel interaction techniques that you can use to create great AR experiences. This chapter concludes your introduction to the world of Augmented Reality development for Android. We hope you are ready to progress onto new levels of AR application development.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

Thank you for buying

Augmented Reality for Android Application Development

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

About Packt Open Source

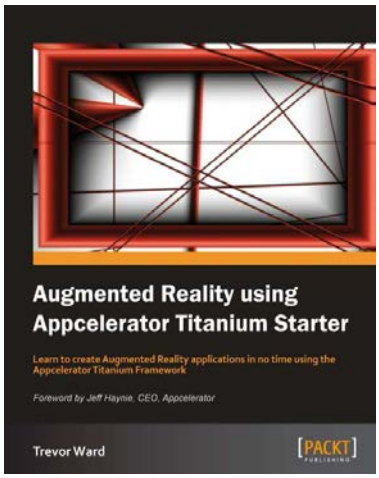
In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

[PACKT] open source
community experience distilled
PUBLISHING

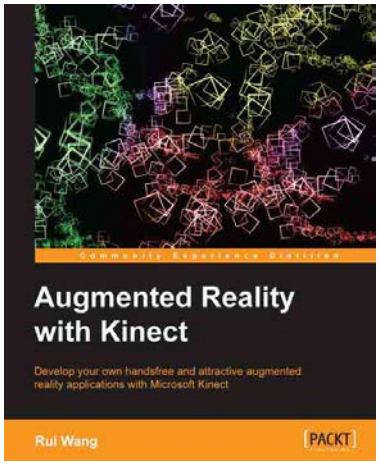


Augmented Reality using Appcelerator Titanium Starter [Instant]

ISBN: 978-1-84969-390-5 Paperback: 52 pages

Learn to create Augmented Reality applications in no time using the Appcelerator Titanium Framework

1. Learn something new in an Instant! A short, fast, focused guide delivering immediate results.
2. Create an open source Augmented Reality Titanium application
3. Build an effective display of multiple points of interest

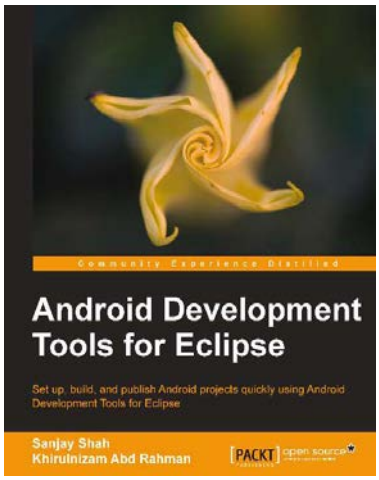


Augmented Reality with Kinect

ISBN: 978-1-84969-438-4 Paperback: 122 pages

Develop tour own handsfree and attractive augmented reality applications with Microsoft Kinect

1. Understand all major Kinect API features including image streaming, skeleton tracking and face tracking
2. Understand the Kinect APIs with the help of small examples
3. Develop a comparatively complete Fruit Ninja game using Kinect and augmented Reality techniques

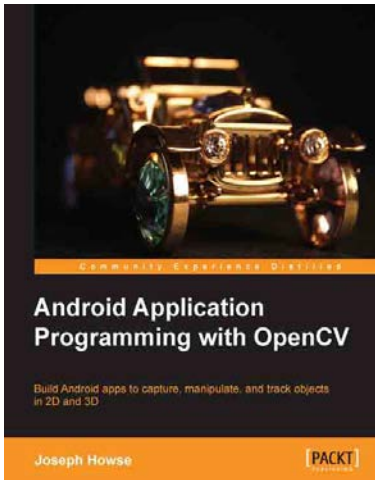


Android Development Tools for Eclipse

ISBN: 978-1-78216-110-3 Paperback: 144 pages

Set up, build, and publish Android projects quickly using Android Development Tools for Eclipse

1. Build Android applications using ADT for Eclipse
2. Generate Android application skeleton code using wizards
3. Advertise and monetize your applications



Android Application Programming with OpenCV

ISBN: 978-1-84969-520-6 Paperback: 130 pages

Build Android apps to capture, manipulate, and track objects in 2D and 3D

1. Set up OpenCV and an Android development environment on Windows, Mac, or Linux
2. Capture and display real-time videos and still images
3. Manipulate image data using OpenCV and Apache Commons Math