ulm university universität ulm

Universität Ulm | 89069 Ulm | Germany

Faculty of Engineering and Computer Science
Institute of Databases and Information Systems

Implementation and evaluation of a mobile Android application for auditory stimulation of chronic tinnitus patients

Master's thesis at Ulm University

Submitted by:
Jan-Dominik Blome
jan-dominik.blome@uni-ulm.de

Reviewers:
Prof. Dr. Manfred Reichert
Dr. Winfried Schlee

Supervisor:
Marc Schickler

2015

Version October 21, 2015

Abstract

Tinnitus is a common symptom where the affected person perceives a sound without an external source. To support the development of new therapies a tinnitus tracking platform, including mobile applications, was developed at Ulm University in cooperation with the tinnitus research initiative. In the future, these mobile applications should be extended to include a simple game that requires the user to concentrate on an auditory stimulation, distracting them from their tinnitus. This is accomplished by using localization of an audio source as a game mechanic. The measurement of the offset between the position the user guessed for an audio source and its actual location could also serves as an additional data point. In this thesis an application for the Android operating system is designed that implements such a game and serves as a proof of concept. Since the Android API does not include the capability for positional audio, a separate audio API based on OpenAL was created as part of this thesis. This API as well as the framework developed to implement the game are designed to be reusable for future, similar projects. The game concept was also evaluated in a study using the demonstration application.

Contents

1 Introduction **1**

1.1 Goal of this Thesis 1

1.2 Tinnitus 2

1.3 Structure of this Thesis 3

2 Requirements Specification **5**

2.1 Functional 5

2.2 Non-Functional 8

3 Audio Fundamentals **9**

3.1 Digital Audio 9

3.2 Binaural Audio 13

3.3 Head-Related Transfer Functions 16

4 Architecture **19**

4.1 Project Structure 19

4.2 Build System 22

4.2.1 Gradle 22

4.2.2 CMake 24

4.3 External Dependencies 25

4.3.1 Java Dependencies 26

4.3.2 Native Dependencies 27

4.4 Documentation 28

5 Framework **29**

5.1 Utility Libraries 29

5.2 Audio API 31

5.2.1 Basic Classes 32

5.2.2 Positional Audio API . . . 33
5.2.3 Abstract Helper Classes . . . 36
5.2.4 Audio Filters . . . 39
5.2.5 Audio Utilities . . . 42
5.3 Sensor API . . . 44
5.3.1 Rotation Sensor API . . . 45
5.3.2 Sensor Data Filters . . . 47
5.3.3 Available Sensor Implementations . . . 49

6 OpenAL **53**
6.1 The OpenAL Library . . . 53
6.2 Build Process . . . 54
6.3 Configuration . . . 55
6.4 Audio API Implementation . . . 58

7 Game Engine **61**
7.1 Architecture . . . 61
7.1.1 Structure . . . 62
7.1.2 Components . . . 63
7.2 Engine . . . 64
7.2.1 Components API . . . 65
7.2.2 Game Data Classes . . . 70
7.2.3 Utility Classes . . . 73
7.3 Graphics . . . 75
7.3.1 Rendering Process . . . 77
7.3.2 Panorama Renderer . . . 82
7.3.3 Entity Renderer . . . 82
7.4 Audio . . . 84
7.5 Input . . . 87

8 Application **91**
8.1 Common Application Classes . . . 91
8.2 Demo Application . . . 93

9 Evaluation **97**
9.1 Participants and Methods . . . 97
9.2 Results . . . 98

9.3 Comparison 101

10 Conclusion **105**

10.1 Results 105

10.2 Requirements Comparison 106

10.3 Future Work 108

List of Figures **109**

List of Source Codes **111**

List of Tables **113**

List of Acronyms **115**

Bibliography **117**

1 Introduction

Tinnitus, that is the perception of a sound without an external source, is a common problem for a significant number of people [8]. In 2013 a mobile and web application to track individual tinnitus perception was developed at Ulm University in cooperation with the tinnitus research initiative (TRI) [44]. This application, called Track Your Tinnitus, allows people affected by a tinnitus to regularly record the severeness of their tinnitus perception. Collecting this data over the course of several weeks can reveal regular patterns that in turn might give clues to the cause of the increased severity. Besides allowing the user to track their own tinnitus, the application also enables researchers to access the anonymized data to further research the tinnitus symptom [44].

1.1 Goal of this Thesis

In this thesis, the foundation necessary to extend the Android version of the Track Your Tinnitus application with a game that requires the users to concentrate on a specific

sound played through their headphones will be developed. Using binaural audio the source of the sound can be placed at a virtual location around the listener. The goal of the game is to locate this sound source. To further challenge the user the difficulty of this task may be increased by adding additional sound sources or ambient audio. The data gathered from a game session could then for example show if it is possible to distract the user from their tinnitus using this method. By measuring the offset between the actual position of the sound source and the position guessed by the user the game could also show if their ability for audio localization is affected. In this thesis the framework necessary to create such a game for the Android operating system will be developed and demonstrated in an application. This application will also be evaluated using a study. This framework can then serve as a basis to integrate the described or similar games into the Android version of the tinnitus tracking application, enabling it to be used for further data collection.

1.2 Tinnitus

Over the course of their lives many, if not most, people will experience a tinnitus [8]. The term tinnitus (from the Latin word *tinnire*, to ring) describes the perception of a sound without an external sources. In some cases the sound has a physical source inside the body of the affected person, it is then called an objective tinnitus. But in case of the more prevalent type of tinnitus, the subjective tinnitus, there is no physical source of the noise [49]. This perceived sound can vary for each affected person, and it is typically describes as a pure tone, a hissing or a roaring noise [67]. It can be localized to the left or right ear, or it can appear to originate from the middle of the head [51]. Most of the time the tinnitus was triggered by an external source, like loud music or medication, and vanishes by itself over a time span of between a few seconds and up to a few days [8]. This type of tinnitus is called a transient tinnitus. If the tinnitus lasts more than six months it is called a chronic tinnitus. The chronic type of tinnitus is experienced by between 5 and 15 percent of the general population [67].

It is easy to imagine that such a condition can impact the quality of life of affected individuals. Effects can include sleep disturbances, difficulties to concentrate and psychiatric distress. About 1 to 3% of the general population report to be affected by a tinnitus strong enough to have these or other effects reduce their quality of life [67].

1.3 Structure of this Thesis

After the introduction, this thesis will detail the design, implementation and evaluation of the developed application. As first part of this process, the requirements for the project, both functional and non-functional, are specified in chapter 2. Since the auditory stimulation using positional audio is a core concept of the application, the basics of such a system are explained in chapter 3. Then, in chapter 4, the overall structure of the project is explained. Besides an overview of the included modules, this chapter also details the build process and the external dependencies of the application.

The project is split into different modules to improve reusability. The modules making up the basic framework of the application are described in chapter 5. This includes a positional audio application programming interface (API) developed for this project. To implement this API, the OpenAL library is used. How this library was integrated into the application is described in chapter 6. Since the application is designed like a game, a basic game engine was implemented as part of the project. It is described in chapter 7. Finally, the actual application is detailed in chapter 8.

During development, a study was carried out to compare the implementation described in this thesis with similar applications developed for other systems. The results of this study are explained in chapter 9. Finally, in chapter 10, the final state of the project is described and compared to the requirements. The chapter concludes with an outlook over possible future improvements.

2

Requirements Specification

This chapter defines the requirements for the project. Both functional and non-functional requirements are specified. A comparison between these requirements and the implemented functionality of the application is done in section 10.2.

2.1 Functional

This section defines the functional requirements for both the application as well as the underlying framework. These requirements describe the expected behavior of the individual systems.

FR1 **Android application for auditory stimulation**
The main goal of this work is to create an application that lets the user react to auditory stimulations in the form of positional audio. The application should be designed as a game in order to motivate the user.

FR2 Positional audio sources

The application uses positional audio as a game mechanic. Since positional audio support is not part of the Android API this feature has to be implemented as part of this project. The used audio framework should also support multiple simultaneous playing audio sources. This allows for the design of more challenging scenarios.

FR3 Ambient audio sources

In addition to the positional audio sources described in FR2, the audio framework should allow the use of at least one ambient audio track. Unlike the other audio sources this track is unaffected by the actions of the player. This feature could also be used to make the game more challenging.

FR4 Per-source audio volume control

Both positional and ambient audio sources should have individual audio volume control. This allows fine-tuning the challenge presented to the player when creating a game scenario.

FR5 Audio file format

The application should support at least one commonly used compressed audio file format. This reduces the space required by the used audio files and shrinks the overall size of the application package ditributed to the users.

FR6 Audio framework

The capabilities implemented in FR2 to FR5 should be packaged as an audio framework that is usable independently of the actual application. This would allow it to be used in the future to implement different types of applications for Android based on positional audio.

FR7 Rotation detection using sensors

The application should be able to use the rotation sensors available on many Android devices to control the orientation of the player in the game. This could make the game more intuitively controllable by allowing the user to point the device into the direction they are facing and having the game react accordingly.

FR8 Alternative input using the touch screen

The application should support an alternative input scheme that does not depend on the rotation sensors of the device. This can be useful when either the device does not posses the required sensors or the current environment of thr user does not allow for free rotation. In that case the application should still be controllable by using the touchscreen of the device.

FR9 Graphics for the game

The application should be able to visualize the game world. This provides a way of showing the results directly to the user after they locked in their guess for the direction from which the target sound originated.

FR10 Picture representation of the target

When visualizing the game world, the application should be able to display a static picture representing the target at the correct location relative to the user. The picture should be changeable, so it can be chosen to match the sound that is played from that target.

FR11 Panorama picture as backdrop

The application should be able to draw a panorama picture as backdrop for the game world visualization. This panorama should be cropped to only show the parts visible from the current orientation of the player.

FR12 Reusable game engine

The systems described in FR7 to FR11 should be packaged as a game engine that can be used independently of the application. This facilitates the extension of the application with different game modes and allows these systems to be reused in another application.

FR13 Display results to the user

After the user has located a target, the difference between the guessed and the real position, as an angle in degrees, should be displayed to the user. In addition, the shown results should include the time it took the user before committing to a guess.

2.2 Non-Functional

This section defines the non-functional requirements, also for both the application and the underlying framework. These requirements describe characteristics of the individual systems not covered by the functional requirements.

NFR1 **Backwards compatibility**
The application and by extension the used libraries should work and be usable on all Android versions down to 2.3.3, or API level 10. This allows the application to be used on about 99.7% of all Android devices that access the Google Play Store [20].

NFR2 **Small application package**
Because mobile phones have limited storage capacity and application might have to be downloaded using mobile data connections the final package of the application should be kept small. It also should include all necessary assets to allow the application to work without requiring a network connection.

NFR3 **Code documentation**
All public classes and methods should have documentation in the form of JavaDoc comments. This increases their reusability by providing documentation directly inside the code.

NFR4 **Extensibility and reusability**
The application and its libraries should be designed with extensibility and reusability in mind. The goal of this requirement is to simplify future changes to the game and to allow the underlying systems to be reused to create a different application. This requirement is a more general version of the functional requirements FR6 and FR12.

NFR5 **Good audio localization**
The goal of the game designed in this thesis is for the user to localize an audio source by its sound only. For this to be possible the quality of the positional audio framework created as part of this application (see FR6) should be maximized.

3

Audio Fundamentals

The design of the application includes the blind localization of a virtual audio source as a key concept. To allow the user to determine the position of a target only by its sound, a positional audio system has to be used. In this section some concepts of digital audio as well as techniques that may be used to generate positional audio will be described.

3.1 Digital Audio

Sound, as perceived by humans, is a pressure wave in a medium, usually air. The frequencies contained in the wave determine the tonal content. Using a microphone such a sound wave can be converted to an electrical signal. This signal is still analog, to transform it into a digital format it needs to be sampled. The sampling happens by taking measurements of the signal amplitude at regular time intervals. This sampling rate has to be at least double the maximum frequency that should be recorded [4]. Since

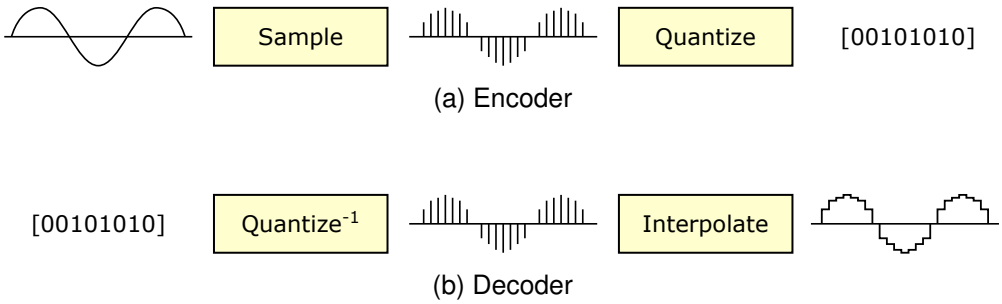


Figure 3.1: A PCM encoder and decoder [4]

humans typically can hear tones up to 20 kHz, a typical sampling rate is 44.1 kHz, used for example on audio CDs. The amplitude of each sample then has to be quantized to a binary value. Common binary representations are 16 bit signed integers, for example used on audio CDs, or 32 bit floating point numbers. This type of encoding is called pulse code modulation (PCM) [4]. Both a PCM encoder and a decoder are shown in Figure 3.1. If the audio signal contains multiple channels they are usually stored interleaved. This makes streaming of the data easier since all output channels can be fed from the same signal. By reversing the encoding process using a PCM decoder the data can again be transformed into an analog electrical signal that can be played back using a speaker.

It is also possible to transform a digital audio signal. This is done by a filter using digital signal processing (DSP). Since the audio data consists of discrete samples, such a filter is applied for each sample individually. A generic signal filter is shown in Figure 3.2, where x_n is the input sample and y_n is the transformed output sample [2]. While the shown filter has exactly one input and one output signal this is not a requirement for a DSP system. A system could, for example, have several input signals and combine them to one output signal or create several output signals for a single input signal. They can also be combined by connecting the output of one system to the input of another.

Each DSP system is constructed from some common operations that are performed on a signal. In the following the operations used by filters and systems described in this thesis will be introduced. The first is the multiplication operator, shown in Figure 3.3. It

Figure 3.2: A generic digital signal filter

multiplies all incoming samples with a value. This scales the input signal by the supplied factor, called g in the figure. For audio signals this is the equivalent of a volume control knob [2].

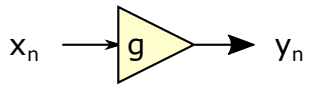


Figure 3.3: Multiplication of a signal with g

The summation operator, shown in Figure 3.4, combines two signals by adding their respective samples. For this to work the signals must use the same sampling rate. A gain factor is often applied to one or both input signals to prevent clipping of the produced signals [2].

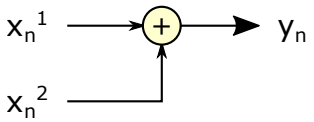


Figure 3.4: Summation of two signals

The final building block is the delay operator, shown in Figure 3.5. This operator buffers any incoming samples, up to a set delay value (k in the figure). Once the next sample is received, the oldest sample in the buffer is removed, becoming the new output value. Then the newly received sample is added instead. The buffer works on the first in, first out (FIFO) principle. If no samples are available yet, the output of this operator is the value 0 [2].

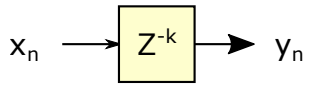


Figure 3.5: Delay of a signal by k samples

A common pattern used in DSP filters is called convolution. It can be thought of as an operation working on two arrays, resulting in a new output array. This can be written as $y(n) = x(n) * h(n)$ where $x(n)$ is the input array, $h(n)$ is the filter array and $y(n)$ is the output array. Each element of the output array is calculated by summing the last

$$y_n = \sum_{i=0}^{N-1} x_{n-i} \cdot h_i \tag{3.1}$$

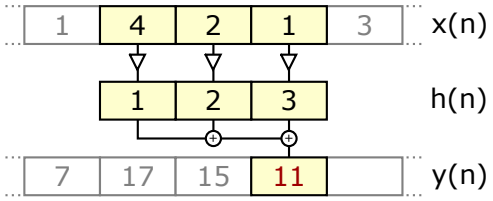


Figure 3.6: Calculation of y_n as the result of the convolution of $x(n)$ and $h(n)$

N elements of the input array $x(n)$, each weighted by a corresponding factor stored in $h(n)$, where N is the length of the $h(n)$ array. This calculation for a single element of the output array is also shown in Equation 3.1 and visualized in Figure 3.6. The resulting output array $y(n)$ has the length of $x(n)$ plus the length of $h(n)$ minus 1. [2]

The convolution operation can be realized as a DSP system, as shown in Figure 3.7. This system is called a finite impulse response (FIR) filter. It is finite because the length of the output $y(n)$ can be determined from the length of $x(n)$ and $h(n)$ [2]. The filter uses delay operations to keep the previous values in a buffer. They are then available to calculate the next output value. Each new output shifts all values down to the next delay operation. The output of these delay operations is multiplied with the associated value of $h(n)$ and finally added with all the other values to generate the result.

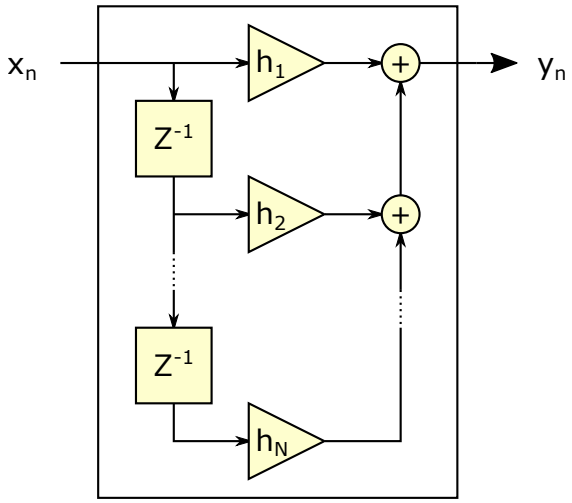


Figure 3.7: A finite impulse response filter

It is also possible to use a past output value of a DSP system as input by storing it using a delay operation. This is called a feedback loop. Using multiple delay operations allows using output values from further into the past. Unlike with the FIR filter, the length of

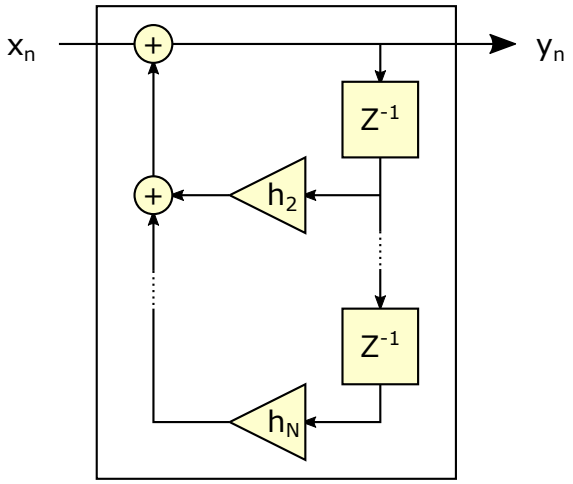


Figure 3.8: An infinite impulse response filter

the output of such a filter can not be determined from the length of the input. For that reason this type of filter is called infinite impulse response (IIR) filter. An example for such a filter is shown in Figure 3.8.

3.2 Binaural Audio

Humans possess the ability of sound localization, that is determining the position of an object based solely on the sound it is emitting. This is possible because the brain can analyze the differences between the audio heard by both ears. In this thesis, the term 3D audio refers to a system that can use this mechanism to create the illusion of a virtual, sound-emitting object for a user. The terms positional or binaural audio are also used synonymously. Such a system has to be able to direct different audio signals to each ear of the listener, otherwise no spatial localization is possible. This can easily be accomplished by using stereo headphones.

If both the consumed content as well as the position of the user are static, a binaural recording can be performed. This is done using a head, either a mannequin or the real head of the recording technician, that is equipped with a microphone in each ear [52]. The result of such a recording is limited in that the recorded content is static and the listener can only experience the acoustics of the position the recording was done from. For interactive applications this is often not an option. They need a system that can generate the appropriate audio signals for both ears based on a virtual world.

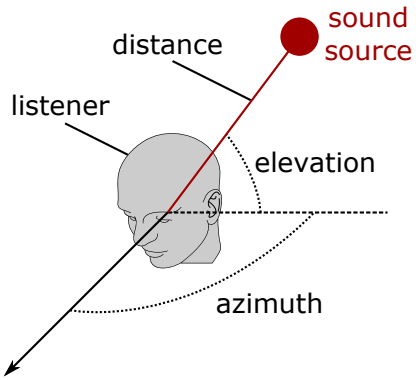
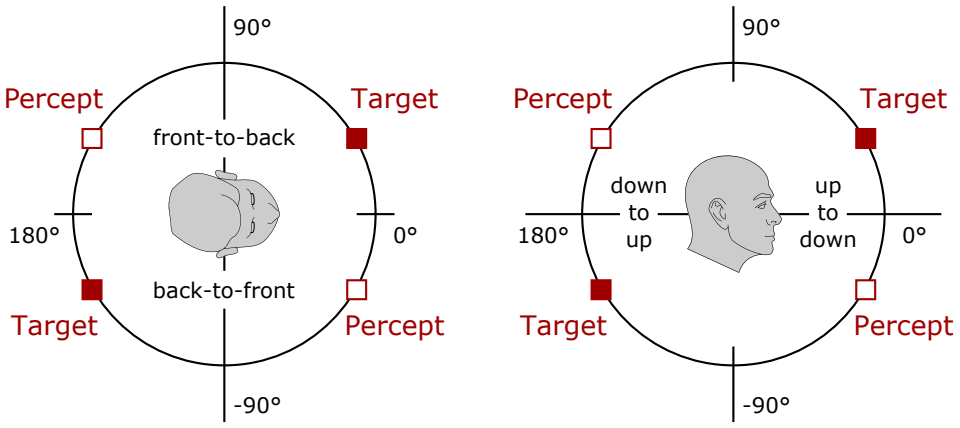


Figure 3.9: The coordinate system used for positional audio

To describe the position of a virtual audio source in the world a coordinate system has to be defined. In this thesis I will use a spherical coordinate system centered on the listener (see Figure 3.9). The position of each virtual sound source is then defined by three value: distance, azimuth and elevation.

Both the azimuth and the elevation angle describe the angular perception of the sound source. The azimuth is the angle from the front of the listener to the sound source around an axis perpendicular to the ground. When expressed in degrees, $0°$ is directly in front of the listener. The object can also be rotated up to $180°$ to either side, with $180°$ meaning the source is directly behind the listener. Since the ears are positioned at almost opposite sides of the head, human hearing is very perceptive to the azimuth of a sound source [2]. The elevation is limited to $90°$, where $90°$ upwards is directly above the listener and $90°$ downwards is directly below the listener. While the placement of the two ears on a horizontal plane helps with the localization of sound sources around the listener, up- or downward positions are more difficult to discern [2].

There are two important cues used by the human brain to localize a sound source. The first is the interaural intensity difference (IID). A sound source to one side of the listener will be heard louder on the ear directed towards the source than on the other one. The difference in intensity is dependent on the frequency of the sound, it gets smaller with lower frequencies. The reason for this is that sound waves with a longer wavelength will diffract more around obstructions, like the head of the listener. Below approximately $1\,\mathrm{kHz}$ IID is no longer an effective localization cue [2]. Using volume differences to provide 3D audio cues is a very common technique used by computer games. Even



(a) Aimuth confusion (b) Elevation confusion

Figure 3.10: Localization errors caused by ambiguity [52]

some of the very first 3D games, like Wolfenstein 3D by id Software, already used stereo-panning to simulate positional audio [72].

The second important cue is the interaural time difference (ITD). Any sound source located to either side of the listener's head will have a different distance to each ear. This means the ear located on the far side of the head as seen by the audio source is farther away and will receive the audio signal later than the other ear. This time difference between the two signals can be interpreted by the brain to help localize the source. It is most effective bellow a frequency of about 1.6 kHz, after that the wavelength of the sound waves is smaller than the width of a typical human head [2].

Even when using both IID and ITD to localize a sound source there are still some possible ambiguities. They can result in a localization error, with a common error being a front and back reversal (see Figure 3.10a). These types of error typically result in a source in the frontal hemisphere being judged by the listener to be in the rear hemisphere, although the opposite is also possible [52]. The same type of error can happen with the elevation of an audio source, where up and down can be misjudged by the listener (see Figure 3.10b). These types of errors have their origin in the so called cone of confusion. Assuming a spherical head with symmetrically located ears canals, there are several possible source locations for any IID or ITD cue. An example of this is shown in Figure 3.11. The four marked points all produce identical IID and ITD cues, possibly leading to reversals on both the front-back and on the up-down axis. Besides these marked points every point on the circle also produces the same cues. They,

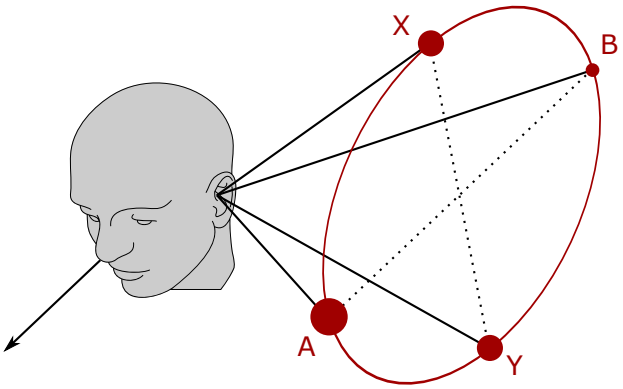


Figure 3.11: The cone of confusion [2]

together with the listener's head, form the cone of confusion. In reality, the pinnae, or outer ears, provide additional cues that are not present when using a simplified model without the outer ears. These cues can also be interpreted by the human brain, possibly resolving an otherwise existing ambiguity [2].

There are more effects that might be desirable when creating a virtual acoustic environment. One of them is reverberation, caused by the reflection of the sound waves from obstacles in the environment [2]. Another is the doppler shift which is a change in perceived pitch in the sound emitted by moving objects [2]. Since neither of these effects will be used by the application discussed in this thesis they will not be explained in detail.

3.3 Head-Related Transfer Functions

To improve the ability of a listener to localize an audio source the effect of the pinnae on the audio signal can be simulated. The necessary transformation is done by applying two convolution operations to the audio signal, one for each ear (see Figure 3.13). The function applied for each ear is called a head-related transfer function (HRTF). The actual factors used by the HRTF (named g_n in the figure) depend on the relative position of the listener to the audio source. These factors can be derived from a measured impulse response. Such a measurement can be done using a dummy head that includes modeled pinnae. This approach was for example used by the Massachusetts institute of technology (MIT) Media Lab using a Knowles Electronics mannequin for acoustics research (KEMAR) dummy head (see Figure 3.12) [10]. Another way to obtain HRTF

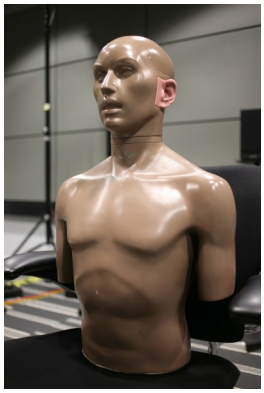


Figure 3.12: A KEMAR dummy head with pinnae [61]

measurements is by using a real human being equipped with microphones in their ear canals. This approach was for example used to create the Listen HRTF database, which contains the measured head related impulse responses of 51 test subjects [46].

Since the geometry of the outer ear varies from person to person, so does the corresponding HRTF. Ideally each user of a binaural audio system could have their own HRTF matched for their individual pinnae. But since the measurements necessary to generate an individual HRTF are complicated and require special equipment, this is not easily realized. Wenzel et al. have shown that using non-individualized HRTFs can still allow for a satisfactory ability to localize individual audio sources [71]. When using a non-individual HRTF, it should be based on data measured using a good localizer, meaning a person that can determine the position of real audio sources with a high accuracy [71].

When using the DSP system shown in Figure 3.13, the head-related impulse responses (HRIRs) applied with the FIR filters must use the same sampling rate as the audio data it is applied to. Otherwise either the audio data or the HRIR has to be resampled. Since the HRTF data is based on measurements, it is only available for the points (defined by azimuth and elevation) where a measurement was taken. If the relative position of an audio source to the listener's head does not exactly match one of these points, the HRTF data has to be interpolated by using the available measurement points surrounding the actual source position.

Since the user can change the orientation of the virtual listener, the latency of the audio system is also important. Experiments with head-tracking audio systems suggest that a latency of 60 milliseconds or below is undetectable to almost all listeners when no

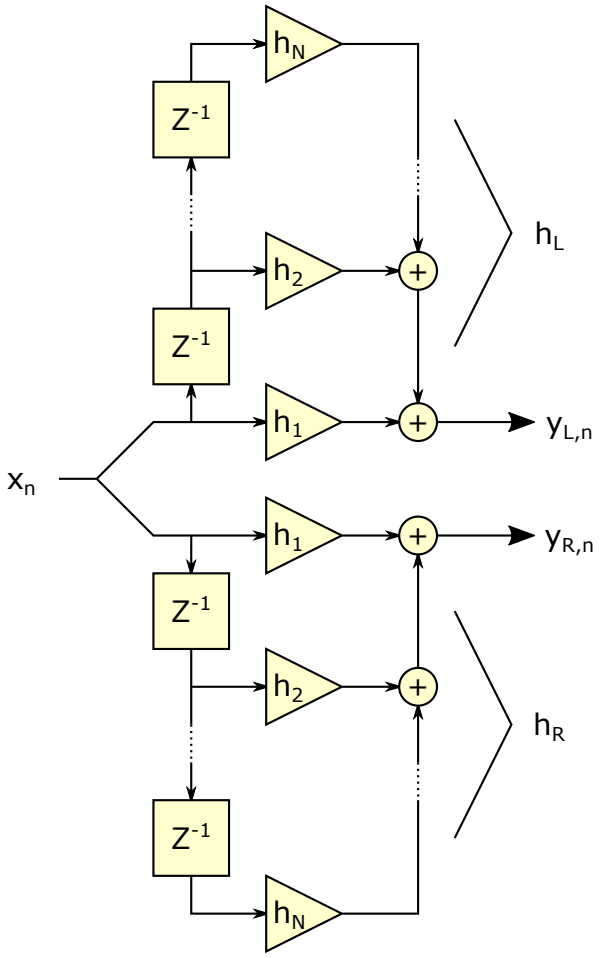


Figure 3.13: Convolution using a separate FIR filter for each ear [2]

low-latency reference tone is present [7]. Since this application requires the user to wear headphones that would block out such an external reference tone this lower bound is applicable to the game described in this thesis.

4

Architecture

In this chapter the overall architecture of the project is described. This includes an explanation of the different modules as well as the build system used to compile the project. The external depencendies used by the application are also described.

4.1 Project Structure

The project is split into several modules, as shown in Figure 4.1. Modules shown in blue include native code. Also shown are the dependencies of the modules to each other. A dashed arrow shows a dependency for the tests only.

The Java code is also organized into different namespaces. A module usually contains all classes of one namespace. Exceptions are the audio modules, which all add to the same namespace and the engine module which uses sub-namespaces to further subdivide the code. The different namespaces used in this project and their dependencies are shown in Figure 4.2

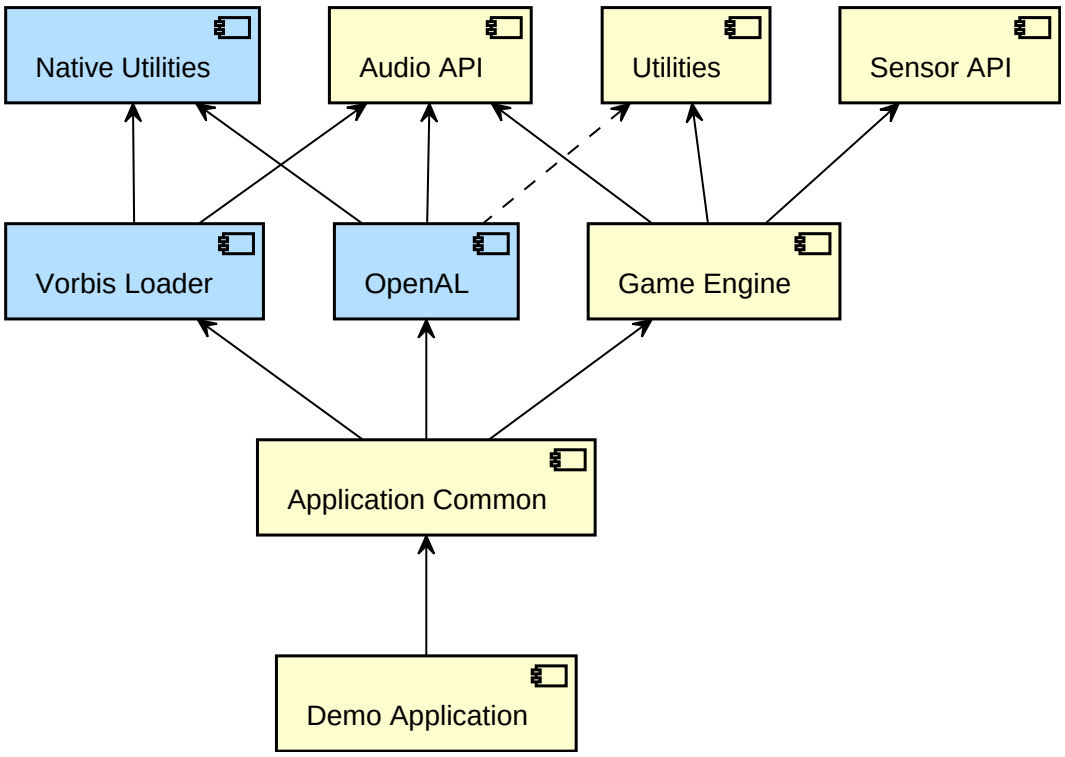


Figure 4.1: Project module dependencies

The **utilities** module contains static utility classes and functions. Currently it only contains functions to convert and normalize angles in degrees or radians. It is implemented in the `utilities` namespace and described in detail in section 5.1.

The **native utilities** module is completely implemented in C++ without any Java code. It contains classes and functions that facilitate the implementation of code using the Java native interface (JNI). Like the Java utility module it is described in section 5.1.

The **sensor API** module defines a simple interface to access a rotation sensor on an Android device. It also includes adapters for some of the potentially available Android sensors to this interface. Both the interface and its implementations are defined in the `sensors` namespace. The module is described in detail in section 5.3.

The **audio API** module defines an implementation-independent API to access a positional audio system. It also contains an audio data class, a WAVE file reader, audio filter implementations and support classes to help with an implementation of the API. Everything in the module is contained in the `audio` namespace. A detailed description can be found in section 5.2.

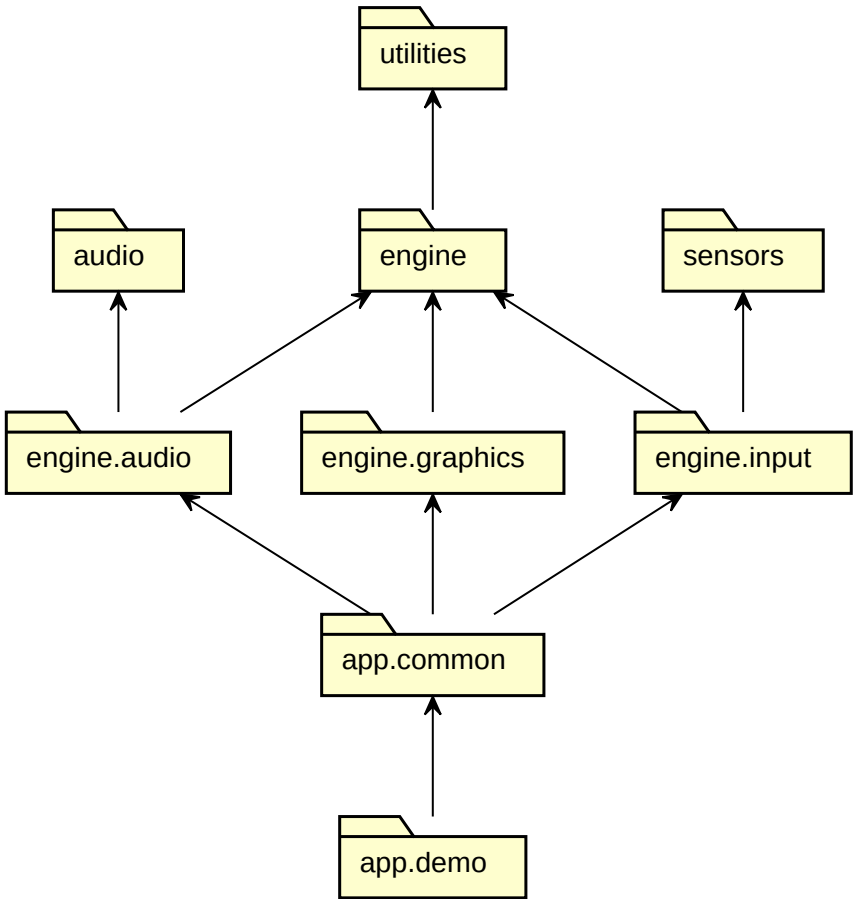


Figure 4.2: Java package dependencies

The **Vorbis loader** module extends the audio API by adding an Ogg-Vorbis file reader to the `audio` namespace. Since the file reader is based on a native library, the module uses JNI and the native utilities module. The file reader class is described in subsection 5.2.5 together with the other audio API utilities.

The **OpenAL** module provides an implementation of the audio API. Since OpenAL is used as a native library this module also uses JNI and the native utilities module. In addition, the included tests depend on the Java utilities module. The implementation is also located in the `audio` namespace. An explanaiton of the functionality provided by this module as well as more details about the OpenAL library can be found in chapter 6.

The **game engine** module defines the framework used in the application. It is implemented in the `engine` namespace, which contains three sub-namespaces: The audio

part of the game engine is implemented in the `engine.audio` namespace and depends on the audio API. The graphics part uses the `engine.graphics` namespace. Finally, the input part is contained in the `engine.input` namespace. It makes use of the sensor API. More information about all parts of the game engine can be found in chapter 7.

The **demo application** module contains the actual Android application that is used to demonstrate the game concept. It is implemented in the `app.demo` namespace. Classes that might be of interest to other applications based on the game engine or the other framework modules are split into the **common** module, using the `app.common` namespace. The application is described in chapter 8.

4.2 Build System

The project uses two different build systems. For the Java part of the project, Google's Android build system, which is implemented as a plugin for Gradle, is used. For the native code CMake is used. Both build systems will be described in this section.

4.2.1 Gradle

The Java part of the project is build using Gradle. Gradle is a build automation tool that uses the Java-based Groovy programming language to configure the build process [40]. It is extensible using plugins. The Android build tools used for this project are one example of such a plugin. They are included in the Android software development kit (SDK) [15]. The Android Studio integrated development environment (IDE) can directly import and build a Gradle project using this plugin. When not using Android Studio, the build process can be started using the Gradle wrapper included with the source code. This is a batch or shell script, that can automatically download Gradle and run the build process [41]. This way no local Gradle installation is necessary.

The project is divided into the root project and several sub-projects. The configuration for the root project is done in the `build.gradle` file located in the root directory of the source tree. Here the used version of the Android build tools as well as the repository from which it is downloaded are specified. The repository used by the sub-projects to download any required Maven packages is also defined. In addition, the

`ExtraPropertiesExtension` plugin is used to define variables that are available to all sub-projects (see Listing 4.1). Here the target Java, Android SDK and build tools versions are set. In addition, a list of all supported application binary interfaces (ABIs) and their priorities are defined. Also in the root directory is the `settings.gradle` file. In this file all sub-projects are added to the main Gradle project.

```
1 minSdkVersion = 10
2 targetSdkVersion = 23
3 compileSdkVersion = 23
4 buildToolsVersion = '23.0.1'
5 sourceCompatibility = JavaVersion.VERSION_1_7
6 targetCompatibility = JavaVersion.VERSION_1_7
7 supportedAbis = ['armeabi-v7a', 'arm64-v8a', 'x86', 'x86_64']
8 abiVersionCodes = \
9     ['armeabi-v7a': 1, 'arm64-v8a': 2, 'x86': 3, 'x86_64': 4]
```

Listing 4.1: Gradle project properties

Each sub-project also has its own `build.gradle` file, located in the root directory of the module. In this file the necessary plugins for the sub-project are applied. The `idea` plugin is used to name the module for display purposes in Android Studio. The sub-projects also define their dependencies in this file. Both external as well as local dependencies are supported. Not shown in Figure 4.1 are the libraries sub-projects. These Gradle projects contain no Java code and are only used to build and manage the external native libraries. This is done using a custom plugin, which will be described in the next chapter. Any Java dependencies are managed and if necessary downloaded by the Gradle build system.

Each sub-project either uses the Android application or the Android library plugin. For either of these plugins, the build options are set by using the global extra properties set in the root project. For the application project some additional configuration is required: For versioning purposes both a version number as well as a name are set. The version number is used to handle automatic upgrades, the version name is only used when displaying the version to the user [39].

To reduce the overall file size of the final Android application package (APK), ABI splitting can be enabled. If this option is set, instead of packing all versions of the native libraries into the same APK file, a different file is generated for each supported CPU architecture. Only the compiled libraries for this ABI are then packed into the associated APK file. Since this project is based on several native libraries, using this option reduces the final

size of the application file. Using the ABI splitting option requires some additional work when versioning. If a device supports more than one ABI it is important that the APK using the preferred ABI has a higher version number. Otherwise it could for example happen that an ARMv8 based device would use its backwards compatibility to run an ARMv7 version of the application instead of using the optimized ARMv8 version. To prevent this the million digit of the version number is set to a different code for each supported API. More modern or advanced ABIs get higher numbers so devices that support them use the optimal version for their hardware. For example, 64-bit ABIs get higher numbers then their 32-bit counterparts. The ABI also gets appended to the version name in a readable format.

The Android build system includes ProGuard, a program that shrinks, optimizes and obfuscates Java code [50]. For this project, ProGuard is used to further reduce the size of the final APK file. This is done by removing unused code from the external Java libraries used by this project. ProGuard is only enabled for release builds. Configuration is done in the `proguard.pro` file located in the root of the application sub-project. Besides the necessary configuration for external libraries this file also disables the obfuscation feature. This is done to help with debugging of the release version since any stack trace generated by the obfuscated version would itself also be obfuscated [33]. If enabled, the obfuscation of the stack trace can be reversed using the `retrace` script included in the Android SDK.

4.2.2 CMake

To compile native C or C++ code for Android applications, the Android native development kit (NDK) is used [13]. It includes the necessary compilers and libraries to build native code for the different Android platforms. Different versions of both the GNU compiler collection (GCC) as well as the Clang compiler are included. Native Android libraries can interact with Java code by using JNI. The Android NDK includes a version of GNU Make that can be used to build native libraries and applications.

While Android Studio allows the compilation of NDK projects, this capability is deprecated. In May 2015 Google announced that a new system for building NDK projects will be released with version 1.3 of Android Studio [42]. This new version will include syntax highlighting for C and C++ code as well as debugging support for native code. During the development of the application described in this thesis this new build system

was not yet available. While a preview build of Android Studio 1.3 was released in July 2015 [14], including support for the new NDK features, the necessary Gradle plugin was still experimental. Since the API and by extension the domain-specific language (DSL) of this experimental plugin are likely to change [22], this new system to build native libraries is not yet used in this project.

Instead the cross-platform build system CMake is used [48]. This software allows the usage of platform and compiler independent configuration files to control the build process. These configuration files are processed by the CMake application and translated into makefiles for a chosen target compiler and platform. Cross-compilation is supported and the same configuration file can be used to generate makefiles for different target architectures. To build native libraries for the Android platform and its many ABIs, the `android-cmake` build scripts are used. These scripts were originally developed to compile OpenCV for Android by allowing the creation of makefiles that use the Android NDK. Using CMake as native build system also allows for an easy integration of other project that include CMake configration files. Once such project is the OpenAL library that is used by this project.

To facilitate the parallel development of the Java and the native code portions of this project, the CMake build process is integrated into the Gradle build. This is done by defining a plugin that adds new build tasks to the Gradle build process. The first task runs when a build of the Android application is started and generates a CMake configuration file. The contents of this file are defined in the Gradle project configuration file. It also includes directives to build all submodules that apply the CMake plugin. This CMake configuration is then used by another task to generate a makefile that builds all the native libraries required by the application. This task is defined and run several times, once for each supported ABI. Finally, tasks are added that run these makefiles and compile the actual libraries. These tasks are also defined for each ABI separately. All tasks defined by the CMake build plugin are added as dependencies to the regular build process. This makes sure that all native libraries are up-to-date when the Android application is build using Gradle.

4.3 External Dependencies

In this section the external or third-party libraries used by the application described in this theses are introduced. They are split between Java and native dependencies.

4.3.1 Java Dependencies

The project uses several third-party Java libraries. One of them is an implementation of the Java annotations introduced in Java specification request (JSR) 305 [57]. While these annotations don't have any effect by themselves, a compatible IDE can use the additional information provided by using the annotations to find bugs that are not detected by the Java compiler itself. Since it is based on IntelliJ IDEA, the Android Studio IDE [16] can for example use the `@Nonnull` and the `@Nullable` annotations to detect incorrect handling of object references that might be `null`. While the Android support annotations also include nullness annotations [27], I decided to use the JSR 305 annotations because they provide a way of setting a default for method parameters. This allows the annotations to be omitted in a lot of cases without loosing their benefits, reducing the clutter in the method signatures. Such a default value can be specified per Java package. The JSR 305 annotations are used in every Java module of this project.

The game engine module also depends on the Guava Java library developed by Google [25]. This library is mainly used for the collections it adds, including immutable implementations of many basic Java collection classes. The cache implementation as well as the precondition checking utilities defined by Guava are also utilized by the game engine. It also uses the JSR 305 annotations, extending the nullness checking done by the IDE to calls into the library. Since dependencies are transitive, any module depending on the game engine also has access to the full Guava library. To reduce the file size of the final APK, ProGuard is used to remove unused features of the library during the build process.

To facilitate the implementation of backwards compatible Android applications, Google provides several support libraries [38]. The application modules depend on both the v4 "support" as well as the v7 "appcompat" library. The support library can be used with Android API level 4 and above. It includes a backwards-compatible implementation of fragments. These are encapsulations of layouts that can be reused in different activities and were added in API level 11. Since the application targets API level 10, usage of the support library is required to make use of fragments. The appcompat library depends itself on the support library. It can be used with Android API level 7 and above. It provides activity classes that use the action bar user interface [11]. These activities are compatible with fragments designed using the support library. The appcompat library was also updated to enable the creation of user interfaces using the material design

theme introduced with Android 5.0 [29]. This allows designing an application using the modern material theme while still maintaining backwards-compatibility.

4.3.2 Native Dependencies

The external Java libraries are automatically downloaded by Gradle, using Maven package repositories. For the native libraries, this is not an option. Instead, their source code is included in the project in the `libraries` folder. The code is included as git subtrees [60]. This way the native dependencies are available with the project and no additional download of any libraries is required to build the application.

The first external dependency is the `android-cmake` project. It provides scripts that facilitate using CMake to build native Android libraries. This is done by configuring the toolchain used by CMake to point to the compilers and libraries included with the Android NDK for the chosen ABI. Compilation for different ABIs is done by calling CMake multiple times. Since this project uses CMake exclusively to build any native code, this script is essential to building any modules using native code.

To reduce the size of the audio files used by the application, they are encode using the Vorbis audio codec and stored using the Ogg container format [74]. Since Ogg-Vorbis files use a lossy compression, the resulting files can be much smaller than files storing the raw PCM data, like for example WAVE files. Unlike the popular MPEG-2 audio layer III (MP3) codec, Vorbis is patent and royalty free [75]. To decode the Ogg-Vorbis audio files, the `libvorbis` library is used. It is released under a BSD-like license. To parse the Ogg container containing the Vorbis encoded audio data, it uses the `libogg` library. This library is released under the same BSD-like license. Both libraries are written in the C programming language. In this project `libogg` is statically linked into the `libvorbis` library, which is compiled as a shared library. Since neither the Ogg nor the Vorbis library include a CMake build script, one is supplied as part of this project. Also included is the necessary configuration header file with settings appropriate for an Android systems as well as a Gradle build file that makes the resulting library available as Android library module. This library is then accessed by the Vorbis loader module using JNI and used to decode the Ogg-Vorbis files included with the application.

To create positional audio, the OpenAL Soft library is used. This is a software based implementation of the OpenAL API, written in C. It supports multiple platforms, including

Android, and is licensed under the GNU lesser general public license (LGPL). More details about this library can be found in chapter 6.

4.4 Documentation

The modules of this project are documented in two different ways. One is in the form of this thesis, where the content of the different modules will be described in the following chapters. This is only meant to give an overview over the existing interfaces and functionality. To provide more detailed information about classes, methods and fields, the JavaDoc style of commenting is used [55]. These comments can be converted to HTML pages and are understood by many IDEs, including Android Studio. To save space, these comments are not reproduced in the source code listings used throughout this thesis. The annotations introduced with JSR 305 are also used to document the code. The `@Nullable` and `@Nonnull` annotations are used with every method to define if the return value can be `null`. All packages in this project use the `@ParametersAreNonnullByDefault` annotation to mark all parameters as not accepting `null` by default. Where appropriate, the `@Nullable` annotation is used to override this default. The `@Immutable` annotation, also imported from JSR 305, is used to explicitly document immutability in classes.

Both in Java as well as in the native code methods are only documented where they are first declared. This means that for the native code, most documentation is done in the header files. Here the Doxygen style of commenting is used, which is very similar to the JavaDoc style [43]. JNI methods, that is methods that are declared in Java using the `native` keyword and implemented in native code, are documented in the Java file using the JavaDoc syntax. This documentation is not repeated in the native file. Java fields and methods are documented in the interface or class where they are first declared. Methods that override another method are not documented again, here the description provided by the parent class or interface is inherited.

5

Framework

Several parts of the application are implemented in library modules. Each of these modules provides an API to access their functionality. This allows them to be reused easily. In this chapter, the utility library, the audio framework as well as the rotation sensor module used by the application will be described.

5.1 Utility Libraries

The project contains two utility library modules. Both contain classes and functions used by the other modules of the project. One module is implemented purely in Java, the other contains only native code. They can be used independently of each other.

The Java module only contains one class, called `Angle` (see Listing 5.1). This class contains static utility methods to handle angles expressed as floating point numbers of the type `float`. It defines both π (as `PI`) and $\tau = 2\pi$ (as `TAU`) as well as conversion functions between radians and degrees. They make use of their respective Java

standard library versions and exist only to reduce the amounts of casts necessary while writing code using `float`-based angles. It also defines functions to normalize angles to the range $[-180°, 180°)$, $[0°, 360°)$ or $[-\pi, \pi)$ for angles in degrees and radians respectively. The implementation of the normalization functions is based on the `normalizeAngle` function from the Apache commons `MathUtils` class. Since the class only defines static methods, its constructor is private [3].

```
1  public final class Angle {
2      public static final float PI;
3      public static final float TAU;
4      public static float toRadians(float degrees);
5      public static float toDegrees(float radians);
6      public static float normalizeRadians(float angle);
7      public static float normalizeDegrees(float angle);
8      public static float normalizeDegreesPositive(
9                                              float angle);
10 }
```

Listing 5.1: `Angle` class

The native utility library module contains only native code implemented using C++. While some functions and classes are header-only, most of the code is compiled into a shared library called `utilities-jni`. This file has to be loaded by the Java virtual machine (JVM) before any other native libraries that depend on the utility library can be used.

The library defines two header-only template classes: `vector` and `scope_guard`. The `vector` class represents a resizable block of memory in the application heap. It is implemented using the C standard library memory functions. Memory allocated by this class is not cleared before being made available to the user. The `scope_guard` class can be used to create an object that calls a user-supplied function when the current scope is left. It supports the lambda-functions introduced in C++11 [73]. This functionality can be used in a similar fashion as a finally-block may be used in Java, for example to release a previously acquired lock when a function returns.

Also included are some classes to facility interactions with Java classes and class members using JNI. The class called `java_class` finds and holds a reference to a Java class given its fully qualified name. It also optionally registers native method implementations with their Java counterparts for the represented Java class. The library also contains classes to access methods and fields of Java classes. Both static and

instance methods and fields are supported. These classes are specializations of the template class `java_class_member`. The field and method ids are found using a `java_class` instance, the name and the signature of the class members. All of these classes are implemented using the JNI functions of the JVM [56].

The library also contains some standalone functions to facilitate JNI programming further. The function `java_long_from_ptr` converts a pointer to a value that is safely storable in Java `long` field. This pointer can later be retrieved using the `java_long_to_ptr` function. It is the responsibility of the user to specify the correct type when retrieving the pointer. Using a `long` field guarantees that the pointer is safely storable even on a 64-bit architecture. The function `get_thread_java_env` allows getting the Java environment given a pointer to the Java virtual machine object. If this function is called from a thread not yet attached to the JVM this will attach the thread using the JNI Invocation API [58]. A POSIX thread-specific data key [70] is used to detach the thread once it shuts down.

Finally, the library also defines two functions that translate between POSIX file pointers and Java streams. The first is called `fopen_input_stream` and opens a Java `InputStream` as a read-only file pointer. The other is called `fopen_output_stream` and it opens a Java `OutputStream` as a write-only file pointer. Both are implemented using the JNI utility classes described in the previous paragraphs. To create a file pointer that uses custom `fread`, `fwrite` and `fclose` implementations the function `funopen` is used. This function is not part of the POSIX standard, it is instead an extension introduced in the Berkeley software distribution (BSD) C standard library, which bionic, the Android C standard library, is based on [5].

5.2 Audio API

The audio API module defines interfaces that provide an implementation-independent way to access the positional audio functionality needed by the application. It also contains some other audio-related classes that either implement parts of the API or that can be used in conjunction with it. The audio API module does not depend on any native libraries, it is purely written in Java.

Since the API is not meant to be a general purpose audio framework, it only supports single channel, or mono, audio data. The data has to be single channel since the

positional audio API assumes each source only emits sound from one position. When using stereo audio data there would be two sources of sound that have to be positioned. This data also has to be statically available, streaming of audio data is currently not supported. The mixing process has to be fast enough to keep up with the playback and by preparing the complete audio data buffer in advance the necessary performance for real-time audio mixing is reduced.

5.2.1 Basic Classes

To store the single channel audio data the module contains the `AudioData` class (see Listing 5.2). This class stores audio samples as 16-bit signed integers of the Java type `short` as well as the sample rate of the audio data. It is also immutable [3]. To enforce this both the constructors as well as the `getSampleArray` method make a defensive copy of the data array before storing or returning it [3]. The `ShortBuffer` instance returned by `getSampleBuffer` is a read-only buffer, so it can wrap the internal data array directly without violating the immutability guarantee.

```
1  @Immutable
2  public final class AudioData {
3      public AudioData(short[] samples,
4                       int sampleRate);
5      public AudioData(ShortBuffer samplesBuffer,
6                       int sampleRate);
7      @Nonnull
8      public ShortBuffer getSampleBuffer();
9      @Nonnull
10     public short[] getSampleArray();
11     public int getSampleRate();
12     public int getSampleCount();
13 }
```

Listing 5.2: `AudioData` class

An implementation of the audio API might be using resources that should be released. For example, if the API is implemented using native code, there might be memory that has to be freed when a resource is no longer in use. Such a class usually implements the `AutoCloseable` interface, which defines a method called `close` that relinquishes any resources held by the implementing object [17]. This interface would also allow the usage of the try-with-resource syntax [59]. On Android, this interface was added in API

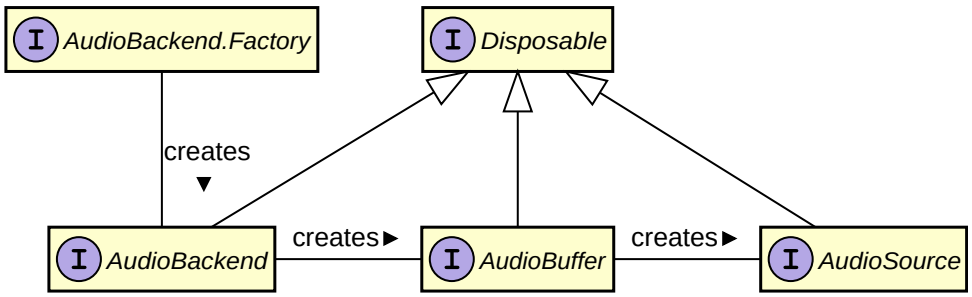


Figure 5.1: Audio API interfaces

version 19 [17]. Since the audio API is designed to be backwards-compatible with API version 10, using this interface is not an option. While the `Closeable` interface exists since version 1 of the Android API, it is defined to throw an `IOException`, which is a checked exception [18]. Releasing the resources that may be in use by an audio API implementation should either be unable to fail or throw a `RuntimeException`, since a failure to release the resources might leave the audio API implementation in a non-recoverable state. Because of these reasons the audio API defines the `Disposable` interface (see Listing 5.3). It serves the same function as the `AutoCloseable` interface, but is available on all supported Android API levels.

```
1 interface Disposable {
2     void dispose();
3 }
```

Listing 5.3: `Disposable` interface

5.2.2 Positional Audio API

The main part of the Positional Audio API are the interfaces that provide a generic method of accessing a positional audio framework. A realization of the API has to implement the `AudioBackend`, the `AudioBuffer` as well as the `AudioSource` interface. The relationship between these interfaces can be seen in Figure 5.1. Since each of them may hold resources that need to be released, they all extend `Disposable`.

Before actually creating a positional audio source, first the chosen audio back-end has to be initialized. An audio back-end is represented by the `AudioBackend` interface (see Listing 5.4). To create an instance of this class, a factory object should be used. This

```
1  public interface AudioBackend extends Disposable {
2      @Nonnull
3      AudioBuffer createBuffer(AudioData audioData);
4      void pauseAll();
5      void resumeAll();
6      float getOrientation();
7      void setOrientation(float azimuth);
8      float getGain();
9      void setGain(float gain);
10 }
```

Listing 5.4: `AudioBackend` interface

object should implement the `Factory` subinterface of the `AudioBackend` interface (see Listing 5.5). This interface defines a factory method to create an audio back-end instance [9]. Having a common factory interface is especially useful when using dependency injection. It allows an application or library to create an audio back-end without having to know its actual type. The `create` method takes a `Context` object as its only argument to allow the back-end to access Android system services or load assets [19]. If a back-end is not supported on a device the creation may fail by returning `null` from the factory method. The `getBackendName` method returns the name of the back-end created by this factory, it can for example be displayed when allowing the user to choose an implementation.

```
1  public interface AudioBackend extends Disposable {
2      interface Factory {
3          @Nonnull
4          String getBackendName();
5          @Nullable
6          AudioBackend create(Context context);
7      }
8  }
```

Listing 5.5: `AudioBackend.Factory` interface

Once created the `AudioBackend` provides methods to get or set the listener orientation and the gain factor of the output. The orientation is given as a counter-clockwise rotation in radians. It also can pause and later resume all audio sources currently managed by the back-end. Finally, the `createBuffer` factory method can be used to create an audio buffer filled with the audio data provided as parameter.

The created buffer implements the `AudioBuffer` interface (see Listing 5.6). It defines only one method: `createSource`. This factory method is used to create an audio source backed by this buffer. Multiple audio sources can be backed by the same buffer.

```
1 public interface AudioBuffer extends Disposable {
2     @Nonnull
3     AudioSource createSource();
4 }
```

Listing 5.6: `AudioBuffer` interface

The created audio sources implement the `AudioSource` interface (see Listing 5.7). This interface defines methods to play or pause playback as well as a method to query if the source is currently playing. Stopping, that is pausing and resettings the playback position, is also supported. The source can be set to loop, it will then not stop playing when it reaches the end of the audio data buffer but instead restart playback from the beginning. The playback position can also be queried or set manually. Each source has its own gain factor that is applied in addition to the gain factor set by the back-end. Finally, since this is a positional audio API, the position of the source can be set. It is set in polar coordinates as an angle and a distance. As with the listener, the angle is given as a counter-clockwise rotation in radians. The distance has no predefined unit, but if the distance is set to one unit or less, the amplitude gain factor of the source will

```
1  public interface AudioSource extends Disposable {
2      void play();
3      void pause();
4      void stop();
5      boolean isPlaying();
6      float getPositionAzimuth();
7      float getPositionDistance();
8      void setPosition(float azimuth, float distance);
9      boolean isLooping();
10     void setLooping(boolean looping);
11     float getGain();
12     void setGain(float gain);
13     int getPlaybackPosition();
14     void setPlaybackPosition(int position);
15 }
```

Listing 5.7: `AudioSource` interface

not be lowered because of the distance. If the distance is more than unit, the amplitude gain factor is calculated as the inverse of the distance (see Equation 5.1) [47]. The distance amplitude gain factor is applied in addition to the source amplitude gain factor. If the distance of a source is set to zero it is an ambient source. The sound of such a source does not change with the orientation of the listener and is equally audible on all output channels.

$$gain_{distance} = \begin{cases} 1 & \text{if } |distance| \leq 1 \\ |distance|^{-1} & \text{if } |distance| > 1 \end{cases} \tag{5.1}$$

5.2.3 Abstract Helper Classes

The audio API module also contains some abstract helper classes that assist with implementing the positional audio back-end interfaces. They are based on the abstract `ObservableDisposable` class (see Listing 5.8). This class implements the Observer pattern [9]. Observers of this class must implement the `Observer` subinterface. They can then be registered using the `addObserver` method and removed using the `removeObserver` method. The class keeps a strong reference to each registered observer. The generic type parameter `O` specifies the type of the class that extends this abstract class. Making the class and by extension the `Observer` interface generic allows for the `onDispose` method to receive a reference to the disposed object using its actual type without needing any casts. The `notifyDispose` method should be called by the implementing class to notify any registered observers when the object is is being disposed.

```
1  abstract class ObservableDisposable
2          <O extends ObservableDisposable<O>>
3          implements Disposable {
4      interface Observer<O extends ObservableDisposable<O>> {
5          void onDispose(O disposable);
6      }
7      final void addObserver(Observer<O> observer);
8      final void removeObserver(Observer<O> observer);
9      final void notifyDispose(O observable);
10 }
```

Listing 5.8: `ObservableDisposable` class

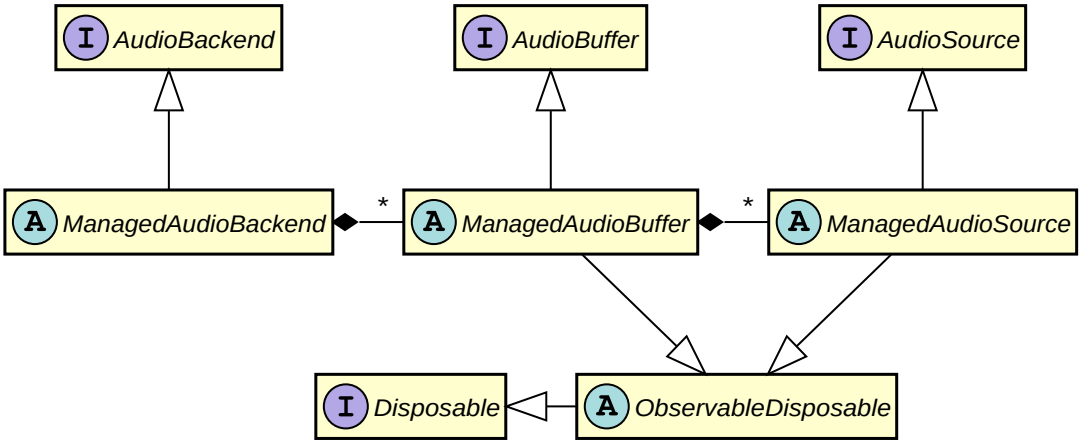


Figure 5.2: Managed Audio API classes

The actual abstract helper classes are called `ManagedAudioBackend`, `ManagedAudioBuffer` and `ManagedAudioSource`. Their relationship to each other can be seen in Figure 5.2. All three classes have to be used in conjunction with each other, meaning a back-end implementation that extends `ManagedAudioBackend` has to create buffers of the type `ManagedAudioBuffer` that in turn create sources of the type `ManagedAudioSource`. All helper classes are implemented in a thread-safe way.

The abstract `ManagedAudioBackend` class implements the `AudioBackend` interface (see Listing 5.9). It keeps a list of all `ManagedAudioBuffer` instances created by its extending class. For this to work, each time the superclass creates a new buffer, it should also call the `registerBuffer` method. This method adds the buffer to the internal list kept by the helper class, this way a strong reference to the buffer is kept. The method also registers the internal observer instance of the helper class with this buffer. When the buffer is disposed the observer is called and removes the disposed buffer from the list, allowing it to be garbage collected once it is no longer reachable by other references. Adding the `ManagedAudioBackend` as observer guarantees that the buffer always has a strong reference to the back-end. This prevents the back-end from being garbage collected while a buffer created by it is still reachable. When the back-end itself is disposed, it should call the `disposeAllBuffers` method. This will dispose any remaining buffers. The class also implements the `pauseAll` and `resumeAll` method defined in the `AudioBackend` interface. Their implementations will iterate through all buffers created by this back-end and call their respective `pauseAll` and `resumeAll` methods.

```
abstract class ManagedAudioBackend
        implements AudioBackend {
    final void registerBuffer(ManagedAudioBuffer buffer);
    final void disposeAllBuffers();
    @Override
    public final void pauseAll();
    @Override
    public final void resumeAll();
}
```

Listing 5.9: `ManagedAudioBackend` class

The audio buffers created by a `ManagedAudioBackend` have to extend the abstract `ManagedAudioBuffer` class (see Listing 5.10). To be observable by the audio back-end this class extends the abstract `ObservableDisposable` class. It also extends the `AudioBuffer` interface, although it doesn't implement any of its methods. When the super-class that extends this helper class creates an audio source, it should be registered using the `registerSource` method. When disposed, the `disposeAll-Sources` method will dispose any sources created and backed by this buffer. A two-way strong reference will be maintained between this buffer and each created source until it or the buffer are disposed. Since the buffer also keeps the back-end reachable, the back-end can not be garbage collected while an active buffer or source exist. If the back-end is disposed, all buffers and by extension all sources are disposed. The `pauseAll` method is used by the back-end to pause all sources created by this buffer. The paused sources are kept in a list and can later be resumed by the back-end using the `resumeAll` method. If a paused source is disposed it is also removed from the paused sources list.

```
abstract class ManagedAudioBuffer
        extends ObservableDisposable<ManagedAudioBuffer>
        implements AudioBuffer {
    final void registerSource(ManagedAudioSource source);
    final void disposeAllSources();
    final void pauseAll();
    final void resumeAll();
}
```

Listing 5.10: `ManagedAudioBuffer` class

The abstract `ManagedAudioSource` class does not define any additional methods (see Listing 5.11). Like the buffer variant, it extends the `ObservableDisposable` class. It also extends the `AudioSource` interface. The declaration of this class is necessary to implement the `ManagedAudioBuffer` class in a type-safe way without any additional type parameters.

```
1 abstract class ManagedAudioSource
2         extends ObservableDisposable<ManagedAudioSource>
3         implements AudioSource { }
```

Listing 5.11: `ManagedAudioSource` class

5.2.4 Audio Filters

The audio API also provides a generic filtering interface. Because filtering audio data may be a potentially expensive operation, it is done once before the audio data is written to a buffer. This allows the filters to be implemented independently of the chosen audio back-end implementation. Most back-end implementations would probably chose to use native code for the audio mixer implementation, including the OpenAL back-end implemented as part of this Project. Applying the filter to the audio data before writing it to the buffer allows the filter code to be implemented purely in Java.

Each audio filter should implement the `AudioFilter` interface (see Listing 5.12). This interface only defines one method which takes an `AudioData` object, applies the filter to the audio data, and returns a new `AudioData` instance containing the filtered audio data. This method is not allowed to fail, it must always return a valid `AudioData` object. Since it is immutable the audio data can not be filtered directly. Instead a copy has to be made. All audio filters should also be implemented so that their instances are immutable. They also have to be able to handle different sampling rates, since they may be used multiple times on audio data with different sampling rates. The interface

```
1 public interface AudioFilter extends Parcelable {
2     @Nonnull
3     AudioData filter(AudioData input);
4 }
```

Listing 5.12: `AudioFilter` interface

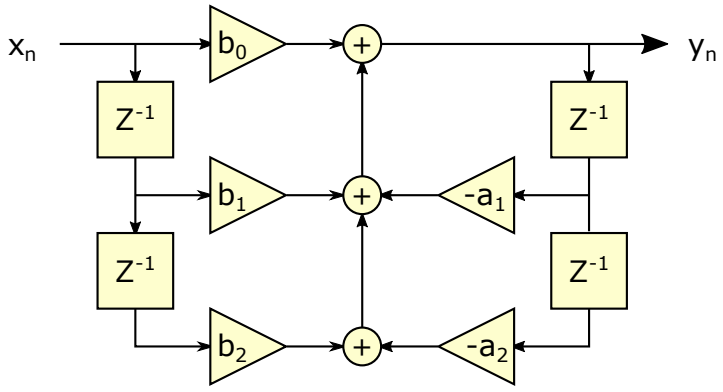


Figure 5.3: A biquad filter in direct form 1 [68]

extends the `Parcelable` interface. This allows filter instances to be stored in and later restored from a `Parcel` object without requiring the user to save the filter parameters manually.

Given this simple interface it is easy to apply several audio filters sequentially. The `AudioFilterChain` helper class can be used to define such a chain (see Listing 5.13). It also implements the `AudioFilter` interface itself, allowing it to be used anywhere a single audio filter might be used. It is initialized with an array of audio filters. When the `filter` method is called, these filters are applied to the input data in the order they where given during construction. Once all filters have been applied the resulting audio data is returned.

```
1 @Immutable
2 public final class AudioFilterChain implements AudioFilter {
3     public AudioFilterChain(AudioFilter... filters);
4     @Nonnull @Override
5     public AudioData filter(AudioData input);
6     /* Parcelable implementation omitted */
7 }
```

Listing 5.13: `AudioFilterChain` class

The audio API also includes three predefined filters. They are a low-pass filter, a high-pass filter and a notch, or narrow band-stop, filter. They are implemented as second-order recursive linear filters using the direct form 1 (see Equation 5.2 and Figure 5.3) [68]. The parameters for the filters are taken from Robert Bristow-Johnson's Filter Cookbook [6].

$$y[n] = \frac{b_0}{a_0}x[n] + \frac{b_1}{a_0}x[n-1] + \frac{b_2}{a_0}x[n-2] - \frac{a_1}{a_0}y[n-1] - \frac{a_2}{a_0}y[n-2] \tag{5.2}$$

These filters are defined in the `BiquadAudioFilters` class (see Listing 5.14). The class is implemented using the abstract factory pattern [9]. Each filter factory method accepts two parameters: a frequency and a Q factor. The frequency in hertz is the cutoff frequency for the low-pass and the high-pass filter or the center frequency for the notch filter. The Q factor is the quality factor of the filter [68]. Each static factory method instantiates an instance of the class using the private constructor. This constructor expects the type of the filter as defined in the `FilterType` enum as parameter. Each value of this enum contains an implementation of the private `FilterImpl` subinterface. This encapsulates the actual filter algorithm by using the strategy pattern [9]. The

```
1  public final class BiquadAudioFilters {
2      @Nonnull
3      public static AudioFilter lowPass(double frequency,
4                                        double q);
5      @Nonnull
6      public static AudioFilter highPass(double frequency,
7                                         double q);
8      @Nonnull
9      public static AudioFilter notch(double frequency,
10                                     double q);
11     @Nonnull
12     private static AudioData filter(AudioData input,
13                                     double a0, ...);
14     private interface FilterImpl {
15         @Nonnull
16         AudioData filter(AudioData input,
17                          double frequency, double q);
18     }
19     private enum FilterType {
20         LOW_PASS(...), HIGH_PASS(...), NOTCH(...);
21         FilterType(FilterImpl filterImpl);
22     }
23     private BiquadAudioFilter(FilterType filterType,
24                               double frequency, double q);
25     @Nonnull @Override
26     public AudioData filter(AudioData input);
27     /* Parcelable implementation omitted */
28 }
```

Listing 5.14: `BiquadAudioFilters` class

actual instance of the `BiquadAudioFilters` class only has to store the filter type enum as well as the chosen frequency and Q factor. This simplifies implementing the required `Parcelable` interface and avoids the code duplication that would be the result of implementing each filter type in its own class. Each implementation of the `FilterImpl` interface is responsible for creating the actual audio filter given the audio data to filter as well as the frequency and the Q factor. This is done by calculating the filter parameters a_0 to a_2 and b_0 to b_2. These have to be calculated for each audio data instance separately since they are not only depend on the frequency and Q factor but also on the sample rate of the filtered audio data [6]. All filter implementations then call the static `filter` method that does the actual filtering.

5.2.5 Audio Utilities

The audio API module also contains several utility classes. The first one can be used to get information about the default audio device. It is called `AudioDeviceInfo` (see Listing 5.15). Once this class is instantiated, the method `getSampleRate` can be used to get the native sample rate of the device. The `getFramesPerBuffer` method returns the native buffer size of the device in samples. These values can be used by an audio API implementation to match its output parameters to the native values used by the device. This can reduce the latency added by the system by allowing access to the fast audio mixer [21]. How these values are acquired depends on the Android API version. Starting with API 17 these values can be read directly from an instance of the `AudioManager` class. On older API versions the `AudioTrack` class is used to get these values. While the native sample rate can be queried directly, the buffer size is more difficult to get. It requires extending the `AudioTrack` class in order to call the protected `getNativeFrameCount` method. Since this function is deprecated since API 19 both methods using the `AudioTrack` class are also annotated as deprecated. Because the audio API should be backwards-compatible down to Android API 10 they are still necessary as fall-back. When instantiating the `AudioDeviceInfo` class the correct method to populate the internal values is chosen automatically.

Another utility class allows the loading of files using the WAVE format into an audio data object. It is called `WaveFile` (see Listing 5.16) and only defines one static method: `decodeStream`. Since no instantiation is needed the access to the constructor is set to private [3]. This method takes an input stream and reads the content of a WAVE file from it. The file must use single-channel 16-bit PCM samples. If the file is not a

```
@Immutable
public final class AudioDeviceInfo {
    public AudioDeviceInfo(Context context);
    public int getSampleRate();
    public int getFramesPerBuffer();
    @TargetApi(Build.VERSION_CODES.JELLY_BEAN_MR1)
    static int getOutputSampleRate(
            AudioManager audioManager);
    @TargetApi(Build.VERSION_CODES.JELLY_BEAN_MR1)
    static int getOutputFramesPerBuffer(
            AudioManager audioManager);
    @Deprecated
    static int getOutputSampleRate();
    @Deprecated
    static int getOutputFramesPerBuffer();
}
```

Listing 5.15: `AudioDeviceInfo` class

valid WAVE file or is uses a different audio format the method throws an `IOException`. Since the input stream is not opened by the method it also does not close it when finished, this has to be done by the user.

```
public final class WaveFile {
    @Nonnull
    public static AudioData decodeStream(
            @WillNotClose InputStream inputStream)
            throws IOException;
}
```

Listing 5.16: `WaveFile` class

WAVE files store the audio data uncompressed. While this makes them very easy to decode, they take up a lot of space. There are several audio codecs that try to reduce the file size of audio files. I decided to implement support for the Vorbis format, since it is a free codec and its decoder library is licensed under a BSD-like license [75]. Like the MP3 codec, Vorbis encodes the audio data in a lossy way. Vorbis encoded audio data is usually contained in a file using the Ogg container format.

The class used to decode Ogg Vorbis files is called `OggVorbisFile` (see Listing 5.17). Like with the `WaveFile` class, the decoding is done using a static method called `decodeStream`. Internally this method calls another native method to do the actual decoding using the libogg and libvorbis libraries. The native method is implemented in a

JNI library, using the native utility library (see section 5.1). By using the `getLibraryABI` method the processor instruction set the native library was compiled for can be queried. In order to not make the audio API module dependent on any native code, this class is contained in a separate module.

```
1 public final class OggVorbisFile {
2     @Nonnull
3     public static AudioData decodeStream(
4             @WillNotClose InputStream inputStream)
5             throws IOException;
6     @Nonnull
7     public static native String getLibraryABI();
8 }
```

Listing 5.17: `OggVorbisFile` class

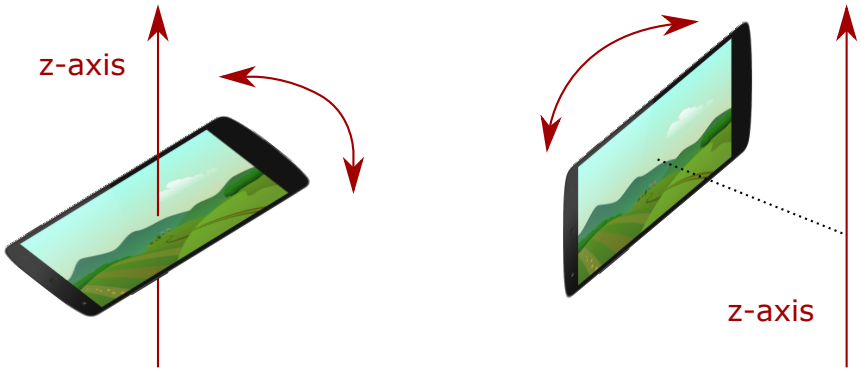
The main audio API module also contains a class that loads audio files using the codecs provided by the device. It is called `MediaFile` (see Listing 5.18). Unlike the WAVE or the Ogg Vorbis decoder this class does not use an input stream. Instead, the `decodeAsset` method takes an Android asset file descriptor as argument. It then determines the type of the audio file, decodes it and returns the resulting audio data. Internally, this class is implemented using the `MediaCodec` class that is part of the Android API since version 16 [30]. Because of this, the `decodeAsset` method can also only be used on API version 16 or above.

```
1 public final class MediaFile {
2     @Nonnull
3     @TargetApi(Build.VERSION_CODES.JELLY_BEAN)
4     public static AudioData decodeAsset(
5             AssetFileDescriptor assetFileDescriptor)
6             throws IOException;
7 }
```

Listing 5.18: `MediaFile` class

5.3 Sensor API

Letting the user control the application by rotating the phone to point into the desired direction can be an intuitive way to change the orientation. This is possible by using the



(a) Lying on a surface (b) Held by the user

Figure 5.4: Possible device orientations

sensors available on the Android device. Which sensors are available can vary from device to device. The goal of the sensor API is to provide a common way to access the sensors capable of providing information about the current orientation of the device.

5.3.1 Rotation Sensor API

All sensors available for the sensor API implement the `RotationSensor` interface (see Listing 5.19). This interface provides access to current the orientation of the device around an axis perpendicular to the ground and pointing to the sky. This value is independent of the position of the device. The sensor should for example work if the device is is lying on a table (see Figure 5.4a) as well as when it is held in front of the user (see Figure 5.4b). It should also work in both landscape and portrait mode. Implementations of this interface serve as an adapter to the real sensor [9].

```
1  public interface RotationSensor {
2      interface OnAzimuthChangeListener {
3          void onAzimuthChange(RotationSensor rotationSensor,
4                               float azimuth);
5      }
6      void addListener(OnAzimuthChangeListener listener);
7      void removeListener(OnAzimuthChangeListener listener);
8      boolean isEnabled();
9      void setEnabled(boolean enabled);
10 }
```

Listing 5.19: `RotationSensor` interface

To access the orientation the observer pattern is used [9]. The methods `addListener` and `removeListener` can be used to register or unregister an observer of the type `OnAzimuthChangeListener`. By extending this interface the observer must provide a method called `onAzimuthChange`. Every time an active sensor the observer is registered with gets a new reading, this method is called. The current orientation (called azimuth) is provided as counter-clockwise rotations in radians. For a sensor to become active it first has to be enabled. This is done using the `setEnabled` method. If the application does not require orientation data any more it should disable the sensor. The sensor is enabled and disabled on a per-instance basis, other instances of the sensor are not affected.

To create a `RotationSensor` instance, each type of sensor should provide a factory object. This factory should implement the `RotationSensor.Factory` subinterface (see Listing 5.20). By defining a common factory interface, a sensor instance can be created without having to know the actual type of the sensor [9]. Instead, only an instance of the factory creating the chosen sensor needs to be available. This factory interface is very similar to the audio back-end factory interface (see subsection 5.2.2). Like its audio API counterpart, the `create` method takes a `Context` object as its only parameter. It then either returns the created sensor instance, or `null` if the sensor is not supported on the device. The `getSensorName` method returns a short description of the sensor type created by the factory.

```
1 public interface RotationSensor {
2     interface Factory {
3         @Nonnull
4         String getSensorName();
5         @Nullable
6         RotationSensor create(Context context);
7     }
8 }
```

Listing 5.20: `RotationSensor.Factory` interface

To facilitate the implementation of rotation sensors, the `AbstractRotationSensor` class (see Listing 5.21) contains several methods common to most implementations of the `RotationSensor` interface. One of its features is that it manages the addition and removal of the observers. When the sensor reads a new value, the super-class should call the `notifyListeners` method. This method will then notify all registered observers. The class also manages the state of the sensor. When the state changes,

that is the sensor is enabled or disabled, the `onEnabled` method is called. The superclass can then react to the change. Finally, it defines the static field `HANDLER` using the type of the same name, which represents a thread with a message queue [26]. This thread can be used to receive sensor value updates. This prevents blocking the main thread of the Android application, even if a listener does an expensive or blocking operation in its notification method. By defining one thread common to all sensor instances, unnecessary thread creation can be avoided. The thread is started when the class is first loaded and exists until the application is exited. Because a sensor implementation can make use of this update thread, the listener notification method has to be implemented in a thread-safe way.

```
1  abstract class AbstractRotationSensor
2          implements RotationSensor {
3      protected static final Handler HANDLER;
4      protected final SensorManager sensorManager;
5      protected AbstractRotationSensor(
6              SensorManager sensorManager);
7      @Override
8      public final boolean isEnabled();
9      @Override
10     public final void setEnabled(boolean enabled);
11     protected abstract void onEnabled(boolean enabled);
12     @Override
13     public final void addListener(
14             OnAzimuthChangeListener listener);
15     @Override
16     public final void removeListener(
17             OnAzimuthChangeListener listener);
18     protected final void notifyListeners(float azimuth);
19 }
```

Listing 5.21: `AbstractRotationSensor` class

5.3.2 Sensor Data Filters

Some sensors need filters to improve the stability of the data they receive. The filters created as part of the sensor API all implement a common interface called `SensorFilter` (see Listing 5.22). Unlike the audio data filters (see subsection 5.2.4) the sensor filters are designed to manipulate input data in real time. Each sensor filter implementation must implement two methods. The first is called `applyFilter`. It

takes an array of floats as its only parameter. Each time a sensor on Android updates, a `SensorEvent` is generated [36]. This event can then be received by event listeners registered for this sensor. The actual sensor readings are stored in an array of floats accessible using the `values` field of the sensor event object. This field can be used directly as parameter for the `applyFilter` method. While the length of the value array depends on the type of the sensor, each filter expects to always receive arrays of the same length. This means a single instance of a filter class should not be used with different types of sensors. Instead a new instance should be created for each sensor. The `applyFilter` method returns an array of filtered values of the same length as the input array. Like the original values array, this array should not be directly modified. If the filter has an internal state it can be reset using the `clearFilter` method.

```
1 interface SensorFilter {
2     void clearFilter();
3     @Nonnull
4     float[] applyFilter(float[] values);
5 }
```

Listing 5.22: `SensorFilter` interface

Like the audio data filters, multiple sensor filters can be applied consecutively. This is done using an instance of the `SensorFilterChain` class (see Listing 5.23). Its constructor takes a list of sensor filter instances. After instantiating the object these filters should no longer be used on their own. When the `applyFilter` method of the sensor filter chain is called, all filter are applied in the order they were given to the constructor. Each filter is applied to the results of the previous filter. Once all filter have been applied, the result is returned. The `clearFilter` method simply calls the same method for each filter instance that is part of the chain.

```
1 final class SensorFilterChain implements SensorFilter {
2     SensorFilterChain(SensorFilter... filters);
3     @Override
4     public void clearFilter();
5     @Nonnull
6     @Override
7     public float[] applyFilter(float[] values);
8 }
```

Listing 5.23: `SensorFilterChain` class

Currently there are two filter classes implemented. The first is the `LowPassFilter`. This class implements a simple low-pass filter that can be used to isolate the force of gravity from an accelerometer reading. It is implemented using the formula shown in Equation 5.3, which is also described in the Android sensor event documentation for the accelerometer [36]. The formula uses only a constant factor (α), the previous filtered value ($gravity_i$) and the new sensor reading ($sensor_i$). To create an instance of the filter the user has to supply the dimension of the arrays that will be filtered as well as the α value. The filter is applied to each value of the array individually. Since the filter has to support a high frequency of calls, it uses an internal array to store the filtered values. They are then overwritten on the next call. The filter itself does not make a defensive copy when returning the results [3]. This prevents the unnecessary allocation of a new array on each run of the filter function. If the user needs to preserve or modify the values returned by the filter, a copy has to be made. Using the returned array with another filter or a function designed to take the values array of a sensor event as argument is safe, in that case no copy has to be made.

$$gravity_i = \alpha \cdot gravity_i + (1 - \alpha) \cdot sensor_i \tag{5.3}$$

The other implemented filter is the `HistoryFilter`. It can be used to smooth the output values of a filter. This is done by keeping a history of the previous readings, up until a maximum number of values. When applied to a new set of values, the oldest pair of values is removed, the new pair is saved and the average of all currently saved values is returned. The history is implemented as a circular buffer, removing the need for allocations to save a new set of values. The calculation is done independently for each dimension of the filter values. Like the low-pass filter, the array used to return the filtered values is reused the next time the filter is applied. This means the returned array has to be copied by the user if their value should be preserved after returning from the callback functions. The constructor of the filter takes the dimension of the arrays that will be filtered as well as the size of the circular buffer used to save the history. Clearing the filter resets all values in the buffer to zero.

5.3.3 Available Sensor Implementations

There are currently four supported sensor types, implemented in two classes. In addition there is a virtual sensor, called `NullSensor` (see Listing 5.24). This sensor

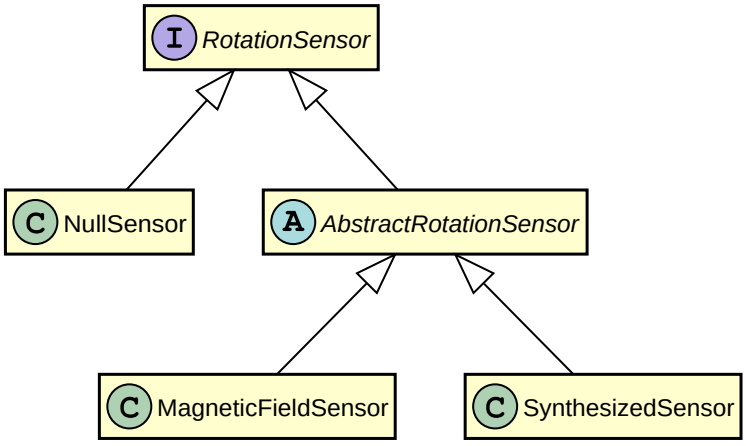


Figure 5.5: Sensor API classes

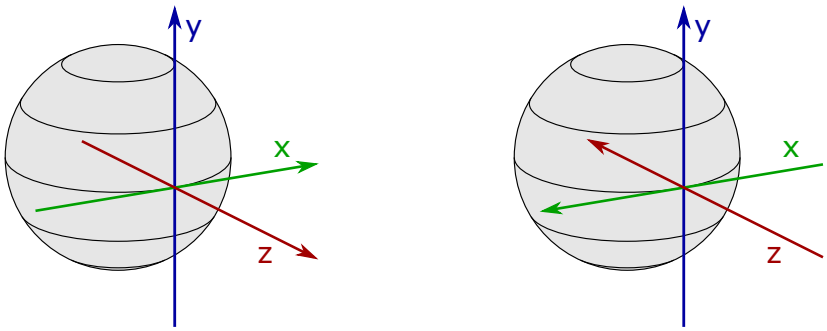
implementation is not backed by a real sensor and never updates. Is is always enabled and all of its methods do nothing. Since it has no internal state, the class is immutable and implemented as a Singleton using the enum type [3]. The factory of the null sensor always returns this Singleton instance. This class can be used when code expects a sensor object but either no real sensor is available or any sensor input should be ignored.

```
1 @Immutable
2 public enum NullSensor implements RotationSensor {
3     INSTANCE;
4     public static final Factory FACTORY;
5     /* RotationSensor implementation omitted */
6 }
```

Listing 5.24: `NullSensor` class

The real sensor implementations are grouped into two classes. Both extend the `AbstractRotationSensor` class to implement the basic sensor functionality. Their class hierarchy can be seen in Figure 5.5.

The first class is called `MagneticFieldSensor`. It uses the magnetic field sensor in combination with a gravity sensor to allow the device to function as a compass. This sensor measures the ambient magnetic field relative to the device, which can be transformed into a real-world orientation if the gravity vector is known. The sensor class supports two ways to determine this vector. One is to use the accelerometer to determine the direction of the acceleration caused by gravity. The accelerometer data



(a) Normal (b) Inverted

Figure 5.6: Sensor world coordinate systems [37]

is transformed with a low-pass filter to try and remove any acceleration caused by the user [36]. Some devices also provide a virtual gravity sensor [35]. This sensor type returns the gravitational acceleration directly. Which sensor is used depends on the factory that is used to create an instance of the `MagneticFieldSensor` class (see Listing 5.25). A history filter is then applied to the data before any further calculations are done. In case of the accelerometer this is done after the low-pass filter was applied.

```
1 public final class MagneticFieldSensor
2         extends AbstractRotationSensor {
3     public static final Factory FACTORY_ACCELEROMETER;
4     public static final Factory FACTORY_GRAVITY;
5     /* AbstractRotationSensor implementation omitted */
6 }
```

Listing 5.25: `MagneticFieldSensor` class

To get the actual device rotation the `getRotationMatrix` method of the sensor manager is used [37]. This returns a rotation matrix using the coordinate system shown in Figure 5.6a. The method may fail when the device is in free-fall since in that case the gravity vector can not be determined. If the creation of the rotation matrix succeeds, the `getOrientation` method of the sensor manager is then used to determine the Euler angles describing the rotation. The rotation around the Z axis, called the azimuth, is then forwarded to any listeners of the sensor. Since the Euler angles are relative to a different world coordinate system (shown in Figure 5.6b) with an inverted Z axis, the azimuth is negated before the listeners are notified.

The second sensor class is called `SynthesizedSensor`. It uses the rotation vector sensor to get the orientation of the device in the world space. This sensor is a virtual sensor that can use several real sensors present on the device to derive the orientation. The used sensors typically include the accelerometer, the magnetic field sensor and, if present, the gyroscope. The sensor readings are also already filtered using a Kalman filter, making additional filtering unnecessary [66]. When a new reading becomes available, it is returned as a rotation vector [36]. Using the `getRotationMatrixFromVector` method of the sensor manager this rotation vector can be converted to a rotation matrix [37]. Then the orientation can be extracted using the same procedure as with the `MagneticFieldSensor`. On Android API 18 or above, a second type of rotation vector sensor is available, called the game rotation vector sensor. This sensor does use the same techniques as the normal rotation vector sensor, except it does not use the geomagnetic field [36]. The `SynthesizedSensor` (see Listing 5.26) class provides factory objects for both types of sensors. Thanks to their use of the gyroscope and a Kalman filter both sensors can be expected to provide much more accurate readings than the magnetic field sensor on its own. Since, for the purpose of a game, the true position of the earth's poles is not relevant, the game version of the sensor should be used if available.

```
1 public final class SynthesizedSensor
2         extends AbstractRotationSensor {
3     public static final Factory FACTORY_DEFAULT;
4     public static final Factory FACTORY_GAME;
5     /* AbstractRotationSensor implementation omitted */
6 }
```

Listing 5.26: `SynthesizedSensor` class

6

OpenAL

For this project the positional audio API for Android that was introduced in section 5.2 is implemented using the OpenAL Soft audio library. Both the library as well as the code used to integrate it into the project will be described in this chapter.

6.1 The OpenAL Library

OpenAL (for "Open Audio Library") was designed as an API to access audio hardware. It was first implemented by Loki Entertainment Software, a company specializing in porting games to the Linux operating system. The API was then standardized in collaboration with Creative Labs and the OpenAL 1.0 specification was released in the year 2000. Implementations for Windows, Linux, MacOS and BeOS were released in the same year. Creative Labs also released a hardware-accelerated version supporting their SoundBlaster sound card. Support for other operating systems and hardware has since

been added. The OpenAL API is specifically designed to produce multichannel audio output of three-dimensional arrangements of sound sources around a listener [45].

OpenAL Soft is a software implementation of OpenAL, licensed under the LGPL. It supports multiple platforms and several audio output back-ends, including OpenSL ES [65]. OpenSL ES is another audio API designed to access audio hardware. It was designed as a cross-platform audio interface for embedded and mobile devices by the Khronos Group [69]. It is supported on Android since version 2.3 or API version 9 [32]. Although OpenSL ES defines a profile that support 3D audio [69], this functionality is not available on Android [32]. Since OpenAL Soft implements its own 3D audio mixer and does not rely on the 3D audio functionality of OpenSL, it can be still be used to create positional audio on Android.

OpenAL Soft includes support for head-related transfer functions since version 1.14. This feature has to be explicitly enabled using an option in the configuration file. The HRTF data has to be converted to a special format before it can be used. The source for a conversion tool is supplied with OpenAL Soft. The following description of the format is based on the documentation included in the OpenAL Soft source code. In the HRTF data set format used by OpenAL Soft, the data is stored in a little-endian binary format, using a separate file for each supported sample rate. The conversion tool can be used to resample such a data file for other sampling rates. To reduce the necessary filter length when applying the HRTF, the stored HRIRs are minimum-phase. When using minimum-phase HRIR data, it is necessary to apply an additional delay to generate the correct ITD [53]. This delay is also stored for each HRIR. The position of each individual pair of HRIR values and corresponding delay is defined by an azimuth and an elevation angle. The format supports a minimum of 5 and up to 128 different elevation angles. For each elevation between 1 and 128 azimuth angles can be used. Only the data for the left ear is stored. For the right ear the same data is used, but with the azimuth reversed. The HRIR data is stored as signed 16 bit samples. OpenAL Soft defaults to using a 32 point filter. To reduce the workload for the processor I also created 16 point filter versions of the HRTFs used by the application.

6.2 Build Process

The OpenAL Soft library source code includes a CMake project configuration file that provides several options to configure and build the library as well as the included HRTF

file conversion utility. This allows the library to be built for Android using an appropriate toolchain configuration file. The actual build process of the library as used in this project is started using another CMake configuration file.

This CMake project configuration file first builds a small helper library created for this project. This library is called `fopen-wrapper` and can be used to wrap calls to the `fopen` function and redirect them. It only consists of one source and one header file and has one publicly visible function called `fopen_wrapper_set_callback`. This function takes a pointer to a callback function as its only argument. The callback function defined by the pointer has the same signature as the `fopen` function. Internally the library also defines a wrapper function for `fopen`. When it is called, it first checks if a callback function was set. If it was, the callback is called and its result is returned. Otherwise the function will simply return `NULL`, indicating a failure to open the requested file. This helper library is built as a shared library.

After that, the CMake project file configures the build of the OpenAL library. First it disables the generation of any example or documentation files. It also prevents the HRTF utility from being build. To use this utility it should be build separately for the host system. It also disables the WAVE file output back-end and instead forces the build of the OpenSL back-end. Finally, the global CMake linker flags are used to configure the linker program to wrap any calls to `fopen` and to link the `fopen-wrapper` library to handle these calls instead. This is done using the `wrap` option of the GNU linker used by the Android NDK. Then the OpenAL library is build by including its subdirectory, which completes the build process. The resulting shared libraries are included in an otherwise empty Android library module. The OpenAL JNI module can then declare this module as dependency and have access to the compiled files.

6.3 Configuration

OpenAL does not use HRTFs by default, instead they have to be enabled in a configuration file. On systems that are not Windows, this configuration file is searched in the `etc` configuration directory as well as in the home directory of the user. Once configured, the HRTF files are also loaded from the filesystem. One way to configure OpenAL on Android would be to create he configuration file on the internal data storage path of the application and using environment variables to make OpenAL search at the correct location. But in this project another approach is used. As described in the previous

section, the OpenAL shared library file build by this project it is linked against another library that wraps all calls to the `fopen` function. Since OpenAL uses `fopen` to open its configuration files, this wrapper can be used to provide the content of these files from other sources.

To handle the intercepted calls to `fopen`, the `OpenAlFileMapper` class is used. This class is implemented as a singleton because only one callback function can be active at any time [9]. Since the class needs to be initialized with a reference to an Android context object, the singleton uses lazy initialization instead of defining an enum class for the singleton [3]. Except for the static `initialize` method that does the lazy initialization, the class only has private Java methods. For this reason the initialization function does also not return the singleton instance of the class. When initialized, the class uses JNI to register a native callback function with the `fopen-wrapper` library. It also sets some environment variables to cause OpenAL to use a logfile and to set a specific logging level.

When OpenAL now tries to open a file, the call is handled by the wrapper function defined in the `fopen-wrapper` library, which then calls the native callback function of the `OpenAlFileMapper`. This function first uses the JNI utility library to attach the calling thread to the JVM if it is not already attached. It then, again using the utility library, calls the `fopenCallback` Java method of the `OpenAlFileMapper` class to determine how to handle the request. The class can then use the `fopenInputStream` and `fopenOutputStream` functions to map Java input or output stream to a file pointer. Internally these functions also use the utility library described in section 5.1. The pointer is then returned to the JNI callback function and from there through the wrapper to the original call to `fopen`. This whole process is also shown in Figure 6.1.

There are three types of file request that are handled by the file mapper class. The first is a request to open the main OpenAL configuration file for reading. Instead of trying to open the, on Android systems non-existing, global configuration file in `etc`, the

```
1 [general]
2     channels = stereo
3     stereo-mode = headphones
4     hrtf = true
5     hrtf_tables = /etc/openal/mit-kemar-%r-16.mhr
```

Listing 6.1: OpenAL configuration file

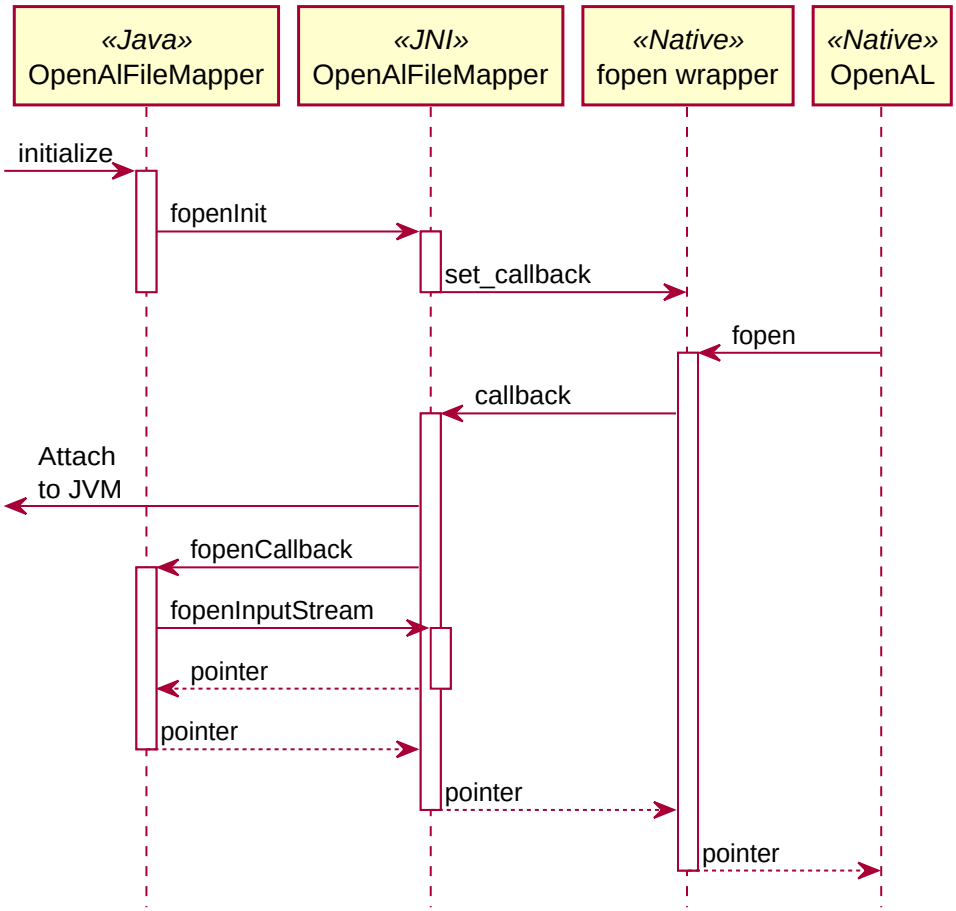


Figure 6.1: OpenAL file mapper operation

file is instead dynamically created in memory as a `ByteArrayInputStream`. This stream is then mapped to a file pointer and returned. The generated configuration file (see Listing 6.1) sets the output mode to stereo for headphones and enables the use of HRTFs. It also configures which HRTF file to use. Currently this is set to use the MIT KEMAR files with a 16 point filter. The necessary files are included as assets in the OpenAL module of this project. While OpenAL has configuration options to set the native sample rate and buffer size of the audio device, these settings are ignored by the OpenAL library when using the OpenSL ES back-end. After specifying them in the virtual configuration file, the file mapper also has to handle the calls to open the HRTF definition files. These requests are handled by loading the requested files from the asset store of the Android application and then mapping their input streams to file pointers. Finally, calls to open the log file defined per environment variable for writing

are also handled. Here a special output stream is created that takes all messages written into it and prints them using the Android logging capabilities. This facilitates the debugging of any OpenAL related problems. Any other file request are rejected by returning a null pointer.

6.4 Audio API Implementation

The Java wrapper for OpenAL created for this application implements the audio API described in section 5.2. Before using OpenAL, an instance of the `OpenAlBackend` class has to be created using the provided factory. Since this factory object is a subclass of the back-end class, the classes static initialization block is already executed when the factory object is accessed. In the static initialization block the required shared libraries are loaded. The factory also initializes the file mapper before creating an instance of the OpenAL back-end class (see Figure 6.2). Since OpenAL only loads its configuration on the first initialization, this is necessary to guarantee that the file mapper is already initialized and its callback registered when OpenAL is used for the first time.

The `OpenAlBackend` class extends the abstract `ManagedAudioBackend` class described in subsection 5.2.3. This helper class keeps track of all created audio buffers and disposes them when the back-end itself is disposed. The class uses native functions to access the OpenAL API. The device and context pointers used by OpenAL are stored in the Java class as fields using long integers. This allows them to be safely stored even on systems using 64-bit pointers. Since the native resources acquired by OpenAL can not be released by the garbage collector, the `dispose` method is used to free these resources. This method has to be called on an instance of the class before it is collected by the garbage collector. As a safeguard the class overrides

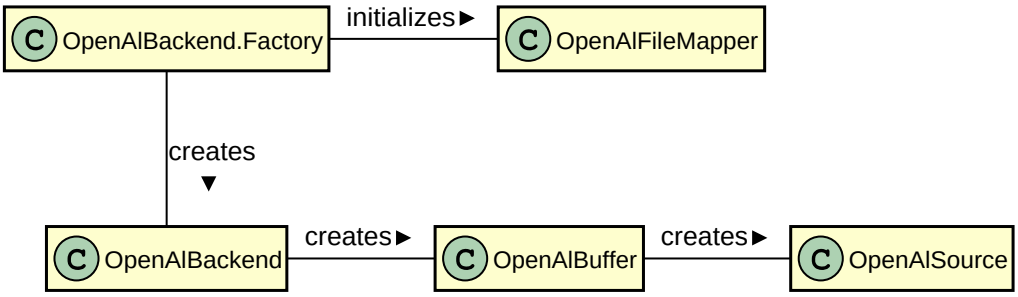


Figure 6.2: OpenAL classes

the `finalize` method [3]. When an instance of the class is finalized, it checks if the `dispose` method was called. If it wasn't, it is called now and a warning is written to the log. The `OpenAlBackend` class also defines a method called `getLibraryABI` which returns the name of the ABI the library was compiled for as a string. This can be used to check if the correct version of the shared library was loaded. Otherwise, when using split APKs, it can happen that a version for an ABI compatible only through emulation was installed. This could severely impact the performance of the library and should be avoided.

Both the `OpenAlSource` as well as the `OpenAlBuffer` class also extend their managed counter-parts. Like the back-end, they are mostly implemented in native code. Both also store the pointer to the OpenAL context in a field of the type `long`. In addition, the identification number for their backing buffer or source is simply stored in an integer, allowing the native methods to access them easily. They also both overrider the `finalize` method, implementing it in a similar fashion to the OpenAL back-end class. Since OpenAL uses a Cartesian coordinate system while the audio API uses polar coordinates, the OpenAL source class has to translate between the two.

While calls to OpenAL itself are thread-safe, the implementation of the audio API methods often need to make several calls to achieve the desired result. For these calls to succeed, they have to be executed while the correct global OpenAL context is active. If the global context is changed by another thread the call will fail which is translated into an exception by the JNI code. To prevent this from happening, every native method that implements the audio API using OpenAL has to acquire a global lock before executing. This way they can safely be used from multiple threads.

7

Game Engine

The application described in this thesis is designed as a game in order to motivate the user. The game engine module contains classes that facilitate the development of such a game. The contents of this module are described in this chapter.

7.1 Architecture

A game engine is a software framework designed to facilitate the development of a video game. By moving the code that is not specific for one single game into an engine project, it can easily be reused for other games of the same type. This also allows the engine to be developed and improved separately and have all games based on the engine profit from these changes. For these reasons I decided to put the parts of the application that might be useful for other projects of a similar kind into a game engine module. This game engine uses the audio and sensor APIs described in chapter 5.

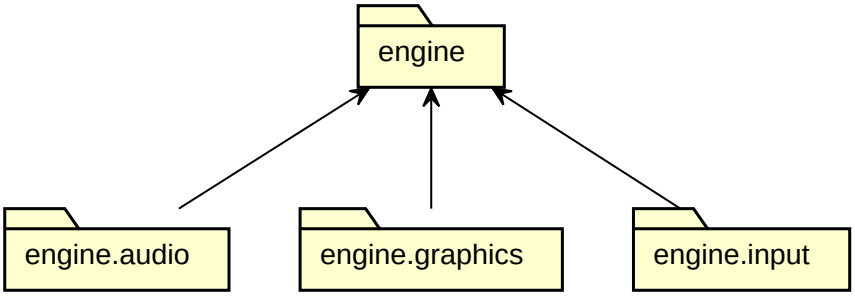


Figure 7.1: Game engine package dependencies

7.1.1 Structure

The game engine designed for this thesis is split into a graphics, an audio and an input package (see Figure 7.1). Each of these depends on the main engine package, but is independent of the other packages. The engine implements the model-view-controller architecture. The data, or the model, is implemented in the main engine package. Both the graphics and the audio package provide views of the data. How the data is rendered is configured using components, which will be described later in this chapter. The input package provides controllers that may manipulate the data. A direct manipulation by the user of the game engine is also possible.

Traditionally many game engines are based around the concept of a main loop [54]. This is a loop that starts running when the game begins and is only stopped when the game ends. It usually processes any user input, updates the game state, renders the game and then repeats. For this game engine I decide to not use a game loop, but to implement a callback based system instead. This means that instead of running a loop and calling the different sub-systems, the engine relinquishes control back to the application and is only called when an update becomes necessary, similar to the Observer pattern [9]. The callback based approach fits well with the different Android API classes that also use this design, like for example the `Activity` class [12] or the `Renderer` interface [24].

The model of the game engine consists of exactly one player and one world object. The world is a 2-dimensional plane with the player at the center (see Figure 7.2). When a view is added, it typically renders the world from the position of the player. Since movement of the player is currently not part of the game, the position of the player is fixed. Rotation of the player on the other hand is possible. The game can also contain entities. These belong to the world object and can be positioned anywhere in it. Their

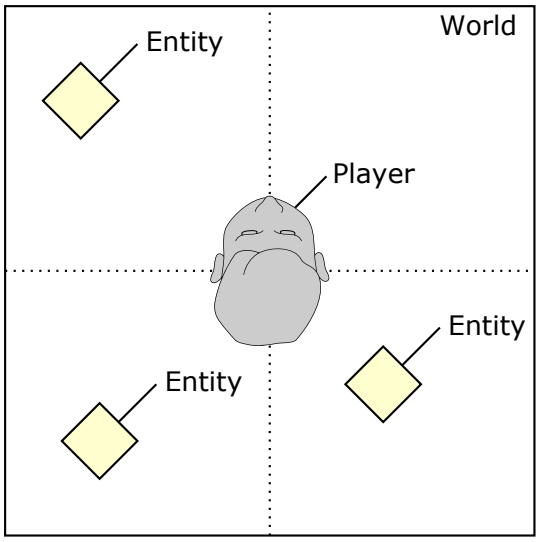


Figure 7.2: The game world, the player and three entity objects

position is expressed in polar coordinates. Unlike the player they are not directional and can not be rotated.

These entities are used to decorate the world, to place ambient audio sources or to create a target for the player. The engine supports creating and destroying entities dynamically during the game. This can for example be used to add a new target or to remove a target the player has discovered. When used to create a target the entity is both an audio source as well as a visible object in the world. But since is should only become visible once the player has decided where they suspect the target to be, the visibility of the entity has to be dynamically controllable.

7.1.2 Components

There are several ways of implementing the entity objects used by the game engine. Since there are different kinds of entities one way would be to use different classes. In this object-oriented design one could for example create a base entity class. This class would then be extended by an audio entity as well as a graphic entity class. The graphic renderer could then check every added entity if it is an instance of the graphic entity class and render the entity if it is. The audio player would do the same, only with the audio entity class. But this leads to a problem when creating the target entity class: since it has an audio as well as a graphical aspect, it should extend both these

classes. This is known as multiple inheritance and is not supported by Java. The type comparison could be avoided by implementing the strategy pattern instead [9]. In that case a common interface for all entities would be defined. This interface would contain a draw method that can be called by the renderer when the implementing entity instance should be drawn. The entity can then draw itself, or simply do nothing if it is invisible. The audio player would be implemented in a similar way. When implementing these methods for an entity type, it is often desirable to be able to reuse code. For example the target entity implementation might want to reuse the drawing code from a decorative entity. Since it might also want to reuse the code of another audio playing entity, the multiple inheritance problem is also present with this solution. This approach would also lead to situations where both graphical as well as audio code is implemented in the same class. Separating the two systems is then not longer easily possible.

To solve this problem a design based on composition instead of inheritance is used [3]. In this design the entity class contains merely the data common to all types of entities. For the purpose of this game engine, this is only the position of the entity in the game world. In addition, each entity object can hold different components [54][63]. These components can then be queried and used by the different systems, like the renderer or the audio player. The component classes can easily be combined in any variation necessary to define the desired type of entity. Since neither the entity class needs to know about the actual types of the components, nor do the components need to know about each other, this design minimizes the dependencies between the different systems. As seen in Figure 7.1 the package containing the graphics rendering classes is completely independent of the package containing the audio playback classes, and vise versa. The main engine package, which implements the model, is itself also not dependent on the different packages that implement either views or controllers. This system also allows users of the game engine to define and implement their own components and their associated systems without having to change the engine module.

7.2 Engine

The `engine` package contains the basic classes used by the game engine. Here the interfaces and classes of the component system described in the previous section are defined. The package also contains the classes used to create and manage the game

world. Both these and the utility classes contained in the package will be described in this section.

7.2.1 Components API

The components API allows to add and remove objects, called components, from a parent object at runtime. These components can then be retrieved from the parent object in a type-safe way. The parent class does not need to know about the component at compile time. By using observers, other engine systems can be notified when a component of a certain type is added, removed or updated.

All components used with the engine must implement the `Component` interface (see Listing 7.1). This interface specifies two generic type parameters. The first one, called `C`, defines the type of the component. It is meant to be used as a recursive type parameter, similar to the one in the `Enum<T>` type definition. While there is no way to prevent the definition of a component class that is not an instance of the type specified in this generic parameter, such a component can not be used with the other parts of the component API. The second generic type parameter, called `P`, defines the class of the parent this component can be added to. This interface also extends the `Parcelable` interface. This means any component can be stored and later retrieved from an Android `Parcel` instance.

```
1  public interface Component<C extends Component<C, P>,
2                             P extends ComponentContainer<P>>
3          extends Parcelable {
4      @Nonnull
5      Class<C> getComponentType();
6      @Nonnull
7      C newInstance(P parent);
8      void onRemoved(P parent);
9      void onParentUpdate(P parent);
10 }
```

Listing 7.1: `Component` interface

The interface also defines several methods. The `getComponentType` method returns the type of the component, as defined by the generic type parameter `C`. This is necessary, since because of type erasure, the generic parameter is not longer available at runtime without using reflection. When a component is added to a parent object its

`newInstance` method is called. This method gets the parent object of the component as its only argument and should return a copy of the component in its current state. This way the component can be used as a prototype [9]. The component can than be modified without these modifications affecting the other component instances previously added to different parent objects. Since the component is copied every time it is added to a new container, each component instance can only have one or zero parents. Once the component is added to a component container, it may receive two additional callbacks. When the component is removed from the container, the `onRemoved` method is called. For convenience it also receives the parent object the component was removed from as parameter. If the parent is modified, the `onParentUpdate` method is called. It also receives the parent object as parameter. This callback can be used to notify any listeners of the component that the container object was changed.

Since most component implementation only require a reference to their container to notify it about any changes to the component, this functionality is implemented in an abstract helper class. It is called `BaseComponent` (see Listing 7.2). This class implements the `newInstance` method of the `Component` interface by first calling an abstract method, also called `newInstance` but without parameters, that has to return a new instance of the component using the current instance as prototype. Before this copy is then returned by the original `newInstance` method, a reference to the parent object is stored in the new instance. If the component is removed, the `onRemoved` implementation clears this reference. When the component is modified, the protected `notifyUpdate` method can be used to notify any listeners. Since a parent object is required to notify the listeners, this method does nothing if the component does not have a parent. The `onParentUpdate` method is implemented to also call `notifyUpdate`.

```
1 public abstract class BaseComponent
2         <C extends BaseComponent<C, P>,
3          P extends ComponentContainer<P>>
4         implements Component<C, P> {
5     protected final void notifyUpdate();
6     @Nonnull
7     protected abstract C newInstance();
8     /* Component implementation omitted */
9 }
```

Listing 7.2: `BaseComponent` class

Unlike the other overridden methods it is not declared final, so this behavior can be suppressed.

To use these components, they have to be added to a component container. Such a container is defined by the abstract `ComponentContainer` class (see Listing 7.3). Like the `Component` interface, this class expects its actual type as type parameter. This parameter, called `P`, has to be identical with the one used by any components that may be used with this container. The `getThisAsParent` method has to be implemented by any subclasses to return a reference to `this` using the actual type of the implementing class. The class is package-private since it is only meant to be used by the game engine itself. Each instance of this class keeps a reference to the event manager used by the game it belongs to. The event manager can be queried using the `getEventManager` method. Components can be added and removed using the `addComponent` and

```
abstract class ComponentContainer
        <P extends ComponentContainer<P>> {
    /* Generic type parameter declarations for methods
       are omitted, where applicable C is defined as
       C extends Component<C, P> */
    protected ComponentContainer(EventManager eventManager);
    @Nonnull
    public final EventManager getEventManager();
    @Nonnull
    protected abstract P getThisAsParent();
    @Nullable
    public final C getComponent(Class<C> type);
    @Nonnull
    public final C getComponentUnchecked(Class<C> type);
    @Nonnull
    public final Collection<Component<?, P» getComponents()
    @Nonnull
    public final P addComponent(C component);
    @Nonnull
    public final P removeComponent(Class<C> type);
    @Nonnull
    public final P clearComponents();
    public final void postComponent(C component);
    public final void postComponentCreatedTo(
            ComponentListener<C, P> listener);
    protected final void updateComponents();
    /* Parceling and copying methods omitted */
}
```

Listing 7.3: `ComponentContainer` class

`removeComponent` methods. The generic parameter signature prevents any invalid types of components from being added to the container. To store the components, a map is used where the type of the component serves as the key. This means only one instance per component type can be registered with the container. By using the `clearComponents` method all components can be removed from the container.

To get a specific component from the container the `getComponent` method can be used. It returns the component of the specified type, or `null` if no such component is registered. There is also an alternate version of this method, it throws an unchecked exception instead. This can be useful if the user is certain that a component of the specified type is registered with the container. The `postComponent` method causes the container to notify the event manager that the specified component, which has to be registered with this container, was updated. When a new listener for a component type is initialized, it may want to receive creation events for all currently existing components of the matching type. This can be accomplished by using the `postComponentCreatedTo` method. Calling this method sends a creation event if the container has a component of the type specified by the listener. This bypasses the event manager, the event is only received by the listener specified as parameter. The protected `updateComponents` method can be used to notify all registered components that this container was updated. Since the engine is designed to be used with multiple threads, the whole component container class is implemented in a thread-safe way.

As mentioned in the description of the component container, event listeners can be used to be notified whenever a component is added or removed from a container or otherwise updated. To create such a listener the `ComponentListener` interface has to be implemented (see Listing 7.4). The `getComponentType` method of this interface returns the type of the component the listener wants to receive updates for. This has

```
1 public interface ComponentListener
2         <C extends Component<C, P>,
3          P extends ComponentContainer<P>> {
4     @Nonnull
5     Class<C> getComponentType();
6     void onComponentCreated(P parent, C component);
7     void onComponentUpdated(P parent, C component);
8     void onComponentRemoved(P parent, C component);
9 }
```

Listing 7.4: `ComponentListener` interface

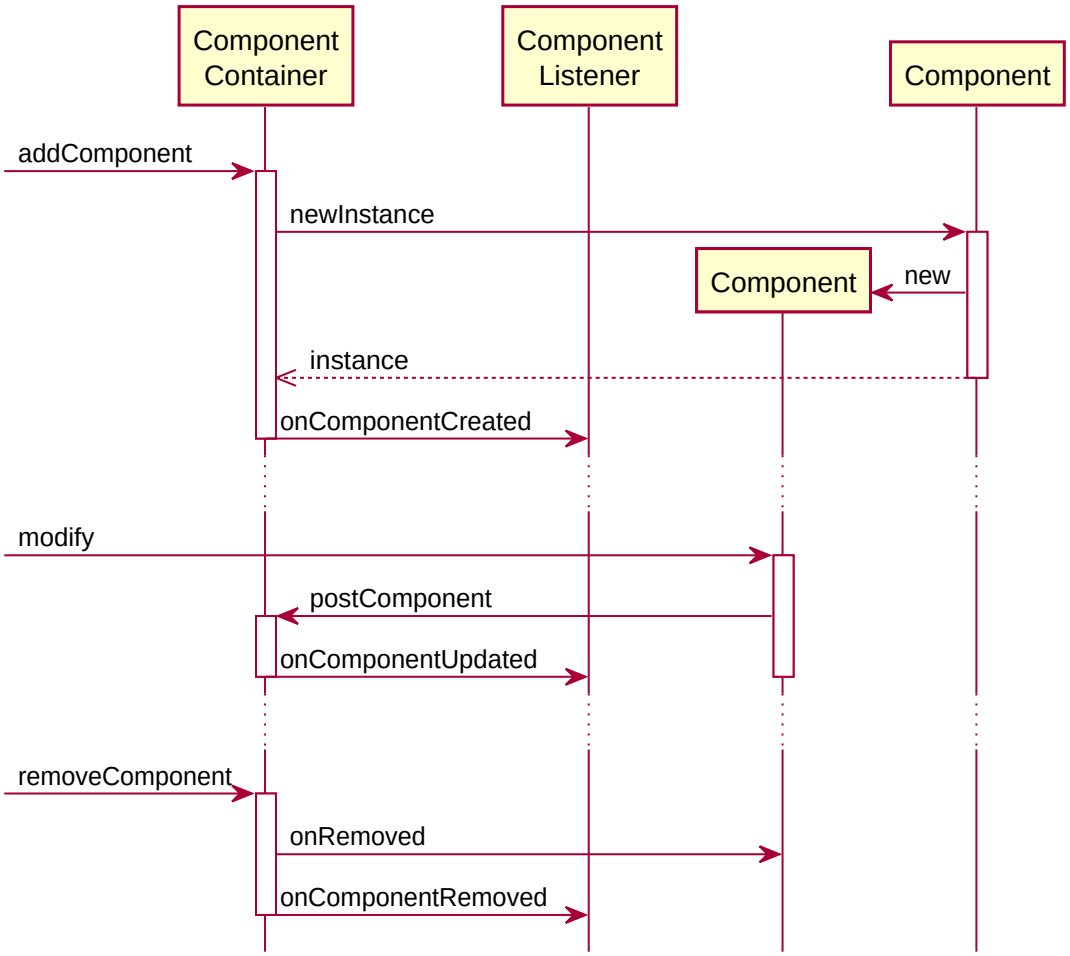


Figure 7.3: Component instance lifecycle

to be the same value that is returned by the method of the same name in the relevant implementation of the `Component` interface. The other three methods are callbacks. They are invoked by the event manager when a component of the relevant type is created, updated or removed. How these events relate to the lifecycle of a component instance is also shown in Figure 7.3.

To manage these listeners, an instance of the `EventManager` class is used (see Figure 7.4). Each game has exactly one event manager. This class simple defines methods to register and unregister listeners for components as well as the player class, which will be described later. It also has the same callbacks as both the component and the player listener interfaces. When any of them are called it propagates them to all currently registered listeners.

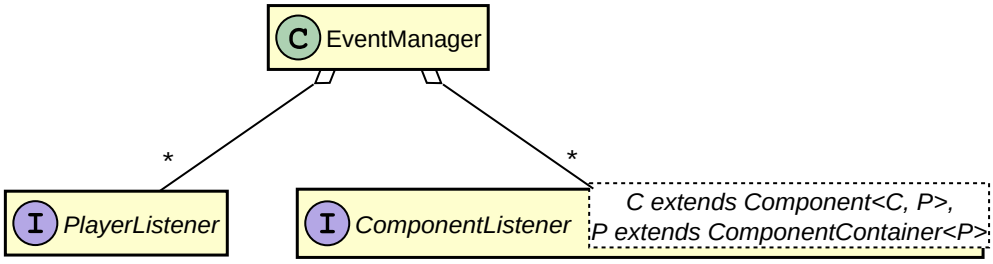


Figure 7.4: The event manager class

7.2.2 Game Data Classes

When using the game engine, it is necessary to create a world in which the game can take place. This is done using the `GameData` class. It serves as root of the model used by the game, as described in subsection 7.1.1. Every instance of the `GameData` class has its own `EventManager`. It also owns exactly one `Player` and one `World` object (see Figure 7.5). To allow the current state of a game to be saved, it implements the `Parcelable` interface.

The `Player` object represents the human player in a game (see Listing 7.5). It is only created by the `GameData` class and every game has its own instance. The player object currently only manages the orientation of the player in the world. It is expressed in radians in the range $[-\pi, \pi)$, with the the angle increasing when the player rotates in counter-clockwise direction. This angle can be queried using the `getOrientation` method and manipulated using the `setOrientation` method. The player object keeps a reference to the event manager used by the game it belongs to. Whenever the orientation of the player changes, the event manager is notified. It then calls all registered `PlayerListener` implementations (see Listing 7.6). To force an update of

```
1 public final class Player {
2     Player(EventManager eventManager);
3     public void postTo(PlayerListener listener);
4     public float getOrientation();
5     @Nonnull
6     public Player setOrientation(float orientation);
7     /* Parceling and copying methods omitted */
8 }
```

Listing 7.5: `Player` class

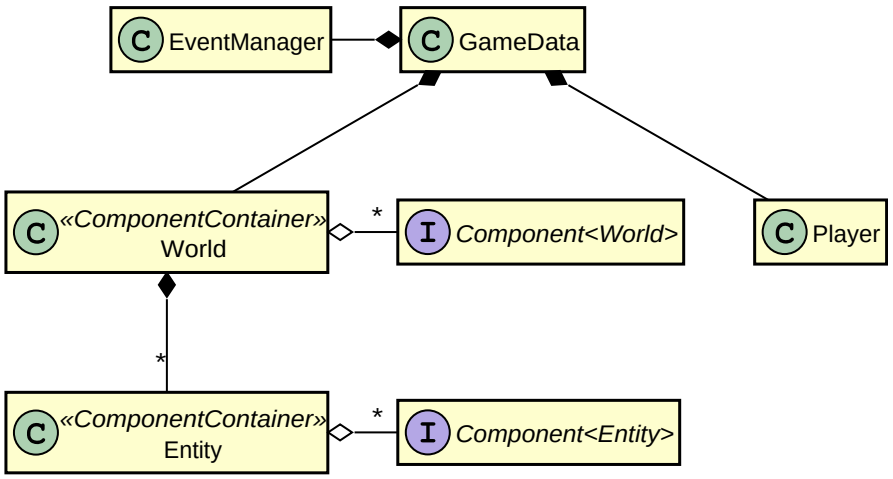


Figure 7.5: Game data classes

such a listener, the `postTo` method of the `Player` class can be used. It updates only the listener supplied as argument. This is useful when a new listener is added to the game, but it should not have to wait for an update until the player is actually modified the next time. While this class does not implement the `Parcelable` methods, it still can be stored in an Android `Parcel` using its own parceling methods. This is necessary to allow the `GameData` class to implement `Parcelable` correctly.

```
1 public interface PlayerListener {
2     void onPlayerUpdate(Player player);
3 }
```

Listing 7.6: `PlayerListener` interface

The `World` class represents the game world surrounding the player (see Listing 7.7). Like the `Player` class, it is created by the `GameData` object it belongs to and each game has its own instance. The `World` class extends the abstract component container class. This makes it possible to add and remove Components from the game world itself, using the methods described in subsection 7.2.1. The world object also serves as container for the entities used in this instance of the game. An entity is created by calling the `createEntity` factory method. The newly created entity object returned by this method is automatically part of the world that spawned it. It can be removed by using the `removeEntity` method. The `clearEntities` method removes all entities that are currently part of the game world. To get a list of all entities in a specific

world instance, the `getEntities` method can be used. This method returns an immutable collection, it can not be modifies and any changes to the original world object are not reflected by it. This is also the case for the `getEntitiesWithComponent` method, but it returns only entities that have a component of the specific type provided as argument. Finally, the class provides a method that is a shortcut for calling the `postComponentCreatedTo` method on all entities currently part of the world. This class can store both its components as well as its entities in an Android parcel. The methods to do so are only used by the game data object the world belongs to and are not declared public. Access to the underlying collection is synchronized, this class can therefore be safely accessed by multiple threads in parallel.

```
1  public final class World extends ComponentContainer<World> {
2      /* Generic type parameter declarations for methods
3         are omitted, where applicable C is defined as
4         C extends Component<C, Entity>. */
5      World(EventManager eventManager);
6      @Nonnull
7      public Collection<Entity> getEntities();
8      @Nonnull
9      public Collection<Entity> getEntitiesWithComponent(
10             Class<C> type);
11     @Nonnull
12     public Entity createEntity();
13     public void removeEntity(Entity entity);
14     public void clearEntities();
15     public void postEntityComponentsCreatedTo(
16             ComponentListener<C, Entity> listener);
17     /* ComponentContainer implementation omitted */
18     /* Parceling and copying methods omitted */
19 }
```

Listing 7.7: `World` class

Every game world can have an arbitrary amount of entities. They are implemented in the `Entity` class (see Listing 7.8). To create an instance of this class, the factory method of the world object should be used. Because of this, the constructor for the class is declared package-private. Like the game world itself, each entity is also a component container. In addition, every entity has a specific position in the world, which can be queried by using the `getPosition` method. This position is not static and can be modified during the game using the `setPosition` method. When this is done all current components of the entity are notified that their parent was updated. If they are

using the default implementation defined in the abstract base component class, this will notify any listeners of this component type. Since the listeners also get the parent of the component they were notified by as parameter, they can then read the new position and update any views accordingly. Like the game world class, this class can store its current state, consisting of its components and its position, in a parcel.

```
1  public final class Entity
2          extends ComponentContainer<Entity> {
3      Entity(EventManager eventManager);
4      @Nonnull
5      public PolarPosition getPosition();
6      @Nonnull
7      public Entity setPosition(PolarPosition position);
8      /* ComponentContainer implementation omitted */
9      /* Parceling and copying methods omitted */
10 }
```

Listing 7.8: `Entity` class

7.2.3 Utility Classes

The game engine, like the audio API, uses polar coordinates to describe positions. To define such a position, the package contains the `PolarPosition` class (see Listing 7.9). This class is final and immutable [3]. Both the azimuth and the distance field are public, since they are final there is no danger of them being changed. The class also implements the common object methods `toString`, `hashCode` and `equals`. But since the `equals` method is implemented by comparing the azimuth and distance floating point values for equality, it is of limited usefulness. The class also implements the Android `Parcelable` interface, allowing it to be stored to and restored from a `Parcel`. Since the class is immutable, its usage is inherently thread-safe. The static field called `ORIGIN` is a `PolarPosition` instance pointing at the origin of the coordinate system. Both its azimuth and its distance is set to zero.

All three parts of the engine, namely the graphics, the audio and the input system, require some configuration. This is done by supplying a configuration object to them on initialization. The actual interfaces of these objects are defined for each sub-system independently in their respective package. But since they all have some parameters in common, the `EngineContext` interface (see Listing 7.10) serves as a common base interface. The configuration object implementing this interface has to provide two

```java
@Immutable
public final class PolarPosition implements Parcelable {
    public static final PolarPosition ORIGIN;
    public final float azimuth;
    public final float distance;
    public PolarPosition(float azimuth, float distance);
    /* Common object methods omitted */
    /* Parcelable implementation omitted */
}
```

Listing 7.9: `PolarPosition` class

methods. One is called `getGameData`. It should return the root game data object that defines the game the sub-system will be attached to. For views this defines which game is rendered and for input systems which game is controlled. The other method is called `getApplicationContext`. It provides access to an Android application context [19]. This context can be used to load assets, to create views or to access sensor data. Since all sub-systems need at least access to these two methods, all configuration objects inherit this interface.

```java
public interface EngineContext {
    @Nonnull
    GameData getGameData();
    @Nonnull
    Context getApplicationContext();
}
```

Listing 7.10: `EngineContext` interface

Both the graphics as well as the audio rendering systems store data in buffers. To be usable this data has to be uncompressed. Both uncompressed audio data as well as textures can take up considerable amounts of memory. To reduce the impact on the memory requirements of the application, buffers can be shared. If there are for example several objects using the same texture in the world, they all can share the same texture buffer. At the same time unused buffers should be released to free the memory space they occupy. To help implement such a buffer, the main engine package contains the `AssetBufferReference` class (see Listing 7.11). This abstract class represents a reference counted buffer that is re-usable by different objects. It is initialized with the name of the asset it represents, which can later be queries using the `getAssetName` method. To increment or decrement the internal reference counter, the `acquire` or the

`release` method is used. Each call to `acquire` increases the counter by one, while `release` decreases it. When `acquire` is called and the current reference count is zero, the abstract `createBuffer` method is called. Here the implementing class can create a buffer containing the data identified by the asset name. Providing access to the buffer is also the responsibility of the implementing class. When the last reference to a buffer is release, that is when the reference count reaches zero again, the abstract `deleteBuffer` method is called. Here the implementing class can delete the buffer and free its memory. The reference counting methods are synchronized, the class can safely be used by multiple threads.

```
1 public abstract class AssetBufferReference {
2     protected AssetBufferReference(String assetName);
3     @Nonnull
4     protected final String getAssetName();
5     public final synchronized void acquire();
6     public final synchronized void release();
7     protected abstract void createBuffer();
8     protected abstract void deleteBuffer();
9 }
```

Listing 7.11: `AssetBufferReference` class

7.3 Graphics

The graphics system of the game engine is used to display the game world to the user. This is done using OpenGL ES 2.0, which is part of the Android API since version 8 [31]. On Android, all OpenGL rendering happens on a separate thread, decoupled from the main UI thread. To access this thread, the Android `Renderer` interface provides callbacks that are called whenever the target surface is changed or it is time to render a frame. One challenge faced when using OpenGL on Android is that the OpenGL context can be destroyed by the system at any time, for example when the device goes to sleep. When this happens, all resources are deleted and have to be recreated when a new context becomes available [24].

To setup the rendering system, the user must provide a configuration object. This object has to implement the `GraphicsContext` interface (see Listing 7.12). Like all configuration object interface definitions used by the game engine, it extends the `EngineContext` interface. This allows accessing the root game data object as well as

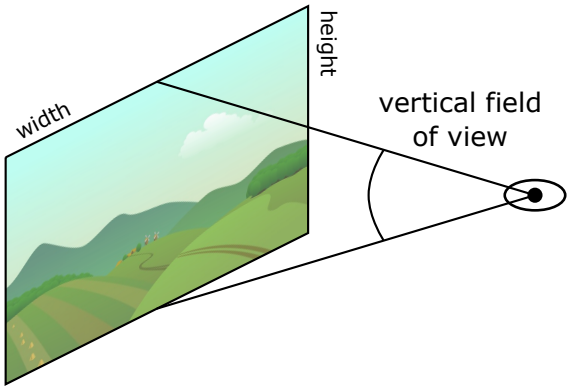


Figure 7.6: The field of view when rendering the game world

the Android application context. Additionally, this interface defines two more methods. The `getVerticalFov` method set the vertical field of view used when rendering the game world. This is the angle in radians covered by the screen in the vertical direction, as illustrated in Figure 7.6. The second method is called `getBitmapAssetCache` and has to return an instance of a `LoadingCache` as defined by the Guava library. This is a class that can load and optionally cache assets given their name. In this case the assets are textures which have to be provided as uncompressed bitmaps. Making the implementation of the cache the responsibility of the user results in more flexibility when using the game engine. The user can then decide how much memory, if any, to reserve for the texture cache and how the textures are loaded.

```
1 public interface GraphicsContext extends EngineContext {
2     float getVerticalFov();
3     @Nonnull
4     LoadingCache<String, Bitmap> getBitmapAssetCache();
5 }
```

Listing 7.12: `GraphicsContext` interface

The root class of the graphics system is the `OpenGLRenderer` class (see Listing 7.13). To create an instance of this class an object implementing the `GraphicsContext` interface has to be provided. Internally, this class then creates an OpenGL surface and registers its private `Renderer` implementation. After construction is finished, the `getView` method can be used to get a reference to this surface. It can then be added to a layout and displayed. The `stop` and `start` methods should be called when the view is no longer visible or when it becomes visible again. This allows the underlying surface

```
public final class OpenGLRenderer {
    public OpenGLRenderer(GraphicsContext context);
    public synchronized void start();
    public synchronized void stop();
    @Nonnull
    public View getView();
    private final class Renderer
            implements GLSurfaceView.Renderer {
        /* GLSurfaceView.Renderer implementation omitted */
    }
}
```

Listing 7.13: `OpenGLRenderer` class

to free resources while rendering is not required. The `Renderer` implementation defined in this class does not do any actual rendering. Instead, it delegates the task to different sub-renderers, which implement the `OpenGLSubRenderer` interface (see Listing 7.14). These sub-renderers get called when the surface is recreated or when a frame is drawn, using the `onSurfaceCreated` and `onDrawFrame` methods. Both of these methods receive the current rendering parameters as argument. Since the game world is rendered form the view of the player, the `OpenGLRenderer` class contains a `PlayerListener` implementation, allowing it to react to changes in the player orientation. When the renderer is not needed anymore, for example because the activity containing it is destroyed, the `stop` method has to be called. This will unregister the instance from the event manager which otherwise would hold a reference to the object and prevent its garbage collection.

```
interface OpenGLSubRenderer {
    void onSurfaceCreated(RenderParameters renderParameters);
    void onDrawFrame(RenderParameters renderParameters);
    void start();
    void stop();
}
```

Listing 7.14: `OpenGLSubRenderer` interface

7.3.1 Rendering Process

There are currently two sub-renderers implemented. The first one renders the panorama used as backdrop of the game world. Its implementation will be described in subsec-

tion 7.3.2. The other one is the entity renderer, which draws all entities with a visual component onto the screen. This renderer will be described in subsection 7.3.3. In order for these entities to be visible, they have to be drawn above the panorama. One way this can be accomplished by using a depth test. Here, the depth of every drawn pixel is stored into a buffer. When another pixel should be drawn at the same location, its depth can be compared against the one already stored in the buffer. If the new pixel would be in front of the already drawn one, it overwrites it. Otherwise it is discarded. This depth test is already part of OpenGL ES 2.0. In this game engine, transparency is implemented using alpha blending. That means each pixel can have an alpha value in addition to the usual red, green and blue color values. This value represents the opacity of the pixel, with zero being transparent and one being opaque. When using alpha blending, each object is rendered on top of the scene behind it. For the result to be correct, this requires sorting the objects in the scene and rendering them back to front rtrendering. For sorted objects that are drawn back to front, a depth test is unnecessary since it can not discard any pixels. Since in this engine all object except the panorama are rendered using alpha blending, requiring them to be sorted, no depth test is used at all. Instead the panorama backdrop is rendered first, followed by rendering all entities in back to front order. The ordering happens relative to the center of the scene, where the player is located.

On Android, accessing the OpenGL ES 2.0 API is done using the `GLES20` class. It provides static methods that mirror many of the native OpenGL functions. Since the OpenGL API is not object oriented, neither are these functions. They can be called anywhere and modify the global state, although they can only be used on the thread the OpenGL context belongs to. To make some common tasks used by the engine simpler and to avoid code duplication, the graphics package includes several utility classes.

During the rendering process, OpenGL uses shader programs to define how something is drawn. These programs define the positions and the colors of each pixel that is rendered to the screen. For the purposes of this game engine, only one shader program is necessary. It is implemented in the `BlendingShaderProgram` class (see Listing 7.15). The methods of this class can only be used from the OpenGL thread since they directly use the functions provided by the `GLES20` object. When initialized, this class compiles both a vertex and a fragment shader and links them into a shader program. This program is then ready to be executed by the graphics processing unit (GPU). Should the compilation fail, the class will throw a runtime exception and print the cause description to the Android error log. Otherwise the program can then be

used by calling the `use` method. After this method is called, the program stays active until a different shader program is used. All other methods of this class depend on the program represented by their object being the one currently in use. The shader program compiled by this class supports the use of a texture as well as both alpha and color blending. Before a texture buffer can be bound, a texture has to be activated. This is done using the `activateTexture` method. The blending factors can be set using the `setBlendingFactors` method. To determine both the perspective as well as the camera position and orientation a matrix is used. This matrix is the result of multiplying three other matrices. The first is the model matrix. It determines where the currently rendered object is positioned in the world. This also includes any rotation or scaling done to the object. The second is the view matrix. It provides the position and orientation of the camera in the scene. Finally there is the projection matrix. It determines the projection used when rendering, usually either perspective or orthogonal. This matrix also determines the aspect ratio and, in case of a perspective projection, the field of view the scene is rendered in. The combined matrix is then supplied to the shader object using the `setMvpMatrix` method. The array given as parameter is compatible with the arrays used by the methods defined in the `Matrix` class that is part of the Android API. The `Matrix` class also defines several methods that calculate or manipulate any of the matrix types described above.

```
1  final class BlendingShaderProgram
2          implements TextureShaderProgram {
3      public void initialize();
4      public void delete();
5      public void use();
6      public void activateTexture();
7      public void setBlendingFactors(float r, float g,
8                                     float b, float a);
9      public void setMvpMatrix(float[] matrix);
10     /* TextureShaderProgram implementation omitted */
11 }
```

Listing 7.15: `BlendingShaderProgram` class

Currently all object rendered by the engine are drawn as textured squares. When rendering in OpenGL, all objects are build using triangles. They are defined by their vertices. A quad can be built by combining two triangles that share two vertices with each other. This is illustrated in Figure 7.7. Also shown are the texture coordinates associated with each vertex. When rendered with a texture, these coordinates are used

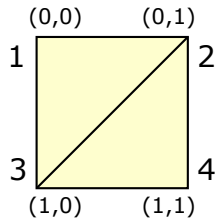


Figure 7.7: A quad build by using two triangles

to map the texture to the polygon. In this case the texture is mapped to simply cover the entire square with the edges of the texture lining up with the edges of the quad.

Since a textures quad is used for every object currently supported by the engine, the graphics package includes a utility class to create such a square. It is called `VertexQuad` (see Listing 7.16). As with the shader program class, all methods of this class must only be called from the OpenGL thread. The vertex quad is initialized with an instance of a shader program. After the call to the `initialize` method, the quad can only be used when this shader is active or until it is re-initialized with a different shader. During initialization, an OpenGL array buffer is created an filled with the vertices that create a single quad. This quad is square with an edge length of exactly one unit. It is created with its center at the origin. Both its position and size can be modified while rendering by using a model matrix. When it is time to draw the textured quad, the `bind` method has to be called first. This binds the underlying vertex array buffer in the current OpenGL context. It also points the position and texture coordinate attributes of the shader to the correct locations inside the buffer. After that, the quad can be drawn using the `draw` method. This method can be called multiple times. The texture that is used is defined by the shader used to draw the quad. When done drawing, the `release` method should be called.

```
1 final class VertexQuad {
2     public void initialize(
3             TextureShaderProgram shaderProgram);
4     public void delete();
5     public void bind();
6     public void draw();
7     public void release();
8 }
```

Listing 7.16: `VertexQuad` class

Before a texture can be used by a shader, it has to be stored in an OpenGL texture buffer. Such a buffer can be used by multiple objects that share the same texture and should be deleted once the texture is no longer needed. The `TextureBufferManager` class (see Listing 7.17) is designed to facilitate such a reuse of texture buffers. It is initialized with the OpenGL surface it manages the texture buffers for as well as a bitmap cache used to load texture data. An instance of this class is created by the OpenGL renderer used by this engine, which is then passed to all the sub-renderers. It uses the bitmap loading cache defined in the graphics context. If a sub-renderer wants to use a texture, it can call the `get` method with the name of the image asset it wants to use as a texture. This method will then return an object implementing the `TextureBuffer` interface. Internally, the class first checks if a texture buffer for the requested asset already exists. If it does not, a new texture buffer reference is created for this asset and added to the map. Since it extends the asset buffer reference class it is reference counted and the asset will be loaded and stored in a texture buffer the first time it is accessed. This buffer will also be deleted as soon as there are no more references to it. The texture buffer reference is then wrapped in a texture buffer proxy, which implements the `TextureBuffer` interface. This class is simply an adapter for the actual texture buffer reference [9]. The `get` method can be safely called on any thread. The texture buffer manager also defines a method called `invalidate`. It should be called if the OpenGL context is recreated after it was deleted. Since the previous texture buffers are not longer valid, this method recreates all of them. After this is done all buffer references created by this manager are valid again. Since this method must be called on the OpenGL thread, preventing any other renderers from running, the users of this manager will not be affected and can simply continue using their texture buffer objects.

```
1  final class TextureBufferManager {
2      TextureBufferManager(GLSurfaceView glSurfaceView,
3              LoadingCache<String, Bitmap> bitmapCache);
4      @Nonnull
5      TextureBuffer get(String textureName);
6      void invalidate();
7      private final class TextureBufferReference
8              extends AssetBufferReference;
9      private static final class TextureBufferProxy
10             implements TextureBuffer;
11 }
```

Listing 7.17: `TextureBufferManager` class

7.3.2 Panorama Renderer

The panorama renderer implements the sub-renderer interface and draws a panorama image as backdrop for the game world. Because such a panorama picture has to be very wide to show a full 360 degrees image its width might be larger than the maximum supported texture size supported by the Android device. The panorama texture is therefore split into an arbitrary number of identically sized square textures. Since the textures must, when arranged side by side, cover exactly 360 degrees the renderer can calculate the horizontal angle covered by one texture form the number of parts. Because the textures are square and the renderer does not support stretching the textures in one dimension only, the angle covered by each texture is identical along the horizontal and the vertical axis. It is therefore important to choose an appropriate number of pictures to split the panorama into. The application uses by default a vertical field of view of 40 degrees. In order to have the panorama fit the screen exactly with this field of view, the panorama has to be split into $360°/40° = 9$ images. The names of these images is defined in the panorama component, of which an instance can be added to the game world. This component also allows setting the blending factors the panorama is rendered with.

To render the panorama, the texture quad and shader program classes introduced in the previous section are used. After they are set up, the renderer determines how many of the square texture the panorama image is split into have to be drawn to fill the screen. If the left side of the screen does not fall exactly on the cut between two textures, this offset has to be factored in when determining the number of quads to draw. They are then drawn from left to right, with the vertex quad being reused for each individual texture. Each time the quad is draw, the next texture has to be selected. The panorama is drawn using orthogonal projection. This means the distance of the backdrop has no influence on the final appearance of the panorama on the screen. Because of the alpha blending technique used by this game engine, the backdrop should be drawn first. This allows all other objects to cover it while fully supporting transparency.

7.3.3 Entity Renderer

Entities in the game world, like for example the target or decorative objects, are rendered using a technique called screen-aligned billboarding [1]. This means they are rendered as a textured quad that always faces the camera. The texture is determined by the user

of the game engine when adding the billboard rendering component to the entity. The billboard renderer then draws this texture using the vertex quad utility class. To make the quad face the screen, the top left three by three sub-matrix of the model-view-matrix is replaced with the identity matrix. This part of the model-view-matrix is responsible for the scaling and rotation of the drawn object. While the removal of the rotation component is necessary to draw the entity as a billboard, the loss of scaling information is not necessarily desirable. The scaling transformations can therefore be reapplied after this modification. After the projection matrix is applied, the rendered entity will be at its correct position on the screen while still facing the camera. Unlike the panorama renderer, the entity billboard renderer uses the perspective projection matrix. This means objects that are farer away will be rendered smaller.

The entity billboard renderer also uses the shader program class introduced in subsection 7.3.1, including support for alpha blending. The blending factors used when rendering can be defined at two places. The first is in a world component. Since it belongs to the world object that is unique for each game, only one instance of this component can be active for a single game. The blending factors set in this component are applied to all entities rendered with this renderer, as well as to the background panorama. Another set of blending factors can be defined for each instance of the entity component. These factors are multiplied with the global ones defined by the world component before being applied to the entity billboard they belong to. The entity component also allows the modification of several other settings that affect how the entity is rendered. One is the scaling factor that is applied to the vertex quad after it was made into a billboard. An offset along the Y axis can also be set. Since the entity position does not include a Y component, this property is necessary to affect the vertical drawing position of the final billboard. Finally, the texture can be flipped along the X axis.

Before the entities can be drawn they have to be sorted, with the entities that are the furthest away being the ones that are drawn first. This is necessary when using alpha blending so billboards in the front can have transparent pixels that show the billboards behind them. Fixing the drawing order also prevents the billboards from intersecting each other since the ones that are drawn later will always appear in front of the ones drawn before. Because the positions of the entities is not static, this sorting has to be redone every time an entity moves closer or farther away from the player. To reduce the load on the processor this is only done once it is actually time to draw and even then only if there was a position change.

7.4 Audio

The audio system of the game engine uses the API introduced in section 5.2 to create and manage the audio sources defined by the game world. This system, like the graphics system, is set up using a configuration object. This object has to implement the `AudioContext` interface which extends the `EngineContext` interface (see Listing 7.18). One of the methods the `AudioContext` interface adds is called `getAudioBackendFactory`. This method has to return the audio back-end factory that will be used to create the audio back-end instance used by the game engine. Using a factory instead of simply passing an instance of the audio back-end allows the engine to destroy and later recreate the audio back-end. This might for example be necessary when the application looses the audio focus. The other method defined by the interface is called `getAudioDataAssetCache`. Like its counterpart in the `GraphicsContext` interface this method has to return a `LoadingCache` that will be used to retrieve the audio data. This allows the user to create a cache and reduce loading times if the data is used again in the future.

```
1 public interface AudioContext extends EngineContext {
2     @Nonnull
3     AudioBackend.Factory getAudioBackendFactory();
4     @Nonnull
5     LoadingCache<String, AudioData> getAudioDataAssetCache();
6 }
```

Listing 7.18: `AudioContext` interface

To use the audio system, an instance of the `AudioPlayer` class has to be created (see Listing 7.19). Its constructor takes an object implementing the `AudioContext` interface as its only argument. Like the graphics system, the player has to be started using the `start` method before it becomes active. Once it is not longer needed, the `stop` method stops the audio playback. To temporarily silence the audio, the `mute` method can be used. Alternatively, the `duck` method lowers the output volume by 80 percent. These methods are useful to allow notification sounds to be heard without completely stopping the audio player. The `resume` method undoes the volume reduction caused by ducking or muting.

When playing audio on a mobile device there are situations where an application should stop its audio output without requiring any user input. For example when the mobile

```
public final class AudioPlayer {
    public AudioPlayer(AudioContext context);
    public synchronized void start();
    public synchronized void stop();
    public void mute();
    public void duck();
    public void resume();
}
```

Listing 7.19: `AudioPlayer` class

phone is receiving a call, the running application should automatically lower or even mute its output volume to allow the ringing tone to be heard clearly. On Android devices this is done using an audio focus system [28]. Each application that wants to playback audio should first request one of the two types of audio focus. The first type is a transient focus. This can for example be used to play a notification sound and the application should keep the audio focus only for a short time. The other type is a permanent focus, which can for example be used to play music or for a game. Once an application has the audio focus, it will receive a notification when it looses the focus again, either temporarily or permanently.

The `AudioSystem` class is a wrapper for the `AudioPlayer` that requests the audio focus and handles any changes in the focus state of the application (see Listing 7.20). It is initialized with an implementation of the `AudioContext` class which is saved to be used when creating a new audio player instance. When the `start` method is called, the class requests a permanent audio focus. Only if the request is granted a new instance of the audio player class is created and started. It also registers an audio focus listener. If the request is declined, the method simply returns `false`. Once the system is not longer needed, the `stop` method abandons the audio focus and disposes of the wrapped audio player instance. It is safe to call this method even if a previous call to `start` returned `false`. The audio focus listener registered by the class handles a temporary focus loss by either muting or, if allowed by the operating system, ducking the volume of the audio player. A permanent focus loss is handled like a call to the `stop` method. The current state of the audio focus can be queried using the `hasFocus` method. The class also allows registering its own version of a simplified focus listener, which can be set or cleared using the `setFocusListener` method. Unlike the Android version, this listener is only notified when the focus is lost permanently. This allows the user to react to such an event. Once lost, a new request to regain the audio focus can

```
public final class AudioSystem {
    public interface FocusListener {
        void onFocusLost();
    }
    public AudioSystem(AudioContext context);
    public synchronized boolean start();
    public synchronized void stop();
    public synchronized boolean hasFocus();
    public synchronized void setFocusListener(
            @Nullable FocusListener focusListener);
}
```

Listing 7.20: `AudioSystem` class

be triggered by calling `start` again. If the focus is abandoned because of a call to `stop` by the user, the listener is not called.

The `AudioBufferManager` class (see Listing 7.21), like its counterpart in the graphics system, facilitates the reuse of existing audio buffers and takes care of their disposal once they aren't used any longer. It is initialized with the audio back-end that is used to create the buffers as well as a loading cache instance that is used to load the audio data. This class creates two types of audio buffers. The first type simply contains the audio data read from the cache and is implemented with reference counting using the `AssetBufferReference` class. A call to the `get` method, either with only a single argument or with the second argument as `null`, will create such a buffer if it not already exists and then return an adapter object referencing that buffer. This adapter intercepts calls to the `dispose` method of the buffer and instead of deleting the buffer only decrements the reference count. Only when the reference count reaches zero, meaning the buffer is not used anymore, the actual underlying audio buffer is deleted. The other type of buffer is created by passing an audio filter as second argument to the `get` method. This will create an audio buffer which content is modified using the supplied audio filter instance. While these types of buffers are not cached in the manager, they are otherwise implemented in the same way as regular audio buffers.

To create an audio source in the game world, it has to be added to an entity as a component. Currently only looping audio sources are supported. Once such a component is added to an entity, it will play the audio data stored in the backing buffer until the component is removed again. The audio source has the same position as the entity it belongs to, allowing it to move in the game world. Each instance of this

```
final class AudioBufferManager {
    AudioBufferManager(AudioBackend audioBackend,
            LoadingCache<String, AudioData> audioDataCache);
    @Nonnull
    AudioBuffer get(String audioName);
    @Nonnull
    AudioBuffer get(String audioName,
                    @Nullable AudioFilter audioFilter);
    private final class AudioBufferReference
            extends AssetBufferReference;
    private static final class AudioBufferProxy
            implements AudioBuffer;
}
```

Listing 7.21: `AudioBufferManager` class

component class is initialized with the audio data asset name and optionally the audio filter to apply to the audio data. It is also possible to set an individual gain factor that is only applied to the audio source belonging to this component. Unlike the audio data this gain is dynamic and can be changed at any time. It can for example be used to lower the volume of an individual audio source temporarily. This component is also used to create ambient audio sources, that is audio sources that have no positional component. Such a source is created by adding the audio component to an entity that is places at the center of the game world, which is also the position of the player.

7.5 Input

The input package of the game engine module contains classes that provide different methods to control the player orientation in the game world. Like the other systems of the game engine, the input system is set up using a configuration object. This object has to implement the `InputContext` interface (see Listing 7.22). This interface defines a method that returns the vertical field of view used when rendering the game. The method has the same signature as the method of the same name in the `GraphicsContext` interface. Knowing the field of view is necessary when touchscreen controls are used. The other method returns the factory that is used to create the rotation sensor used. If no sensor input should be used, this method can return the `NullSensor` as described in section 5.3.

```
1 public interface InputContext extends EngineContext {
2     float getVerticalFov();
3     @Nonnull
4     RotationSensor.Factory getRotationSensorFactory();
5 }
```

Listing 7.22: `InputContext` interface

An intuitive way to control the player orientation is by using a rotation sensor to follow the direction the device is pointing to. This can be done using the `SensorInput` class (see Listing 7.23). It is initialized with an instance of the input configuration object that defines the sensor to use. Once the `start` method is called, it maps the rotation read from the sensor to the orientation of the player in the game world until `stop` is called. The first value received from the rotation sensor is used to calibrate the orientation. This prevents the player orientation from suddenly jumping once the sensor is activated. The input can be temporarily disabled by using the `setEnableRotation` method. Finally, the `isNullSensor` method can be used to check if a real sensor is used. If this method returns `true`, the sensor input class will never receive a sensor reading and another form of input should be used.

```
1 public final class SensorInput {
2     public SensorInput(InputContext context);
3     public synchronized void start();
4     public synchronized void stop();
5     public void setEnableRotation(boolean enableRotation);
6     public boolean isNullSensor();
7 }
```

Listing 7.23: `SensorInput` class

Another way to control the player orientation is by using the touchscreen of the device. This is a useful fallback input method that can be used when no appropriate sensors are present or the sensors are unreliable. When the user touches the screen, the input is usually received by the view element currently visible at that position of the screen. While some view classes allow the user to register a touch listener. this is not the case for the `GLSurfaceView` class used to render the game [34]. To still enable touchscreen controls for the game, the `TouchViewInput` class can be used (see Listing 7.24). This class is initialized with the configuration object and the view that defines the area of the screen where touch inputs should be received. Internally the class puts this view

into a view group that intercepts all touch inputs and allows the input class to handle them instead. This works independently of the actual type of the wrapped view. Once `start` is called, the intercepted horizontal scroll gestures are translated into orientation changes for the player in the game world until `stop` is called. To allow this translation to match the view of the game world, the field of view returned by the configuration class should be the same that was used to set up the game view. Like in the sensor input class the `setEnableRotation` method can be used to temporarily disable this type of input. Whenever this class is used, the view returned by the `getView` method should be used in place of the wrapped view. The returned view also supports setting a click handler that can be used as an additional input method.

```
1 public final class TouchViewInput {
2     public TouchViewInput(InputContext context,
3                           View wrappedView);
4     public synchronized void start();
5     public synchronized void stop();
6     public void setEnableRotation(boolean enableRotation);
7     @Nonnull
8     public View getView();
9 }
```

Listing 7.24: `TouchViewInput` class

8

Application

To demonstrate the game concept a demo application was created. It uses the game engine introduced in chapter 7 and, by extension, the framework described in chapter 5. As implementation of the Audio API it uses the OpenAL back-end outlined in chapter 6. The code for the application is split into two modules. The common application module is implemented as a library. It contains classes that would be useful when designing or embedding a similar game into another application. The actual demo application, including its assets, is located in a separate module. Both modules will be described in this chapter.

8.1 Common Application Classes

The common application library mainly contains classes that facilitate the integration of the game engine in a typical game application. One of these classes is a rotation sensor manager. Since the sensor input system of the game engine expects to be provided

with a rotation sensor factory, it falls to the application to choose the appropriate sensor implementation. The sensor manager can take care of finding the best available sensor supported by the device. To do this it can either try to find any working sensor or it can only return a sensor if it is likely that it uses a gyroscope. If no such sensor can be found, it returns the null sensor. The demo application uses the rotation sensor manager to allow the user to choose if they want to use a sensor and, provided they do, if sensors must utilize the gyroscope to be considered valid.

The module also contains factory classes for the bitmap and audio data cache required by the game engine. Both provide loaders that can read their respective data files from the assets embedded in the application. While the bitmap cache loader used the decoder provided by the Android system, the audio data cache loader uses the WAVE and Ogg-Vorbis file decoders implemented as part of the audio API. Also implemented by both caches is the weigher function that allows the creation of a cache with a maximum memory footprint. This enables the application to explicitly allocate a part of its available memory to either of these caches.

In the game engine module the different systems, namely graphics, audio and input, are implemented independently of each other. While this approach increases flexibility, it also increases the amount of code needed to setup a game. The common application library therefore includes a class that simplifies the setup for the common case where exactly one OpenGL renderer, one audio output system and one input method is required. The class is implemented as a fragment, which is a reusable user interface component [23]. This allows it to be embedded in different Android activities. The fragment expects that its parent activity implements a special interface that is used for communication. This way the activity can provide the game data definition, the caches needed to access the assets as well as the used rotation sensor manager. The input method used by the fragment is determined by the type of sensor that is returned by this sensor manager. If it is the null sensor, the touchscreen is used to control the game. Otherwise the returned sensor is used. While the different systems are automatically created by the fragment, they still have to be started and stopped by the activity. To do this the class provides start and stop methods that do this for all systems managed by the fragment. These methods are synchronized which allows them to be called in different threads. This way the main thread is not blocked while the graphics and audio system load or decode assets. The fragment sets the vertical field of view used by the OpenGL renderer to 40 degrees.

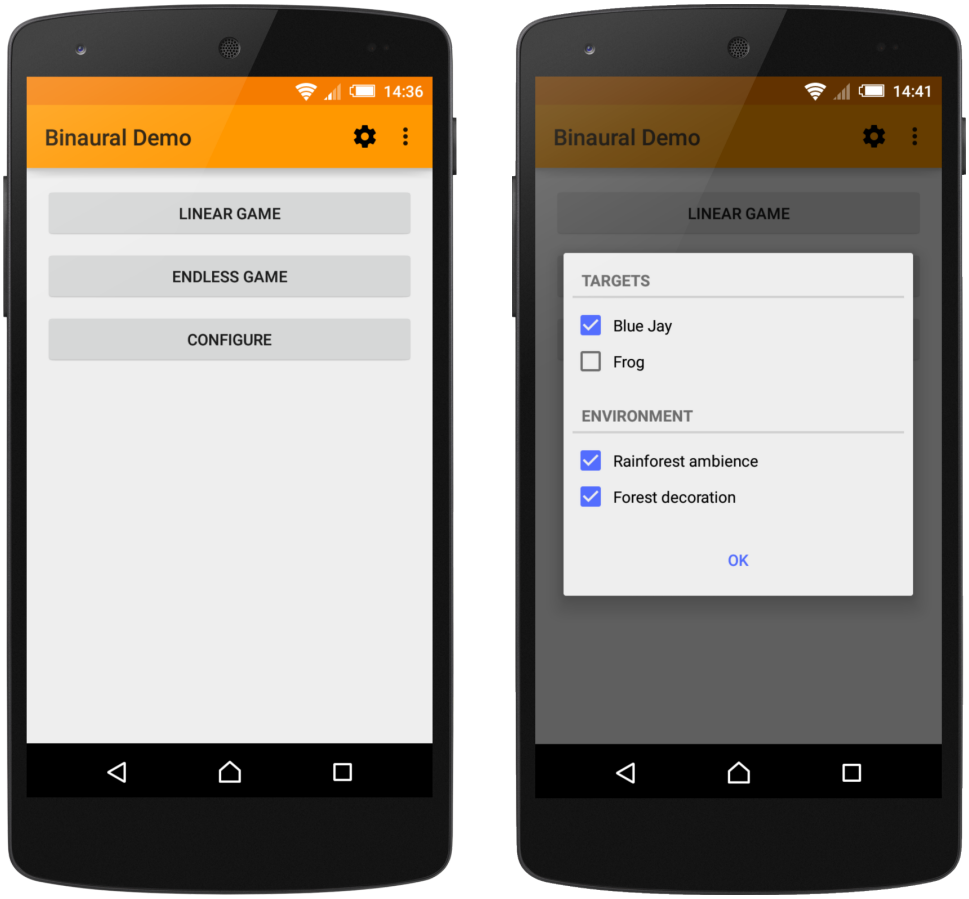
Also included in the module is a fragment that manages a single thread that can be used to load data in the background. While the thread is working on a task, a dialog containing a loading indicator is displayed on the activity managing the fragment. It also allows the execution of code on the main thread after a task is done, similarly to the `AsyncTask` class defined in the Android API.

8.2 Demo Application

To demonstrate the capabilities of the different frameworks implemented as part of this thesis, a demo application was created. It implements a game where the user has to locate a target purely based on the sounds it emits. This means that when the game is started, the target is not visible. Instead it is only audible, with its sound transformed using HRTFs to appear to come from a specific direction. The user can hold their Android device in front of them and start rotating, like when taking a picture, to try and locate it. The game reacts to this rotation and adjusts the audio output. When the user believes the virtual target to be in front of their device, they can press a button to trigger a virtual camera. Instead of using sensors to detect the rotation, the touchscreen can also be used to play the game. In that case the user can simply scroll left or right to rotate the game world. When the virtual camera is triggered, the target is made visible. The game also shows the user how far off the mark they hit as well as the time they took to find the target.

The game can be customized using different settings. Besides the choice between sensors and the touchscreen as primary input device, the user can choose if they want to see a darkened version of the game backdrop while searching or if only a blank screen should be shown. It is also possible to lock the orientation when the result is shown. Otherwise the user is allowed to further look around, which can be useful if they missed the target and want to look where it would have been.

When first started, the main menu of the application, shown in Figure 8.1a, is displayed. Here a game can be started by choosing one of the two currently implemented game modes. The first mode starts a game where the user has to search each enabled target exactly once, then the game is over. The other type of game does not have a predetermined end. Instead, after every round a new target is added and the previous one is removed. This way the game will only end when the user quits it by pressing the back button. These games can also be configured using the dialog shown in Figure 8.1b.



(a) Main menu screen (b) Game configuration dialog

Figure 8.1: Main menu of the demo application

Here the user can choose which targets should be used in the games and which are disabled as well as if ambient background audio should be played during the game. The application settings can be accessed using the action bar.

Both game modes are implemented based on a common abstract game activity class. This class handles setting the screen orientation to the format preferred by the user and initializing the rotation sensor manager. The asset caches used by the game systems are managed by the application object. This way they are global and can keep assets cached after a game is ended. If the Android system requests the application to trim its memory usage all cache entries are invalidated which makes them available for garbage collection. This happens when the system starts to run out of memory or when the application is moved into the background. The game interface is implemented in

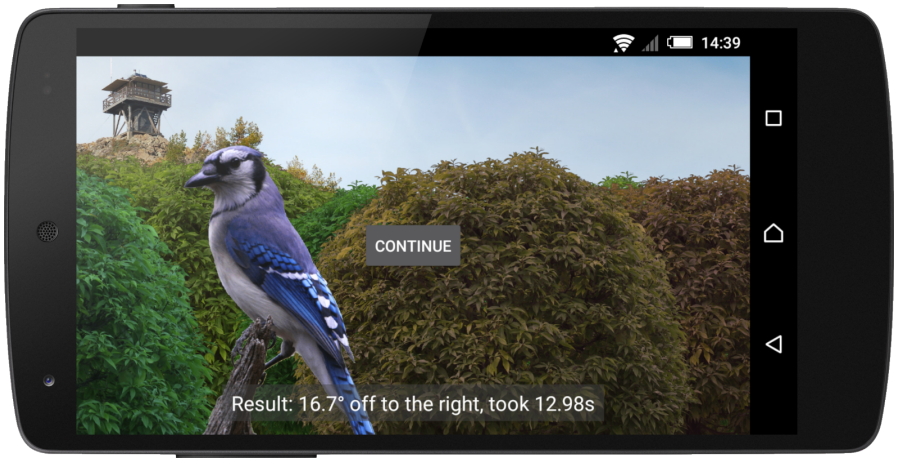


Figure 8.2: Game result screen of the demo application

a separate fragment. In the game activity an instance of this fragment is displayed on top of an instance of the game render fragment. This effectively projects the user interface elements on top of the rendered game world (see Figure 8.2). The base game activity class also handles saving the game state when the activity is destroyed and restoring it on recreation. This way a game can be continued even after the activity was interrupted. When restoring the game world, a special component is used to identify the target entities in the game world. Any call to the game engine that might result in assets being loaded is executed in a background thread, using the loading fragment described in the previous section. The game is paused while this happens and a progress dialog is shown if loading takes longer than a few milliseconds.

The common game activity also handles the setup of the game world. It creates the panorama component and, if enabled by the user, adds the decorative tree entities. If ambient background audio is enabled the activity also creates the entity necessary for its playback. The target entities are not created by this class. Instead, this task is delegated to the implementing subclasses. In the endless game variant, this is handled by creating a new target entity after the search for the previous one is finished. In the other game mode all targets are created during the game world initialization. The methods that are called when the game state changes, for example from the search state to showing the result screen, are all designed to be overridable. This flexibility makes it possible to implement these and other game modes using the abstract base game activity class.

9

Evaluation

To evaluate the quality of the audio localization when using the application a study was carried out. In this chapter the methods used to conduct the study and its results will be described. Then these results will be compared to the results achieved by two other, similar applications that were developed for different mobile operating systems.

9.1 Participants and Methods

The study had a total of 24 participants. Four of these participants were female. The average age of the subjects was 27.2, with a standard deviation of 8.6. The youngest participant was 21 and the oldest was 64 years of age. One participant did not disclose their age. Two of the subjects indicated that they were affected by tinnitus.

In addition, each participant was asked to rate their experience level with both mobile devices and videogames. This was done on a scale of one to five with one being the least and five being the most experience. The experience with mobile devices had an

average rating of 4.2 with a standard deviation of 1.1. The experience with video games had an average rating of 3.5 with a standard deviation of 1.4. Each subject also was asked what smartphone operating system (OS) they usually use. With 19 users, Android was the most popular mobile OS. Apple's iOS had 3 users and Microsoft's Windows Phone had 2 users. One participant indicated they don't usually use a smartphone and one participant uses both Android and Windows Phone regularly.

Each subject played the game in three different modes. In the first mode, the participant had to locate a bird by the sounds it was emitting. No ambient audio was used at this stage. The second mode added a second target in the form of a frog. For this round the subjects had to first find the bird, which was identical to the one used in the first mode, and then the frog. Finally, the third round added ambient audio in the form of rainforest noises. Otherwise it was identical to the second mode. The participants used headphones and were asked to repeat each mode three times. For each round the time required by the subject to find the target as well as the difference between the actual and the guessed position was recorded. The sounds used for the targets were identical to the ones available in the demo application described in the previous chapter.

9.2 Results

During the study 378 measurements were recorded by the Android application. The expected number of measurements would have been 360, meaning several participants replayed a game mode after they reached the required number of three repetitions. There are also three cases where a participant quit a game before completing it. Both can be attributed to accidental inputs while handling an unfamiliar device. If a user accidentally quit a game session prematurely, they might have decided to replay the game, resulting in the higher than expected number of individual play sessions. Since the subjects had to have full control over the smartphone while playing the game in order to use the sensor input method, monitoring for such errors proved to be difficult. For the purposes of the following analysis, all recorded measurements are taken into account.

Using R [64] a linear mixed effect model [62] of the relationships between the time each participant took to complete a game and the different game settings was created. The game mode and the target were used as fixed effects, the identification number of the participant was entered as a random effect. The analysis of variance (ANOVA)

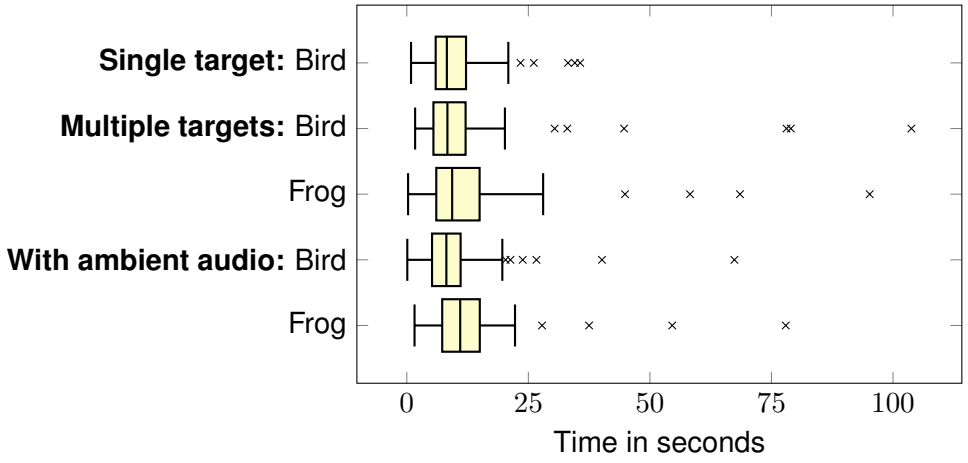


Figure 9.1: Playtime on the Android device by game mode and target

calculated for this model shows a possible significance for the game mode ($F(2, 351) = 4.20$, $p = 0.016$). The target the participant searched for is probably not significant ($F(1, 351) = 3.77$, $p = 0.053$). For further analysis the results of the measurements taken with the first game mode, which uses only a single target, are excluded. The remaining two game modes both use two targets, but only one uses ambient audio. The new model uses the game mode, the target and their interaction as fixed effects. Here neither the remaining game modes ($F(1, 276) = 3.78$, $p = 0.053$), nor the target ($F(1, 276) = 3.49$, $p = 0.063$) show convincing significance. The interaction between the game mode and the target is also not significant ($F(1, 276) = 1.00$, $p = 0.319$). A boxplot of the playtime measurements, grouped by game mode and target, is shown in Figure 9.1.

A similar analysis was done for the absolute offset angles achieved by the participants. The same fixed and random effects as in the time analysis where used. Here the ANOVA shows no significance for the game mode ($F(2, 351) = 0.37$, $p = 0.69$). The target on the other hand is significant ($F(1, 351) = 8.68$, $p = 0.003$). As before, the measurements taken using the single target game mode are removed for further analysis. This shows that the interaction between game mode and target is also not significant for the offset angle ($F(1, 276) = 0.57$, $p = 0.451$). Otherwise the target is still significant ($F(1, 276) = 8.55$, $p = 0.004$) while the game mode is not ($F(1, 276) = 0.27$, $p = 0.605$). A boxplot of the offset angle measurements, grouped by game mode and target, is shown in Figure 9.2.

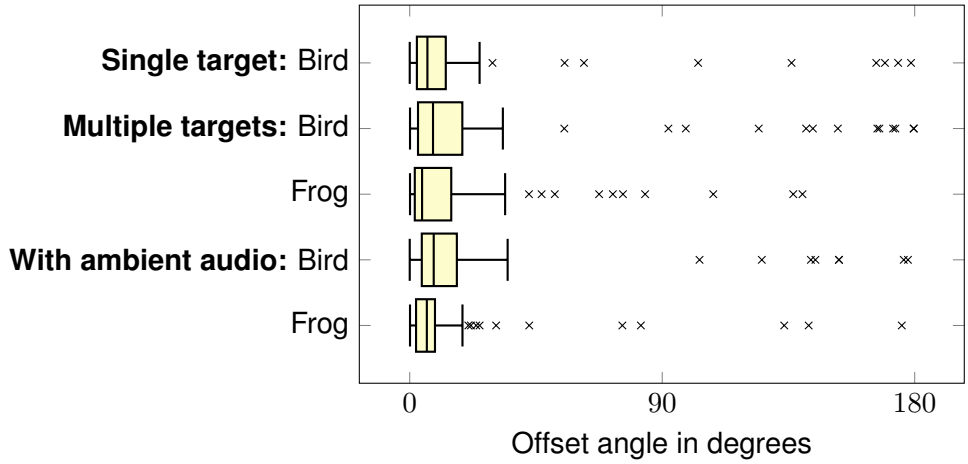


Figure 9.2: Offset angles on the Android device by game mode and target

To further analyze the data the measurements for each valid combination of user, game mode and target were aggregated by calculating the mean of both time and absolute offset angle. This results in a new set of 120 aggregated measurements. Using the aggregated data the average time and offset angles were calculated. The average time the participants took to find the target was 12.1 s with a standard deviation of 11.5 s. The average offset angle was 20.6° with a standard deviation of 32.1°. Table 9.1 shows the averages itemized in regard to the different game modes and targets. The impact of the outliers on the calculated averages can be reduced by using the median instead of the mean. The median time was 9.2 s and the median offset angle was 7.2°.

Mode	Target	Time		Offset	
		Average	SD	Average	SD
Any	Bird	11.3 s	11.4 s	23.6°	37.5°
	Frog	13.4 s	11.8 s	16.1°	21.5°
Single target	Bird	10.4 s	6.0 s	18.2°	28.6°
Two targets	Bird	13.1 s	16.5 s	27.8°	39.2°
	Frog	14.0 s	13.6 s	16.4°	21.2°
With ambient audio	Bird	10.4 s	8.4 s	24.9°	41.8°
	Frog	12.8 s	9.2 s	15.9°	21.4°

Table 9.1: Averages for the aggregated evaluation results itemized by game mode and target

9.3 Comparison

Similar applications to the one described in this thesis were developed for Apple's iOS and Microsoft's Windows Phone. A device independent version based on web technologies was also developed. During the study each participant used all four applications. In this section, the data measurements recorded by the iOS and Windows Phone devices will be compared to the data measured on the Android device.

First a linear mixed effect model is created using the combined data of the Android, the iOS and the Windows Phone application. In addition to the game mode and the target the used operating system is also entered as a fixed effect. The participant remains the only random effect. This model is then used to explore the relationship of the measured time values to the different factors. An ANOVA calculated for this model shows strong significance for both the operating system ($F(2, 1082) = 44.05$, $p < 0.0001$) and the target ($F(1, 1082) = 20.71$, $p < 0.0001$). While the game mode is not significant ($F(2, 1082) = 1.60$, $p = 0.203$), the interaction between operating system and game mode possibly is ($F(4, 1082) = 3.01$, $p = 0.017$). By keeping the effects, the same model is also used to analyze the absolute angle offset values. The OS again shows strong significance ($F(2, 1082) = 8.90$, $p = 0.0001$). The game mode ($F(2, 1082) = 0.91$, $p = 0.401$), the target ($F(1, 1082) = 0.02$, $p = 0.882$) and the interaction between operating system and game mode ($F(4, 1082) = 1.74$, $p = 0.138$) are not significant. To analyze the interaction between the target and the other effects, the data from the

Effect	DF	Time		Offset	
		F	p	F	p
OS	$(2, 852)$	33.55	< 0.0001	10.67	< 0.0001
Mode	$(1, 852)$	0.60	0.438	0.11	0.745
Target	$(1, 852)$	24.76	< 0.0001	0.020	0.889
OS $\times$ Mode	$(2, 852)$	4.28	0.014	1.46	0.233
OS $\times$ Target	$(2, 852)$	0.90	0.407	9.24	0.0001
Mode $\times$ Target	$(1, 852)$	1.54	0.214	1.52	0.218
OS $\times$ Mode $\times$ Target	$(2, 852)$	3.88	0.021	0.23	0.796

Table 9.2: ANOVA of the combined data of any multi-target game mode played on the Android, iOS or Windows Phone version

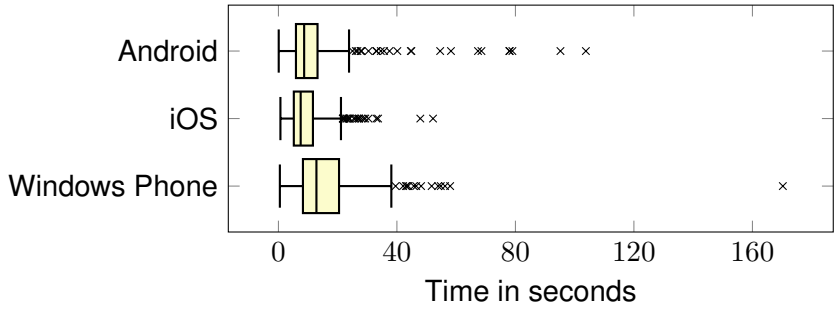


Figure 9.3: Playtimes on the different operating systems

single target game mode is removed again. The results of an ANOVA on this data is shown on in Table 9.2. This data shows a strong significance for the interaction of the operating system with the target regarding the achieved angel offset. The interaction between the operating system and the game mode is also possibly significant regarding the time value. A boxplot comparing the playtime values measured using the different operating systems is shown in Figure 9.3. A comparison of the absolute offset angles is shown in Figure 9.4.

To better compare the individual results of the applications, every valid combination of operating system, user, game mode and target was aggregated by calculating the mean of both time and absolute offset angle. This results in 120 aggregated measurements for both Android and iOS. The Windows Phone version is missing data for two combinations,

OS	Target	Time		Offset	
		Average	SD	Average	SD
All	Bird	11.5 s	9.1 s	17.8°	25.4°
	Frog	14.2 s	9.8 s	19.1°	22.6°
Android	Bird	11.3 s	11.4 s	23.6°	37.5°
	Frog	13.4 s	11.8 s	16.1°	21.5°
iOS	Bird	8.5 s	4.8 s	11.3°	16.6°
	Frog	11.8 s	7.3 s	16.2°	21.5°
Windows Phone	Bird	14.8 s	8.9 s	18.6°	14.0°
	Frog	17.5 s	9.0 s	25.0°	24.1°

Table 9.3: Comparison of the aggregated evaluation results itemized by operating system and target

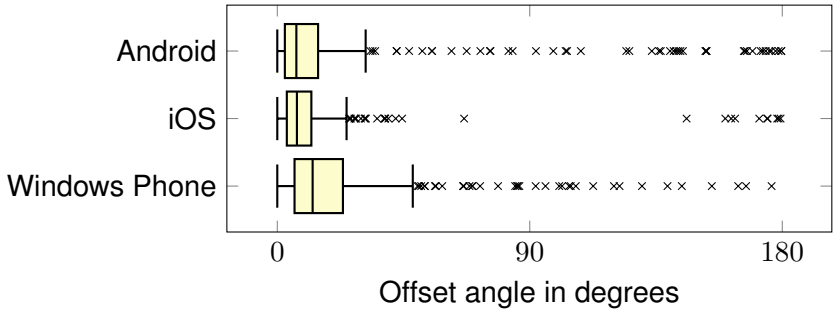


Figure 9.4: Offset angles on the different operating systems

resulting in 118 aggregated measurements. The average time the participants took to finish a game on the iOS version was 9.8 s with a standard deviation of 6.1 s. This compares favorably to the Android version's 12.1 s. The Windows version has an average play time of 15.8 s with a standard deviation of 9.0 s. The average of all available aggregated measurements is 15.6 s with a standard deviation of 9.5 s. When comparing the absolute offset angles, the iOS version achieved the best results with an average of 13.3° and a standard deviation of 18.8°. With 20.6° the Android version has only slightly better results than the Windows version, which has an average of 21.1° and a standard deviation of 18.9°. The overall average of the measured absolute angle offset is 18.3° with a standard deviation of 24.3°. The same calculations were also done for each individual target with the results shown in Table 9.3.

10

Conclusion

In this chapter the results of this project are summarized. This includes a comparison of the requirements for the application and its frameworks with its actual capabilities. An outlook over possible future improvements is also given.

10.1 Results

The goal of this project was the creation and evaluation of an Android application that lets a user react to auditory stimulations. This goal was achieved by implementing a game that uses positional audio as its primary game mechanic. An evaluation was also done in a study where the Android application was compared to similar applications implemented on other operating systems. The positional audio capabilities are usable independently of the main application. They can be accessed using the positional audio API designed as part of this project. This API is implemented using OpenAL and HRTFs. In addition, a loader class for Ogg-Vorbis audio files was created that can

be used in conjunction with the positional audio API. On top of that, a reusable game engine module was designed and implemented. This facilitates the creation of similar games and may serve as a framework for future application with a similar design.

10.2 Requirements Comparison

This section compares the implemented functionality of the frameworks and the application with the requirements specified in chapter 2. Both functional and non-functional requirements are considered.

FR1 **Android application for auditory stimulation**
A demo application with the required capabilities was created and is described in section 8.2.

FR2 **Positional audio sources**
Implemented using OpenAL and HRTFs.

FR3 **Ambient audio sources**
Implemented using OpenAL. Setting the distance of a positional audio source to zero will make it an ambient audio source.

FR4 **Per-source audio volume control**
Implemented using OpenAL.

FR5 **Audio file format**
A class to decode files in the compressed Ogg-Vorbis audio format was implemented as part of the audio API.

FR6 **Audio framework**
A positional audio API was designed and is described in section 5.2. An implementation based on OpenAL that fulfills FR2 to FR4 is also provided.

FR7 **Rotation detection using sensors**
Different types of rotation sensors are supported using the sensor API described in section 5.3.

FR8 Alternative input using the touch screen
Supported as fallback if no sensor is available.

FR9 Graphics for the game
Implemented using OpenGL.

FR10 Picture representation of the target
Implemented using the billboard rendering technique with OpenGL.

FR11 Panorama picture as backdrop
Implemented using OpenGL.

FR12 Reusable game engine
A flexible game engine was designed and implemented. It is described in chapter 7.

FR13 Display results to the user
Implemented as part of the demo application

NFR1 Backwards compatibility
All frameworks and the demo application are backwards compatible to Android version 2.3.3.

NFR2 Small application package
A release build of the demo application has an APK size of 8 Megabytes. The assets, stored as compressed Ogg-Vorbis, PNG or JPEG files, take up 4.8 Megabytes of this APK. With ABI splitting the size of the application package can be reduced to 7.2 Megabytes.

NFR3 Code documentation
All interfaces and classes of the project are documented using JavaDoc or Doxygen comments.

NFR4 Extensibility and reusability
The application was designed with both reusability and extensibility in mind. Most features are located in framework packages that are usable outside the application.

NFR5 Good audio localization

Of the 24 participants that took part in the study, 18 managed to localize the different targets with an average accuracy of less than 20 degrees. Twelve subject even achieved an average of less than 10 degrees. It is currently unclear why the remaining 6 participants could not achieve this level of accuracy.

10.3 Future Work

While the project reached its goals, there are still several ways both the framework as well as the game application could be extended and improved. One possibility would be to improve the audio API by adding support for true three dimensional audio by allowing the user to set the elevation of audio sources. With these capabilities the application could be extended with a game mode that requires the user to guess the position of an audio source in two dimensions. Since OpenAL supports the positioning of audio sources in three dimensions, the implementation should be straightforward. Another way to improve the audio framework would be to allow the user of the API to specify the HRTF definition file that will be used by OpenAL. The challenge with implementing this feature would be that OpenAL only parses its configuration file on startup. Loading a different HRTF file while the application is running would require either significant changes to the OpenAL source code or a way to reload the native library during runtime.

The application itself could also benefit from future improvements. It could be extended with new game modes that employ some of the features provided by the game engine that are not currently used. A new game mode could for example use moving targets to increase the challenge presented to the user. Another currently not utilized feature are the audio filters, which could also be used to create more a new variation of the game. Another way to increase the replayability of the game would be to simply increase the number of assets used by the application. A bigger variation in possible targets, backdrops and ambient audio files would make the game a more interesting experience for returning players.

List of Figures

3.1 A PCM encoder and decoder [4] 10
3.2 A generic digital signal filter 10
3.3 Multiplication of a signal with g 11
3.4 Summation of two signals 11
3.5 Delay of a signal by k samples 11
3.6 Calculation of y_n as the result of the convolution of $x(n)$ and $h(n)$ 12
3.7 A finite impulse response filter 12
3.8 An infinite impulse response filter 13
3.9 The coordinate system used for positional audio 14
3.10 Localization errors caused by ambiguity [52] 15
3.11 The cone of confusion [2] 16
3.12 A KEMAR dummy head with pinnae [61] 17
3.13 Convolution using a separate FIR filter for each ear [2] 18

4.1 Project module dependencies 20
4.2 Java package dependencies 21

5.1 Audio API interfaces 33
5.2 Managed Audio API classes 37
5.3 A biquad filter in direct form 1 [68] 40
5.4 Possible device orientations 45
5.5 Sensor API classes 50
5.6 Sensor world coordinate systems [37] 51

6.1 OpenAL file mapper operation 57
6.2 OpenAL classes 58

7.1 Game engine package dependencies 62

7.2 The game world, the player and three entity objects 63
7.3 Component instance lifecycle 69
7.4 The event manager class 70
7.5 Game data classes 71
7.6 The field of view when rendering the game world 76
7.7 A quad build by using two triangles 80

8.1 Main menu of the demo application 94
8.2 Game result screen of the demo application 95

9.1 Playtime on the Android device by game mode and target 99
9.2 Offset angles on the Android device by game mode and target 100
9.3 Playtimes on the different operating systems 102
9.4 Offset angles on the different operating systems 103

List of Source Codes

4.1 Gradle project properties 23

5.1 `Angle` class 30
5.2 `AudioData` class 32
5.3 `Disposable` interface 33
5.4 `AudioBackend` interface 34
5.5 `AudioBackend.Factory` interface 34
5.6 `AudioBuffer` interface 35
5.7 `AudioSource` interface 35
5.8 `ObservableDisposable` class 36
5.9 `ManagedAudioBackend` class 38
5.10 `ManagedAudioBuffer` class 38
5.11 `ManagedAudioSource` class 39
5.12 `AudioFilter` interface 39
5.13 `AudioFilterChain` class 40
5.14 `BiquadAudioFilters` class 41
5.15 `AudioDeviceInfo` class 43
5.16 `WaveFile` class 43
5.17 `OggVorbisFile` class 44
5.18 `MediaFile` class 44
5.19 `RotationSensor` interface 45
5.20 `RotationSensor.Factory` interface 46
5.21 `AbstractRotationSensor` class 47
5.22 `SensorFilter` interface 48
5.23 `SensorFilterChain` class 48
5.24 `NullSensor` class 50

5.25 `MagneticFieldSensor` class 51
5.26 `SynthesizedSensor` class 52

6.1 OpenAL configuration file 56

7.1 `Component` interface 65
7.2 `BaseComponent` class 66
7.3 `ComponentContainer` class 67
7.4 `ComponentListener` interface 68
7.5 `Player` class 70
7.6 `PlayerListener` interface 71
7.7 `World` class 72
7.8 `Entity` class 73
7.9 `PolarPosition` class 74
7.10 `EngineContext` interface 74
7.11 `AssetBufferReference` class 75
7.12 `GraphicsContext` interface 76
7.13 `OpenGLRenderer` class 77
7.14 `OpenGLSubRenderer` interface 77
7.15 `BlendingShaderProgram` class 79
7.16 `VertexQuad` class 80
7.17 `TextureBufferManager` class 81
7.18 `AudioContext` interface 84
7.19 `AudioPlayer` class 85
7.20 `AudioSystem` class 86
7.21 `AudioBufferManager` class 87
7.22 `InputContext` interface 88
7.23 `SensorInput` class 88
7.24 `TouchViewInput` class 89

List of Tables

9.1 Averages for the aggregated evaluation results itemized by game mode and target 100
9.2 ANOVA of the combined data of any multi-target game mode played on the Android, iOS or Windows Phone version 101
9.3 Comparison of the aggregated evaluation results itemized by operating system and target 102

List of Acronyms

ABI application binary interface

ANOVA analysis of variance

API application programming interface

APK Android application package

BSD Berkeley software distribution

CD compact disc

CPU central processing unit

DF degrees of freedom

DSL domain-specific language

DSP digital signal processing

FIFO first in, first out

FIR finite impulse response

GCC GNU compiler collection

GNU GNU's not unix

GPU graphics processing unit

HRIR head-related impulse response

HRTF head-related transfer function

HTML HyperText markup language

IDE integrated development environment

IID interaural intensity difference

IIR infinite impulse response

ITD interaural time difference

JNI Java native interface

JPEG joint photographic experts group

JSR Java specification request

JVM Java virtual machine

KEMAR Knowles Electronics mannequin for acoustics research

LGPL GNU lesser general public license

MIT Massachusetts institute of technology

MP3 MPEG-2 audio layer III

MPEG moving picture experts group

NDK native development kit

OS operating system

PCM pulse code modulation

PNG portable network graphics

POSIX portable operating system interface

SD standard deviation

SDK software development kit

TRI tinnitus research initiative

UI user interface

Bibliography

[1] Tomas Akenine-Möller, Eric Haines, and Naty Hoffman. *Real-Time Rendering*. 3rd ed. CRC Press, 2008. ISBN: 978-1439865293.

[2] Durand R. Begault. *3-D Sound for Virtual Reality and Multimedia*. National Technical Information Service, 2000.

[3] Joshua Bloch. *Effective Java*. 2nd ed. Addison-Wesley, 2008. ISBN: 978-0321356680.

[4] Marina Bosi and Richard E. Goldberg. *Introduction to Digital Audio Coding and Standards*. The Springer International Series in Engineering and Computer Science. Springer, 2003. ISBN: 978-1402073571.

[5] Patrick Brady. "Anatomy and Physiology of an Android". In: *Google I/O Developer Conference 2008*. (May 28–29, 2008).

[6] Robert Bristow-Johnson. *Cookbook formulae for audio EQ biquad filter coefficients*. URL: `http://www.musicdsp.org/files/Audio-EQ-Cookbook.txt` (visited on June 6, 2015).

[7] Douglas S. Brungart, Brian D. Simpson, and Alexander J. Kordik. *The detectability of headtracker latency in virtual audio displays*. Georgia Institute of Technology, 2005.

[8] Jos J. Eggermont and Larry E. Roberts. "The neuroscience of tinnitus". In: *Trends in neurosciences* 27.11 (2004), pp. 676–682.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. ISBN: 978-0201633610.

[10] Bill Gardner and Keith Martin. *HRTF Measurements of a KEMAR Dummy-Head Microphone*. URL: http://sound.media.mit.edu/resources/KEMAR.html (visited on July 4, 2015).

[11] Google. *Action Bar*. URL: http://developer.android.com/guide/topics/ui/actionbar.html (visited on July 9, 2015).

[12] Google. *Activity class*. URL: http://developer.android.com/reference/android/app/Activity.html (visited on July 12, 2015).

[13] Google. *Android NDK*. URL: https://developer.android.com/ndk/index.html (visited on July 9, 2015).

[14] Google. *Android NDK Preview*. URL: http://tools.android.com/tech-docs/android-ndk-preview (visited on July 11, 2015).

[15] Google. *Android SDK*. URL: https://developer.android.com/sdk/index.html (visited on July 9, 2015).

[16] Google. *Android Studio Overview*. URL: http://developer.android.com/tools/studio/index.html (visited on July 6, 2015).

[17] Google. *AutoCloseable interface*. URL: http://developer.android.com/reference/java/lang/AutoCloseable.html (visited on June 4, 2015).

[18] Google. *Closeable interface*. URL: http://developer.android.com/reference/java/io/Closeable.html (visited on June 4, 2015).

[19] Google. *Context class*. URL: http://developer.android.com/reference/android/content/Context.html (visited on June 4, 2015).

[20] Google. *Dashboards*. URL: https://developer.android.com/about/dashboards/index.html (visited on Aug. 18, 2015).

[21] Google. *Design For Reduced Latency*. URL: https://source.android.com/devices/audio/latency_design.html (visited on June 7, 2015).

[22] Google. *Experimental Plugin User Guide*. URL: http://tools.android.com/tech-docs/new-build-system/gradle-experimental (visited on July 11, 2015).

[23] Google. *Fragments*. URL: http://developer.android.com/guide/components/fragments.html (visited on Aug. 22, 2015).

[24] Google. *GLSurfaceView.Renderer interface*. URL: http://developer.android.com/reference/android/opengl/GLSurfaceView.Renderer.html (visited on July 12, 2015).

[25] Google. *Guava: Google Core Libraries for Java*. URL: https://github.com/google/guava (visited on July 6, 2015).

[26] Google. *Handler class*. URL: http://developer.android.com/reference/android/os/Handler.html (visited on June 9, 2015).

[27] Google. *Improving Code Inspection with Annotations*. URL: http://developer.android.com/tools/debugging/annotations.html (visited on July 6, 2015).

[28] Google. *Managing Audio Focus*. URL: http://developer.android.com/training/managing-audio/audio-focus.html (visited on Aug. 9, 2015).

[29] Google. *Material Design for Android*. URL: http://developer.android.com/design/material/index.html (visited on July 9, 2015).

[30] Google. *MediaCodec class*. URL: http://developer.android.com/reference/android/media/MediaCodec.html (visited on July 27, 2015).

[31] Google. *OpenGL ES*. URL: http://developer.android.com/guide/topics/graphics/opengl.html (visited on July 28, 2015).

[32] Google. “OpenSL ES for Android”. In: *Android NDK documentation*.

[33] Google. *ProGuard*. URL: http://developer.android.com/tools/help/proguard.html (visited on July 9, 2015).

[34] Google. *Responding to Touch Events*. URL: http://developer.android.com/training/graphics/opengl/touch.html (visited on Aug. 10, 2015).

[35] Google. *Sensor class*. URL: http://developer.android.com/reference/android/hardware/Sensor.html (visited on June 10, 2015).

[36] Google. *SensorEvent class*. URL: http://developer.android.com/reference/android/hardware/SensorEvent.html (visited on June 10, 2015).

[37] Google. *SensorManager class*. URL: http://developer.android.com/reference/android/hardware/SensorManager.html (visited on June 10, 2015).

[38] Google. *Support Library*. URL: http://developer.android.com/tools/support-library/index.html (visited on July 9, 2015).

[39] Google. *Versioning Your Application*. URL: http://developer.android.com/tools/publishing/versioning.html (visited on July 8, 2015).

[40] Gradle. *Gradle - Endgame Open-Source Enterprise Build Automation*. URL: https://gradle.org/ (visited on July 9, 2015).

[41] Gradle. *The Gradle Wrapper*. URL: https://docs.gradle.org/current/userguide/gradle_wrapper.html (visited on July 6, 2015).

[42] Chet Haase and Dan Sandler. “What’s new in Android”. In: *Google I/O Developer Conference 2015*. (May 28–29, 2015).

[43] Dimitri an Heesch. *Doxygen*. URL: http://www.stack.nl/~dimitri/doxygen/ (visited on July 9, 2015).

[44] Jochen Herrmann. *Track your tinnitus*. URL: https://www.trackyourtinnitus.org/ (visited on Aug. 27, 2015).

[45] Garin Hiebert. *OpenAL 1.1 Specification and Reference*. 2005. URL: http://openal.org/documentation/openal-1.1-specification.pdf (visited on June 18, 2015).

[46] Information Society Technologies. *Listen HRTF Database*. URL: http://recherche.ircam.fr/equipes/salles/listen/index.html (visited on July 4, 2015).

[47] 3D Working Group of the Interactive Audio Special Interest Group. *Interactive 3D Audio Rendering Guidelines - Level 2.0*. Sept. 20, 1999.

[48] Kitware. *CMake*. URL: http://www.cmake.org/ (visited on July 9, 2015).

[49] Peter M. Kreuzer, Veronika Vielsmeier, and Berthold Langguth. "Chronic tinnitus: an interdisciplinary challenge". In: *Deutsches Ärzteblatt International* 2013 (2013), pp. 278–284.

[50] Eric Lafortune. *ProGuard*. URL: http://proguard.sourceforge.net/ (visited on July 9, 2015).

[51] Aage R. Moller, Berthold Langguth, Goran Hajak, Tobias Kleinjung, and Anthony Cacace. *Tinnitus: Pathophysiology and Treatment*. Progress in Brain Research. Elsevier Science, 2007. ISBN: 9780080554464.

[52] Henrik Møller. "Fundamentals of Binaural Technology". In: *Applied acoustics* 36.3 (1992), pp. 171–218.

[53] Juhan Nam, Miriam Kolar, and Jonathan Abel. "On the Minimum-phase Nature of Head-Related Transfer Functions". In: *The 125th Audio Engineering Society Convention*. San Francisco, CA, USA: AES, 2008.

[54] Robert Nystrom. *Game Programming Patterns*. ISBN: 978-0990582908.

[55] Oracle. *javadoc - The Java API Documentation Generator*. URL: http://docs.oracle.com/javase/7/docs/technotes/tools/windows/javadoc.html.

[56] Oracle. *JNI Functions*. URL: http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html (visited on June 1, 2015).

[57] Oracle. *JSR 305: Annotations for Software Defect Detection*. URL: https://jcp.org/en/jsr/detail?id=305.

[58] Oracle. *The Invocation API*. URL: https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/invocation.html (visited on June 1, 2015).

[59] Oracle. *The try-with-resources Statement*. URL: https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html.

[60] Avery Pennarun. *git-subtree - Merge subtrees together and split repository into subtrees*. URL: https://raw.githubusercontent.com/git/git/master/contrib/subtree/git-subtree.txt.

[61] Chris Pike. *Listen Up! Binaural Sound*. URL: http://www.bbc.co.uk/blogs/researchanddevelopment/2013/03/listen-up-binaural-sound.shtml (visited on July 4, 2015).

[62] Jose Pinheiro, Douglas Bates, Saikat DebRoy, Deepayan Sarkar, and R Core Team. *nlme: Linear and Nonlinear Mixed Effects Models*. R package version 3.1-121. 2015. URL: http://CRAN.R-project.org/package=nlme.

[63] Nick Prühs. “Komponentenweise: Component-Based Entity Systeme in Spielen”. In: *iX Developer* (2015), pp. 70–74.

[64] R Core Team. *R: A Language and Environment for Statistical Computing*. Version 3.2.2. R Foundation for Statistical Computing. Vienna, Austria, 2015. URL: https://www.R-project.org/.

[65] Chris Robinson. *OpenAL Soft*. URL: http://kcat.strangesoft.net/openal.html (visited on June 18, 2015).

[66] David Sachs. “Sensor Fusion on Android Devices: A Revolution in Motion Processing”. In: *Google Tech Talk*. (Aug. 2, 2012).

[67] Winfried Schlee, Isabel Lorenz, Thomas Hartmann, Nadia Müller, Hannah Schulz, and Nathan Weisz. “A global brain model of tinnitus”. In: *Textbook of Tinnitus*. Springer New York, 2011, pp. 161–169. ISBN: 978-1607611448.

[68] Julius O. Smith. *Introduction to Digital Filters with Audio Applications*. URL: https://ccrma.stanford.edu/~jos/filters/ (visited on July 22, 2015).

[69] The Khronos Group. *OpenSL ES Specification Version 1.0.1*. Sept. 24, 2009. URL: https://www.khronos.org/registry/sles/specs/OpenSL_ES_Specification_1.0.1.pdf (visited on June 18, 2015).

[70] *The Open Group Base Specifications Issue 7*. IEEE 1003.1. 2013.

[71] Elizabeth M. Wenzel, Marianne Arruda, Doris J. Kistler, and Frederic L. Wightman. “Localization using nonindividualized head-related transfer functions”. In: *The Journal of the Acoustical Society of America* 94.1 (1993), pp. 111–123.

[72] Jörg Weske. *Digital Sound and Music in Computer Games*. URL: http://3daudio.info/gamesound/games.html (visited on June 23, 2015).

[73] *Working Draft, Standard for Programming Language C++*. ISO N3337. Jan. 16, 2012.

[74] Xiph.Org Foundation. *Vorbis audio compression*. URL: https://xiph.org/vorbis/ (visited on July 10, 2015).

[75] Xiph.Org Foundation. *Vorbis.com FAQ*. URL: http://www.vorbis.com/faq/ (visited on June 8, 2015).

Name: Jan-Dominik Blome Matrikelnummer: 642979

Erklärung

Ich erkläre, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den .....

Jan-Dominik Blome