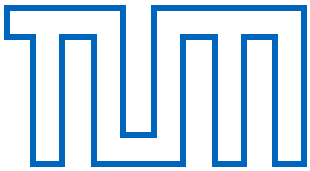
FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelorarbeit in Informatik

Implementation of a games development tool for mobile devices

Julian Sievers



FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelorarbeit in Informatik

Implementation of a games development tool for mobile devices

Implementierung einer Spiele-Entwicklungsumgebung für mobile Geräte

Author:	Julian Sievers
Supervisor:	Prof. Dr. Uwe Baumgarten
Advisor:	Nils T. Kannengießer, M.Sc.
Date:	October 15, 2012

I assure the single handed composition of this bachelor thesis only supported by declared resources.

Munich, 12th of October, 2012 Julian Sievers

Abstract

This thesis describes the development of an editor which is capable of creating games for the mobile operating system Android. Due to the massive possibilities this topic offers, only the basic functionalities are implemented in the "Android Game Studio" application created in the course of this thesis. Nevertheless, the application can be easily improved in the future because of the MVC approach, depending on certain restrictions which are addressed in a separate chapter. The thesis is split into four parts:

The introductory part at the beginning of the thesis gives background information and describes fundamentals of the Android operating system and the OpenGL ES framework, which are used to develop the game studio application.

The main part then builds upon this knowledge and presents the design intentions and requirements , both, the game studio application and its created games have to meet. Additionally, an overview about the structure of editor and game is given and the the organisation of the underlying database is illuminated. After the design part, the actual implementation is considered and details of the game editor components are explained, based on several code parts.

Part three outlines the results of a user test and a code analysis tool to verify the correctness of the game studio application and its created games.

Finally, the thesis is concluded by presenting results about the implementation and future improvements to the editor are considered.

Contents

Outline of the Thesis

Part I: Introduction and Theory

Chapter 1: Introduction

The first chapter presents an overview of the thesis and its purpose.

Chapter 2: Background Information

This chapter outlines the basics of Android and OpenGL ES.

Part II: Editor and Game Implementation

Chapter 3: System Design

In chapter three, the system design and requirements are described.

Chapter 4: Implementation

The implementation process and details of the application components are explained in chapter four.

Part III: Quality Management

Chapter 5: Software Testing

This chapter presents basics of software testing and the performed tests for the game studio application.

Part IV: Conclusion and Outlook

Chapter 6: Conclusion

Chapter six summarises the intention of the thesis and reviews the achieved goals.

Chapter 7: Outlook

The last chapter considers possible future improvements and assesses the prospects of the application.

Part I.

Introduction and Theory

Introduction

Android devices have become more and more popular in these days. Quite recently, Google developer Hugo Barra announced the 500th million activated Android device and stated that 1.3 million devices are added each day [4]. Driven by this development the mobile gaming market will also vastly expand over the next years in Germany: From € 29 million in 2011 to a predicted volume of € 60 million in 2016, which makes an average growth of 18.6 % a year. [28]

Mobile games are different from games for video game consoles and PCs. The device hardware offers many more user interaction methods by utilising touch input and different sensors such as motion, environmental and position sensors. However, they suffer from the small screen which limits the displayed graphics. The possible game mechanics and genres are nearly the same as on dedicated gaming platforms, as proven by the latest single- and multiplayer, as well as high-definition 3D games. Currently, most games designed for mobile devices are "casual games" which do not require the player to spend hours and hours on the game, but rather invite him to play just a few minutes now and then.

Creating applications on an Android device has been possible for quite a while now, but there are no tools for laymen who want to create their own app. All available tools such as AIDE, C# To Go or IDEdroid Free require programming knowledge and are therefore no option to make application programming more accessible to a wider public.

The goal of this thesis was to implement a game editor which is capable of creating two-dimensional singleplayer games for Android devices. Constructing and compiling should be entirely possible on the device and not require any background knowledge of the subject matter. However, using other applications to handle the compiling part was not explicitly excluded and is therefore part of the implementation. All main components which make a game interesting should be included in the game editor, including components to create a scene, add events to provide the logic of the game and select overall game settings such as control methods and gravity. Moreover, basic game design principles are explained and the thesis also takes a closer look at the fundamental game components, for example graphics and physics engine.

Background Information

2.1 Android Framework

Android is a Linux based open-source software stack that runs on different types of devices, especially mobile phones, portable devices and embedded systems. Andy Rubin, Google's director of mobile platforms describes Android as follows:

> *"The first truly open and comprehensive platform for mobile devices. It includes an operating system, user-interface and applications - all of the software to run a mobile phone but without the proprietary obstacles that have hindered mobile innovation."* [21]

Maintenance and development is performed by the Android Open Source Project which is led by Google.

2.1.1 System Architecture

The Android software stack consists of four layers with five major components which include the operating system, middleware and key applications. With the Android **S**oftware **D**evelopment **K**it, developers are able to access all parts of the software stack via special APIs. Therefore, every developer, including Google employees, theoretically has the same possibilities for application development. [12][26]

Figure 2.1 gives an overview on the software stack and its components which are then described bottom-up:

Linux Kernel Android is based on a Linux Kernel, version 2.6. It acts as hardware abstraction layer and provides basic system services, such as memory management, security, process management and a network interface. Another advantage of the Linux Kernel is its high portability. Since most of the Linux code is written in C, it is easy to make Android available for a large variety of devices. [10][12]

Android Runtime The Android Runtime consists of two parts: All of Java's core libraries and a virtual machine, the Dalvik Virtual Machine, executing application code.
The integrated Java libraries like java.math, java.util and java.io provide basic functionality for every application. But not all Java libraries are available in Android, some of them were removed due to unnecessary functionality, for example printing

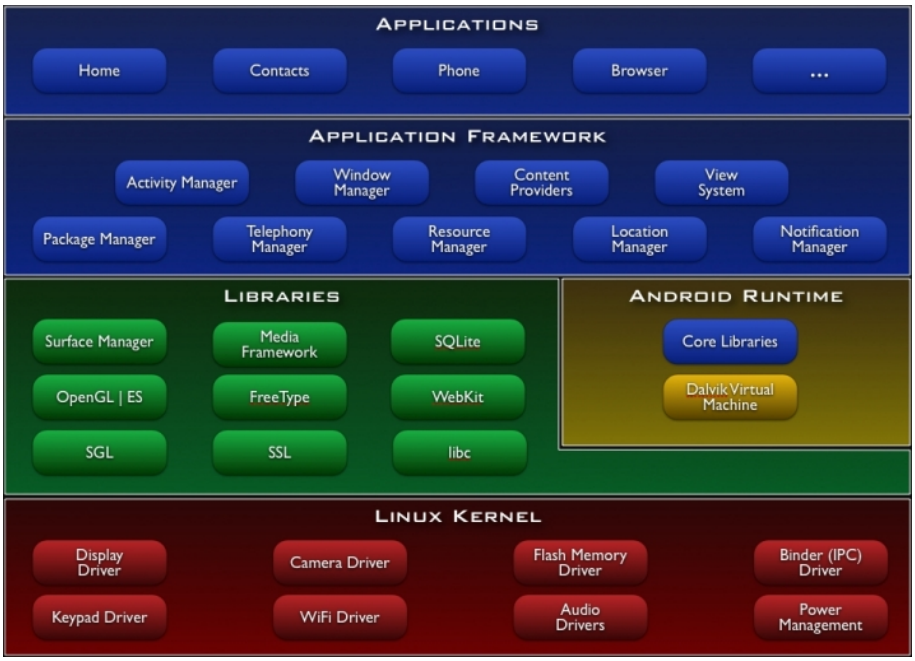


Figure 2.1.: Android System Architecture

(javax.print) and painting graphics (java.awt).

The Dalvik Virtual Machine has its origins in the Java Virtual Machine, but is a complete remake due to license issues. Although using the Java programming language with free tools and libraries, Oracle charges a fee for using the Java Virtual Machine. But Google's own implementation also has other advantages. The DVM is especially designed to run on mobile devices and therefore optimised for battery life, computing power and minimal memory usage. Contrary to the JVM, which has a stack-based architecture, the DVM is register-based. For every single Android application an instance of Dalvik VM is created which runs the application in its own process. Since the Dalvik VM is not compatible with standard Java bytecode, all Java files have to be translated to a special Dalvik bytecode, indicated by the *.dex* file extension. [10][26]

Libraries Android's core libraries are all implemented in C/C++ and accessible through the Android application framework. Some of the libraries are also used in this project, for example the 3D libraries, an implementation of the OpenGL ES APIs for 3D hardware acceleration, or the SQLite library, a lightweight relational database management system. Other available libraries are: [12]

- LibWebCore
- Media Libraries
- Surface Manager
- System C library

Application Framework Android provides a large variety of high-level system services which are wrapped up in the Application Framework.The framework offers a standardised structure for all applications. All developers are able to access the same API functions as Android's core applications. The design of the framework allows applications to interact with each other and therefore increases re-usability of already existing components.

Some of the framework components are an essential part of every application, such as the View System or the Activity Manager, others are only optional and maybe not used at all. The mentioned View System, for example, is responsible for the application's GUI. With Buttons, TextViews, Layouts and much more, the developer has great possibilities of shaping the applications user interface. Another crucial part of an Android application is the Activity Manager. It manages the application's life cycle and provides an Activity stack to navigate through the different active Activities. The last example is the Resource Manager, which is also extensively used in this project. It gives access to non-code resources which are, for better organisation, defined in special files. Colors, localized strings and GUI layouts are just a small excerpt from all the available resources of the Resource Manager. [12]

Applications In delivery state, some applications are already installed. Browser, calendar, contacts and email client are just some examples of the preinstalled apps. But not only these rather basic apps can be installed. Since Android is open source, every manufacturer and even every cell phone provider can include their own applications and ship the phone with more than just the basics.

Additionally, developers can create applications on their own and contribute to the variety of the Android platform. Most applications are written in Java but it is also possible to use the **N**ative **D**evelopment **K**it and implement apps in native-languages such as C and C++. Distributing is done via the build-in Plays Store or other alternative app markets like GetJar. [11] [12]

2.1.2 Application Fundamentals

An Android application consists of loosely tied components (activities, services, broadcast receivers, ...) which are responsible for the behaviour and program flow of an app. Every component used has to be declared in the manifest file which is responsible for their connection. All components are then packed into one single archive file, together with all resources and data of an Android project. This package can then be used to install an application on any Android device.

As described in section 2.1.1, each installed application runs in its own Linux process with a unique ID. This ID is part of Android's security system, as it guarantees that application data can only be read and manipulated by the application with the same ID assigned to a file when it is created. Third-Party programs are therefore not able to read other program data.

However, there are possibilities to enable data exchange between two applications. First of

all, developers can assign the same ID to more than one of their applications. Thus, the OS does not prevent access to their files from each other. Secondly, an application can request different types of permissions on installation. If the permissions are granted by the user, the application can get access to the device data, for instance the users phone contacts, his messages, the SD-Card or the camera. The last possibility is to create a Content Provider and make the data available to third parties with the built in data management system. [13]

2.1.3 Application Components

Some of the application components were already described, but will now be explained in more detail:

Activities Activities represent the applications graphical user interface graphical user interface (GUI). The GUI consists of a set of Views that can be buttons, text fields and other objects and is either defined in an XML layout file, or code based. Usually an activity is a single screen, but in some cases, a screen can show more than one activity at the same time. This is especially true for a TabActivity holding other activities in its tabs. Additionally, to the graphical representation, the activity is also responsible for user input and interaction.

This project is a good example for using multiple activities, each with a different specific purpose. For instance, scene and event editor activities are connected - the event editor in this particular case is started by the scene editor - although completely separated from each other. The scene editor is used to place objects and create the level, the event editor prepares a list of functionality the game should have by providing event and trigger templates. Because of that design, it is also possible to start an activity from everywhere in the program flow, even from other applications if this feature is enabled by the developer. [13]

Services The service component is specifically designed to run in background for an unspecified amount of time and does not have a user interface. A typical example for a service is constantly checking mails or requesting data in a given time interval, even if the activity is inactive or the user interacts with other applications in the meantime. A service can be started and bound by another component in this list. [13]

Content providers Content providers are used to exchange data between applications. The address book is a practical example, as it implements a content provider to add, modify, delete and query contacts. To communicate with a content provider, the application has to create an instance of the ContentResolver class. The functionalities mentioned are the basis of every content provider and applicable to store any data on a persistent storage media including local SQLite databases, the file system or even on the web. The administrative work behind a content provider is handled by Android itself. The OS makes sure a content provider is up and running on demand and handles multiple communication requests from different content resolver objects. A prerequisite for an application to use a content provider is, however, to request the proper permissions. [13]

Broadcast receivers To receive and react to system events a broadcast receiver is the appropriate solution. There are many different events a broadcast receiver can react to, for example if the phone completed booting or the power cable was plugged in. Additionally, developers can create more events on their own and react to these in their applications. Two types of receivers are available: static and dynamic ones. A static receiver has to be declared in the Android manifest file which allows it to get triggered, even if the application is not started (especially applications responding to the BOOT_COMPLETED event make use of this feature). On the other hand, a dynamic receiver is created during application runtime and only active when the application runs in the foreground (it has to be registered and unregistered properly). Similar to a service and a content provider, this component does not have a user interface. However, it is possible to start an activity or display simple messages in a Toast or use the notification system of Android. [13]

Intents Activities, services and broadcast receivers are activated by an asynchronous message called intent. An intent is an abstract description of the action to perform. There are two different types of intents:

- Explicit intents are more likely to be used in development because these are responsible for inner app messages, e.g. starting another activity. To start an activity with an intent, the exact class name is required.
- Implicit intents do not name a specific target, but rather provide information on which the operating system tries to identify the best suitable target component. To react to an intent, a component has to implement an intent filter. Installing an application with an intent filter will register this filter in the Android system and make it available to match intents against it. Examples for implicit intents are calling a number or starting the gallery to pick an image.

The example just mentioned of picking an image in the gallery with an implicit intent makes use of an additional feature. Intents may return data, such as the picked image, which is then sent back as another intent. Since intents are bundles of information, adding extra information is also possible to pass parameters or object data easily to other components. [13]

Android manifest The Android manifest file is not a component itself, but every component used in an application has to be declared in the manifest file in order to work correctly. On top of this, the manifest file is also responsible for many fundamental application settings like target and minimum Android version, intent filters, requested permissions and used additional libraries such as maps or AWT. [13]

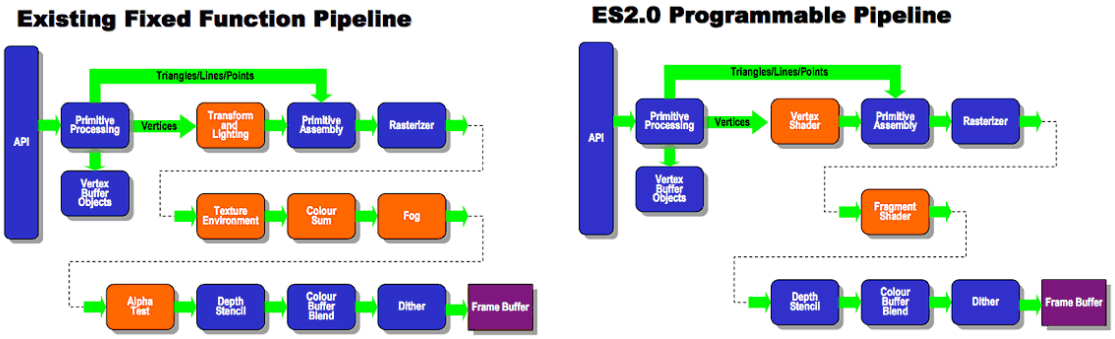
2.2 OpenGL ES

Complex two or three dimensional graphics is the basis for every great up to date game. The editor and the created games in this project therefore make use of a special API called OpenGL ES, designed to create advanced graphics applications. The following sections give a short introduction to this API, provide background information about the development and present basics on how to use OpenGL ES.

2.2.1 The OpenGL ES Graphics System

There are two standard APIs for graphics programming, Direct3D and OpenGL. Both can be used on desktop systems, but only OpenGL is an appropriate solution for mobile systems. OpenGL ES is a cross-platform, free of charge graphics API with 2D and 3D support, specially created for embedded and mobile systems. It is derived from the desktop version of OpenGL with a reduced range of functions (e.g. removed double values) and improved features like increased shader performance to meet the requirements of limited hardware power. The main developer and maintainer of OpenGL and its derivatives including OpenGL ES, the Khronos Group, has published four different versions so far:

- **OpenGL ES 1.0** First official implementation that provides a lightweight interface and a fixed function pipeline. Ships with enabled software rendering and hardware acceleration.
- **OpenGL ES 1.1** Improves image quality, optimizes performance and reduces memory bandwidth. Extension package adds functionality, such as cube maps, frame buffer objects and stencils.
- **OpenGL ES 2.0** Replaced the fixed functionality pipeline with a programmable pipeline and added a basic shading language. This allows developers to write own shader programs to minimize power consumption of graphics implementations.
- **OpenGL ES 3.0** Version 3.0 adds for ETC2/EAC texture compression formats, allows non-power-of-two textures with full wrap mode and support of multiple render targets. This is only a short extract, for the whole specification see the OpenGL ES 3.0 specification paper.



(a) OpenGL ES 1.1 graphics pipeline

(b) OpenGL ES 2.0 graphics pipeline

Figure 2.2.: Fixed function pipeline compared to programmable pipeline

The fundamental idea of OpenGL ES is to provide a software interface to the device's graphics hardware. To draw objects and show the output to the user, a developer has to set up a framebuffer which contains the color, depth and stencil information for every pixel of the rendered image. Hereafter, the context of OpenGL ES is set up and now ready to

draw something. A set of different commands allows the programmer to specify objects and rendering operations like lighting, coloring and transforming to create a scene and show it to the user. [20] [27]

2.2.2 OpenGL ES Basics

Major parts of the editor and the games created in this project are based on OpenGL grahpics, therefore the basics of OpenGL ES shall be addressed to in this section. Both, the connection of the Android framework with OpenGL ES API and basic OpenGL operations to manipulate objects and create a scene are explained briefly. It is also possible to use OpenGL combined with the Android NDK, but since this functionality is not used in the project it will not be explained further here. Basically, only utilised functions in editor and game are explained in this section.

The two fundamental classes of the OpenGL ES API are GLSurfaceView and GLSurfaceView.Renderer. In order to create a two or three dimensional game, a GLSurfaceView has to be instantiated and the Renderer interface has to be implemented in a custom class with the following functionality:

GLSurfaceView The GLSurfaceView displays the rendering output of OpenGL to the user, by attaching an instance of a class implementing the Renderer interface. Originally, the view is not able to react to touch input, which is required in most use-case scenarios. There are two possibilities to interact with View: The most common way - this is also proposed by Google [15] - is subclassing the GLSurfaceView and implementing the corresponding listeners directly. Customizing the GLSurfaceView is possible by using its "set" methods. This leads directly to the second option, adding external listeners to the view.

GLSurfaceView.Renderer The renderer interface provides all necessary methods for drawing objects in OpenGL. Since an interface cannot be instantiated, it has to be integrated in a custom class, which implements the functionality:

- **onSurfaceCreated(...):** On GLSurfaceView creation and recreation, the system calls this method once. All permanent settings to the OpenGL state machine, initialising and loading objects as well as their textures have to be performed here.
- **onDrawFrame(...):** This method is repeatedly called and responsible for drawing the objects. Additionally, some other settings have to be managed here, too. Typically, the model-view matrix is loaded and the framebuffer is reset by clearing its colour and depth buffer.
- **onSurfaceChanged(...):** Altering the GLSurfaceView, such as changing the size or rotating the device, triggers the onSurfaceChange(...) method. To react to these changes, the viewport has to be set once again. If the scene is based on a fixed camera, the projection matrix and viewing frustum can also be loaded.

As the onDrawFrame(...) method indicates, the renderer runs on its own dedicated thread. This architecture decouples the performance of the OpenGL representation

and the user interface, but comes along with problems when the render thread interacts with other threads, for example the logic behind the game. Synchronisation can be achieved by any standard Java technique for cross-thread communication or the queueEvent(Runnable) method, which allows to paste a new Runnable object to the renderer queue and execute it once the renderer becomes active.

To understand the graphics engine of the project, the most important OpenGL ES functionalities are now explained and then addressed to further in section 4.1 Graphics engine on page 31.

OpenGL context The EGL API, developed by Khronos Group, manages the drawing surface including surface creation and destruction. EGL is the link between the devices windowing system and OpenGL ES and defines the communication of Android OS with the hardware. EGL is also responsible for managing rendering resources and all available drawing surfaces. However, Android developers do not have to touch these features, because the GLSurfaceView takes care of that.

By default, double buffering is enabled for GLSurfaceView so that the image is rendered in an invisible framebuffer first, and then displayed to the user as a whole. Otherwise, the user would notice a flickering of the screen while the rendered image is created. [27][30]

Viewport At the beginning and every time the size of the surface view changes, the viewport size has to be set. The viewport determines the portion of the surface, the renderer is drawing to. Typically, the viewport is set up to match the surface area's resolution, but can also be configured to show only a subsection of the surface framebuffer. The purpose of the viewport is to transform the projected object coordinates to device window coordinates. [19][30]

Viewing Frustum In 3D graphics, the viewing frustum defines the portion of the world which actually can be seen. In perspective projections, the frustum is a pyramid like volume, in orthographic projections it is a box. The viewing frustum coincides with the camera's field of view, but additionally is cut off at the near and far clip which exclude objects that are too far away or too close to the camera.[29]

Orthographic Projection Since this project only allows two dimensional graphics, a so-called orthographic (or parallel) projection is used for rendering. The corresponding OpenGL function takes the boundary pixel coordinates as input and creates a view frustum box. All points in the frustum box are then projected to the near clipping plane for rendering. Near and far clip of the box are mostly set to 1 and -1 one, which makes the viewing frustum completely flat. Therefore an object is not influenced by the perspective, as they are all in the same layer. Note that the coordinate system in OpenGL differs from the UI framework and Canvas coordinate system (see figure 2.3), as it has its origin in the bottom left instead of the top left corner. [30]

Geometry In order to render objects, their geometry has to be defined. It is either possible to load an object representation from a resource file, or to create the geometry in the application. The first approach is normally used for complex objects with hundreds

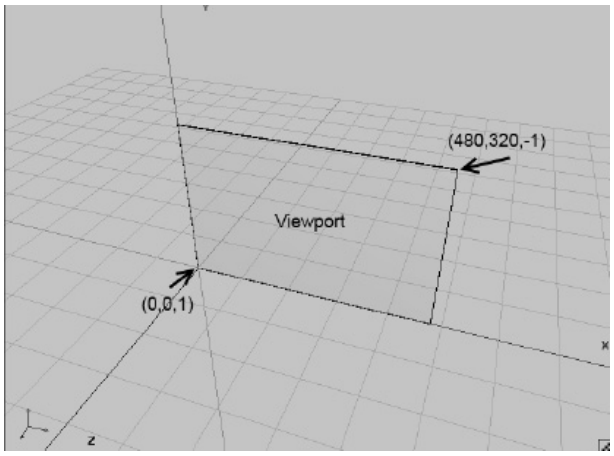


Figure 2.3.: Viewing frustum of an orthographic projection with standard values for a surface resolution of 480 x 320 pixels. [30]

or thousands of polygons, the second one is the right choice for simple geometry like 2D rectangles, which is used as part of this thesis.

Objects in OpenGL are composed of triangles which are for their part composed of vertices. To create a rectangle for displaying textures, two triangles are required. Instead of defining every triangle on its own - this would lead to duplicate edges - only the four edge vertices are defined and then connected to a triangle with an index array. The index array determines the vertices of the triangles in counter clockwise order. Both, the array holding the vertex, as well as the array holding the index data is then translated and put in separate ByteBuffers. After that, the vertex buffers' reference is passed to OpenGL ES for drawing the rectangle. Drawing the objects is then realised by the glDrawElements function of OpenGL ES which takes index buffer data as input parameters. [29][30]

World and model space Two different coordinate systems have to be distinguished in graphics programming: The world and model space. World space is responsible for global positioning of objects and contains every object of the scene. On the contrary, the model space is a local coordinate system for every object, in which the vertices of an object are defined. Therefore, vertices are not positioned relative to the origin of the global coordinate system, but relative to their own local system. The motivation of this design is to display a single object at different positions in the space without redefining the vertex positions, but by using internal transformation functions of OpenGL ES. [30]

Transformations The distinction of world and model space makes it easy to rotate, scale and move objects. It is possible to rotate around specified axis, translate the object by a given amount of units and scale it with a scalar. Note that matrix operations - rotating, translating and scaling is nothing else than multiplying matrices - are not commutative, thus their order is crucial. Figure 2.4 demonstrates the difference of

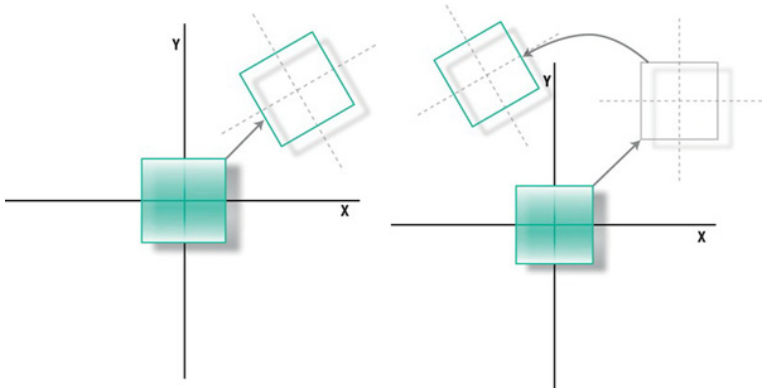


Figure 2.4.: Rotation and Translation versus Translation and Rotation. [29]

first rotating and then translating an object, and vice versa. [29]

The following matrix equations give an insight into the operations performed by the corresponding OpenGL ES functions glRotatef(), glScalef() and glTranslatef():

$$R(x,\alpha) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & \sin\alpha & 0 \\ 0 & -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

(a) Rotation around x-axis

$$R(y,\alpha) = \begin{pmatrix} \cos\alpha & 0 & -\sin\alpha & 0 \\ 0 & 1 & 0 & 0 \\ \sin\alpha & 0 & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

(b) Rotation around y-axis

$$R(z,\alpha) = \begin{pmatrix} \cos\alpha & \sin\alpha & 0 & 0 \\ -\sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

(c) Rotation around z-axis

$$S = \begin{pmatrix} \alpha & 0 & 0 & 0 \\ 0 & \beta & 0 & 0 \\ 0 & 0 & \gamma & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

(d) Scaling in all directions with scalars α, β and γ

$$T = \begin{pmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

(e) Translation in all directions

Figure 2.5.: 3D transformations with matrices in OpenGL ES.

Every matrix operation manipulates the currently active model-view matrix. This is a problem, since all object transformations have to be based on the original model-view matrix, otherwise transforming an object would affect the previously drawn objects. Therefore, the current model-view matrix has to be stored so that the model-view matrix must not be created multiple times. Luckily, OpenGL provides functions to save and restore the currently active matrix.

Texturing Texturing a surface is an essential part of every advanced graphics application as it allows to map all kind of textures to an object instead of just displaying its wireframe representation or simple colors. Two different texturing methods are available in OpenGL ES: Cubemap textures, which are not used as part of this project, and 2D textures. A two-dimensional texture consists of texels, which represent the individual data elements of the image, and generally - there are other representations, as well - holds a red, green, blue and alpha value. Just like the geometry of an object, textures also have their own coordinate space, ranging from $(0, 0)$ to $(1, 1)$. These so called uv-coordinates make it possible to map any portion of the texture to vertices of an object. Figure 2.6 shows the normalized uv-coordinate system, as well as a schematic mapping example.

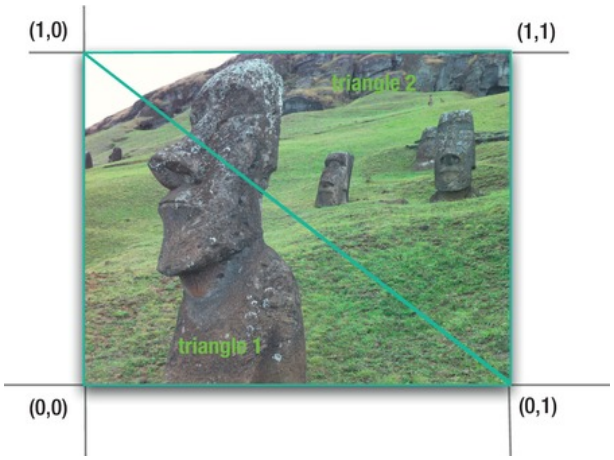


Figure 2.6.: Mapping a 2D texture to a rectangle consisting of two triangles. [29]

Two steps have to be performed in order to map a texture to a polygon (a polygon is a group of vertices): First, the texture coordinate - vertex combinations must be specified in a float array and then, similar to creating the object's geometry, translated into a ByteBuffer. The buffer only holds the coordinate data of the image parts, the connection to the vertices is allocated by their position in the buffer (2 float or 2*4 byte values are connected to one vertex). The second step is telling OpenGL ES which image to use as texture. The state machine is capable of using Bitmaps or other image representations directly for texturing and already implements base functionality to load a texture in the content pipeline which holds all resource data. To make use of the texture, OpenGL creates a unique ID and returns it for further interaction with the texture. Note that in version 1.0, 1.1 and 2.0 a non-power-of-two texture size is not possible.

Finally, some drawing options are missing. Graphic programmers have to specify the texture filtering methods minification and magnification and their parameters. Minification is applied when screen pixels are larger than one texel of the image - the texel is then scaled up - magnification works the other way round. Linear interpolation is the right choice for smooth rendering output, but other parameter values are

possible as well.

There are many other options that influence lighting, blending options, the shading model and further texture filtering settings which are explained in the implementation part of the thesis, if they are used. [27][29][30]

Part II.

Editor and Game Implementation

System Design

3.1 System Requirements

Requirement analysis plays an important role in software engineering and is fundamental for the development process. Thus, this section lists the requirements the system has to meet. The functional requirements - these specify a particular result of the system - are listed in the form REQ-X and the corresponding non-functional requirements - they describe the system characteristics - are listed in the form REQ-X.Y. The analysed requirements for this thesis are divided in three different categories: Editor, Game and Editor & Game requirements.

3.1.1 Editor Requirements

REQ-1 The editor is able to create new games and manage already created games.

REQ-1.1 The user can input information about himself and the game, especially a name for the game is essential. Additionally he can select the orientation of the game and select an icon for the game.

REQ-1.2 The input fields for all information do not allow special characters and the input field for the game name has to be empty.

REQ-1.3 All game information is not final and can be changed at any time.

REQ-1.4 The user can delete an existing game. To prevent accidental deletion, a dialog is shown and the user has to confirm his action.

REQ-2 It is possible to add new levels to the game and delete levels from the game.

REQ-2.1 The levels are identified by their number and a screenshot of the scene

REQ-2.2 Adding a level automatically inserts it at the end of all levels and names it accordingly. The standard screenshot image is solid black.

REQ-2.3 Changing the sequence of the levels is not possible.

REQ-2.4 The user can delete an existing level. To prevent accidental deletion, a dialog is shown and the user has to confirm his action.

REQ-2.5 Removing a level except the last one in the list changes the internal numbering of the levels. The levels' numbers subsequent to the deleted level are decreased by one to maintain the correct sorting.

REQ-2.6 Deleting a level removes all its objects and triggers from the database. Additionally, the screenshot is deleted from the storage device.

REQ-3 The user can change the physical world of the game in scene editor and manipulate it.

REQ-3.1 The scene editor indicates the level boundaries by a green box and the target device's display rectangle with a red box.

REQ-3.2 The level minimum level size is equal to the camera rectangle of the target device, the maximum size is 100 meters.

REQ-3.3 Changing the level size is only possible in "Editing Mode".

REQ-3.4 Changing the level size is immediately applied and the database is updated. This can lead to out of bounds objects. The user has to confirm a dialog to proceed with the action and delete the corresponding objects or cancel the dialog to keep the size unchanged.

REQ-4 The user can place objects from his favorite list in the physical world, remove them and manipulate their size, position and rotation.

REQ-4.1 Placing objects is only possible in "Placing Mode", whereas editing is only possible in "Editing Mode". The editor has a control element to switch between these two modes.

REQ-4.2 The rotation of an object lies between 0 and 360 degrees. The maximum size matches the level size, the minimum size is 0.1 meters. The object can be positioned everywhere in the level, but its dimensions always have to be within the level bounds.

REQ-4.3 Newly added objects have a standard size based on their category.

REQ-4.4 Objects added to the scene are immediately stored in the database, removed objects on the contrary are immediately deleted from the database. When changing the object's attributes, an update to the database values is made at the end of the action.

REQ-4.5 The favorite list is grouped into categories for better organisation.

REQ-4.6 Only one player object is allowed in the game at the same time.

REQ-4.7 The physics engine is able to handle at least 50 objects at the same time.

REQ-5 The editor offers functionality to add events to the game and also to remove events. The trigger and functionality of the event can are selected separately.

REQ-5.1 An event consists of exactly one trigger and functionality.

REQ-5.2 Triggers concerning lives and points only allow non-negative numbers as input.

REQ-5.3 Events concerning objects in the level only show valid objects. Therefore deleted objects in the level are also removed from all events.

REQ-5.4 The user can select objects for triggers and functionality in an object picker which shows the whole level. Clicking the object assigns it to the trigger or functionality.

REQ-5.5 It is obvious which events have adjustable settings and if these have already been edited.

REQ-5.6 The description of a trigger and functionality is obvious, too.

REQ-6 The editor allows the user to select favorite objects which he can place in a level from a large gallery.

REQ-6.1 The gallery and favorites are divided into different categories to give a better overview on the objects. The naming conventions for all objects clearly point out their purpose.

REQ-6.2 The favorite data is linked to the gallery so that both show the same categories at any time.

REQ-6.3 An object item consists of a name and a texture.

REQ-6.4 Adding and removing favorites is performed by clicking on the item in the corresponding list.

REQ-6.5 Duplicate objects cannot be added to the favorite list.

REQ-7 The scene editor makes extensive use of touch controls to reduce the amount of buttons and other control elements.

REQ-7.1 Depending on the mode the scene editor is in ("Placing mode" or "Editing mode") it offers different gestures. Independent from the mode, the editor offers a gesture which shows a navigation menu to switch between all available sub-editors.

REQ-7.2 "Placing mode" offers gestures for scrolling around the level, zooming and placing objects.

REQ-7.3 "Editing mode" offers gestures for scrolling around the level, zooming, selecting objects, moving objects and resizing objects.

REQ-8 The user can export a created game which is then automatically compiled and copied to a specific output folder.

REQ-8.1 The created APK file is signed with debug key.

REQ-8.2 Exporting is fully automated and no user input is required.

REQ-8.3 While exporting, the editor indicates the progression of the process.

REQ-8.4 Errors in the export process are displayed to the user with a description of the error.

REQ-8.5 The system identifies only the necessary data for the game to achieve a minimal package size.

3.1.2 Game Requirements

REQ-9 The game is able to detect the device resolution and adjust the graphics accordingly. The detection is initialised automatically when the game is started.

REQ-9.1 The device resolution is recognised reliably and does not require any user interaction.

REQ-9.2 Object and texture dimensions are preserved and neither of them are deformed when using different resolutions.

REQ-9.3 Adjusting the resolution is not affecting the performance of the game.

REQ-10 The game provides different control methods which allow the user to interact with the physical world of the game.

REQ-10.1 The controls are the same on every device, meaning that yaw, pitch and azimuth are oriented to the device's coordinate system the same way.

REQ-10.2 User input is not delayed and directly affects the game.

REQ-10.3 The game controls are self-explanatory and easy to learn, even without instructions.

REQ-11 The game has a standard main menu to start a new game, exit the game and view game informations like developer, description and publishing date. The HUD (**H**ead-**u**p **d**isplay - a transparent display presenting data as an overlay in the game scene) informs about the player's current points and lives.

REQ-11.1 All main features of the game are accessible with just one button click.

REQ-11.2 The graphical user interface of all menus is standardised and locked to portrait mode.

REQ-11.3 The orientation of the game is declared in the editor, but is then locked to this particular setting.

REQ-11.4 The HUD is always up-to-date and refreshes itself on certain events.

REQ-11.5 The HUD only displays non-negative numbers for lives and score.

REQ-12 The user can start a new game and play all available levels created with the editor.

REQ-12.1 The levels of the game are arranged in a fixed chronological order.

REQ-12.2 Completing a level automatically presents a screen with the scored points and loads the next level.

REQ-12.3 If the last level is completed the users highscore is calculated and displayed. After that the game returns to the main menu.

REQ-12.4 Completing a level automatically loads the next level.

REQ-12.5 If the player has no lives left, the game is over and returns back to the main menu.

REQ-13 The editor can persistently store objects placed in a scene, game and level data.

REQ-13.1 Objects are immediately stored and updated on placement and manipulation.

REQ-13.2 The database tables which store the data have minimal *null* values.

REQ-13.3 All data is stored application specific and deleted on uninstalling.

3.1.3 Editor & Game Requirements

REQ-14 Both, game and editor implement a physics engine to simulate a world with correct physical interaction of the objects.

REQ-14.1 The physics engine is designed specifically for two-dimensional simulations.

REQ-14.2 The physics engine runs fast and smooth for a reasonable amount of objects.

REQ-14.3 The engine allows rectangle and circle shaped objects.

REQ-14.4 The user can influence the gravity of the world physics if the corresponding control methods is activated.

REQ-15 Both, game and editor implement a simple graphics engine to display objects on the screen.

REQ-15.1 The graphics engine is limited to a two-dimensional orthographic view.

REQ-15.2 The graphics engine runs fast and smooth for a reasonable amount of objects.

REQ-15.3 The engine is configured to display arbitrary objects.

3.2 Data Model

The data representation model of the project is designed to store editor and game objects as well as events and convenient data such as user favorites. All data tables except the game objects are populated dynamically while using the application. This design follows the rather simple approach to supply premade objects instead of creating new ones by the user. A more complex approach was not realised due to the limited time of the thesis.

Storing all the data in a database has many advantages. A database is very flexible for future enhancement, e.g. when adding additional objects or completely new editor components and their respective functionality such as a sound editor. Additionally, Android's *SQLiteOpenHelper* and *SQLiteDatabase* classes provide fast and easy access to store persistent data. Especially the possibility to add, update and delete one or many records at a time justifies their implementation. Furthermore, selecting objects from tables based on given parameters and joining table rows is also a crucial part of the editor and therefore underline the database approach as base for the data model.

The exported games use a slightly different approach with a combination of database tables and translated Java code. Similar to the editor, all game objects are stored in a database table (actually in the same table but only with records of the exported game). The only difference is that the events created for a game have to be translated to Java code and then included in the compilation process, otherwise it is not possible to make use of them in the game cycle.

The Entitiy-Relationship model, **ER models** are an abstract way to describe databases and often used for conceptual design, figure 3.1 illustrates the data organisation of the game editor. Section 3.2.1 and 3.2.2 describe the entities and relationships of the diagram in more detail. Note that the actual implementation is not necessarily similar since the ER model has to be translated into a relational database scheme. In the course of this process all relationships between the entities are resolved and integrated in the database tables.

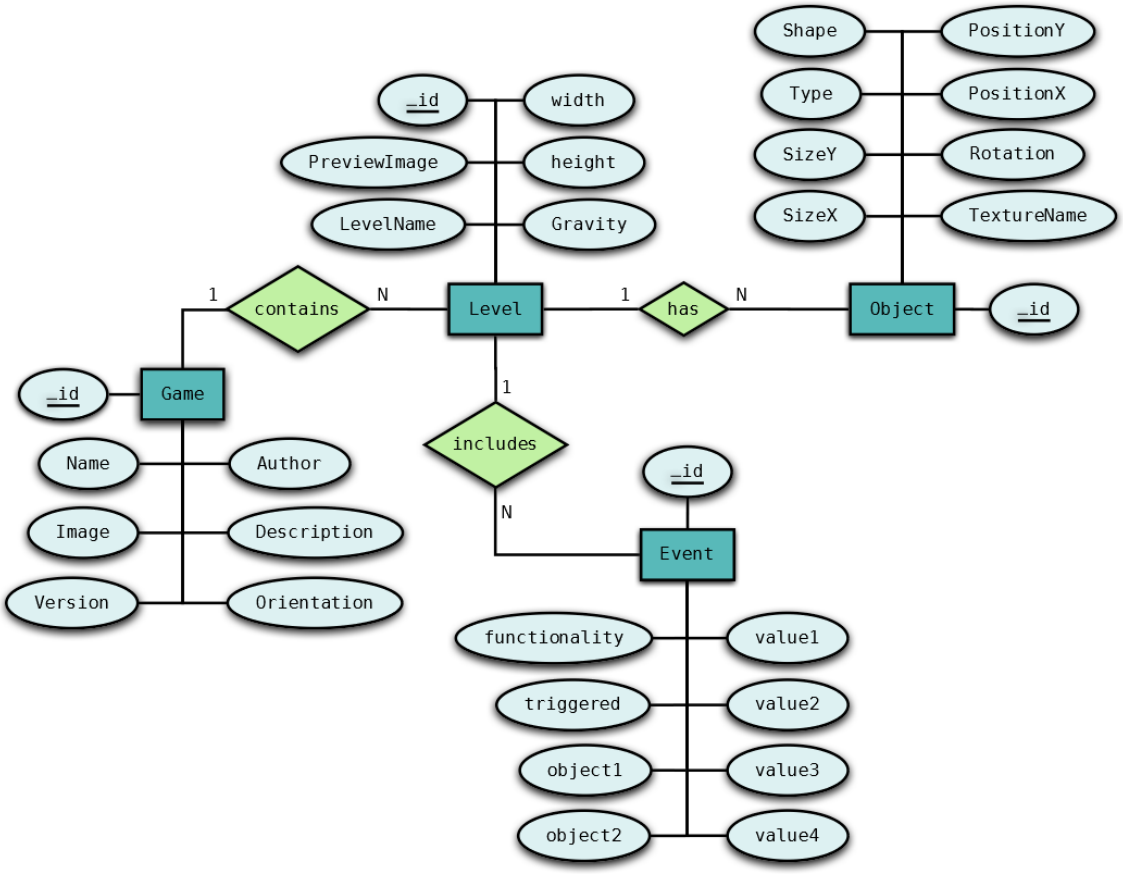


Figure 3.1.: The data model of the game editor with all components.

3.2.1 Entities in the data model

Game The *Game* entity type represents a single game created by the editor. It stores all user and game data, such as *author* (String), *description* (String) and the game's *Name* (String). A *Game* is uniquely identified by its *_id* (Integer, primary key). Most of this data is not only used in the editor, but also designated to be displayed in an exported game.

Level The *Level* entity type holds data concerning a single level instance. *PreviewImage* (String) and *LevelName* (String) help the user to recognise a *Level* in the overview list, *Gravity* (Integer), *width* (Float) and *height* (Float) specify its dimensions and physical background. A *Level* is uniquely identified by its *_id* (Integer, primary key).

Object All objects in a *Level* are stored in the *Object* entity. Its current state including position, size and rotation is defined in the corresponding attribute fields, to name just a few *PositionX* (Float), *SizeY* (Float) and *Rotation* (Float). Additionally, information about the graphical and physical representation are stored in this entity. The *TextureName* (String) field stores the path to the texture of the object, *Shape*

(Integer) - this identifies if the object is a rectangle or a circle - and *Type* (Integer) which specifies whether the object is (non-) physical. The *Version* (Integer) field stores the version number of the latest export of the game. An *Object* is uniquely identified by its *_id* (Integer, primary key).

Event To script game events the *Event* entity has significant benefits. An *Event* consists of a triggering action and a functionality which is started by the the trigger. They are stored in the *triggered* (Integer) and *functionality* (Integer) fields. Some triggers and functions are associated with objects in the level. Instead of introducing a new relationship, the concerned object's *_id*'s are stored as attributes (*object1* (Integer), *object2* (Integer)) of the *Event* to simplify the query structure of the event editor. The value attributes of an *Event* - from *value1* (Integer) to *value4* (Integer) - also store information for triggers and functions such as point values or rectangle coordinates. As well as all other entities, an *Event* is uniquely identified by its *_id* (Integer, primary key).

3.2.2 Relationships in the data model

contains The *contains* relationship between *Game* and *Level* is used to assign levels to a game. A game can consist of multiple levels at a time, but a level can only be part of one game.

has This relationship between *Level* and *Object* assigns objects to be part of a level. A level can contain multiple objects at a time, but an object can only be assigned to one level.

includes Assigning events to a level *includes* relationship. An *Event* is always only part of one *Level*, whereas a *Level* can include several events.

3.3 System Architecture

Now that all requirements for the application and the data model behind the editor are defined, the system architecture and the design intention are described in more detail. First of all the underlying design pattern is explained, then a structural analysis of the editor and the game part is presented.

3.3.1 Model-View-Controller Pattern

Apart from Android's ordinary activity design and program flow, this project is based on the MVC (**M**odel **V**iew **C**ontroller) pattern which clearly separates the presentation (View), logic (Controller) and data (Model) part. It was first introduced by Trygve Reenskaug, a Smalltalk developer, in 1979 and is one of the most successful design patterns in software engineering.

The reason for the distinction in these three subsystems is that the user interface naturally changes more often than the program logic and the data system of the application. Therefore, closely tied components are hard to maintain and change. Additionally, this design allows to build and test the model independent from its graphical representation. The MVC pattern greatly simplifies the complexity of the system and increases maintainability as well as extensibility. Most of the implemented classes can be reused in other projects or even, with slight changes, in the same one. It can be used in both, desktop and web applications. Usually, every object in a project should be part of one of these subsystems: [6][9][24]

Model A model is composed of different attributes and responsible for storing data. Moreover, it provides methods for object manipulation and rules for processing. In an ultimate implementation, nothing but the combination of all models encapsulate the application state. However, in most scenarios some application data is located in a controller. [6][9]

View The view subsystem requests the data stored in a model and renders the output for the user. Additionally, it provides functionality to interact with the data model. It is possible to have multiple views for the same data model which generate different output for the user interface, for example a command line or graphical representation. [6][9]

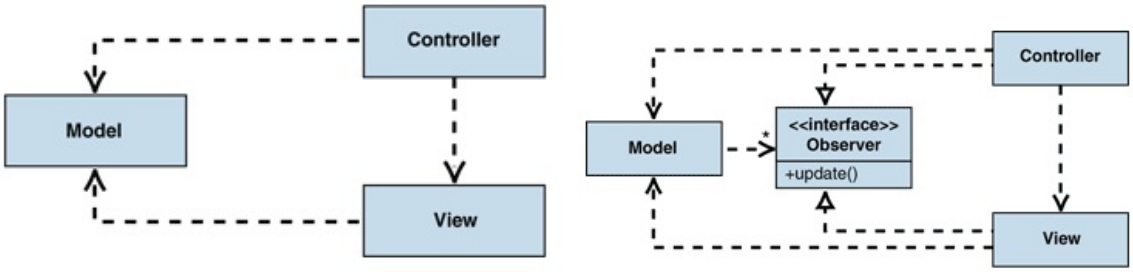
Controller A controller's primary task is to define the application behaviour and map the user interaction with the view subsystem to data model manipulations. Preparing the model data for representation in a view is also part of the controller, for example sorting an unordered list in the model which is then rendered and shown to the user. The controller decouples the model and view subsystems and allows to change or completely replace them easily. [6][9]

Figure 3.2 shows the dependencies between the subsystems model, view and controller. Note that the view as well as the controller subsystem depend on the model, but the model is not connected with them at all. This design results in the above mentioned separation of the application data.

Most of the structural organisation in this project is based on the simpler approach shown in figure 3.2(a). Contrary to the design shown in figure 3.2(b) which additionally uses the Observer pattern, commonly known as *publish-subscribe*, the model does not notify the other subsystems on data changes. Its purpose is to notify objects about a state change without establishing an interdependency of the objects involved. In some applications for example, the view subsystem has to be informed about state changes in the model and refresh the visual representation. [24][25]

3.3.2 Editor Design

Providing a clear and simple user interface is a key feature of every application and is a main goal of this project. A fundamental part in the planning and implementation is the



(a) Model-View-Controller class structure [24]

(b) Using the Observer pattern to decouple model and view [24]

Figure 3.2.: Different implementations of the MVC pattern

arrangement and layout of the editor's activities. Every activity is specialised and responsible for one main task, for example creating events or building the level's scene. Figure 3.3 outlines the layout of the editor and gives information about the connection of the editor's components.

The root of the layout and entry point of the application is the *StartScreen activity*. It offers GUI elements to create a new game (1), load any existing game (2) or resume the lastly created one (3). Each of the three elements starts another activity on activation. Option 1 starts the *GameSetup* activity which is able to create new games and also change an existing game, which is explained later. Option 2, namely loading a game, starts the *LoadGame* activity. Not only loading is handled by this part of the editor, but also deleting games can be initialised. The last leaf of the root is the *EditorLevel* activity and corresponds to option 3, resuming a game. This part is responsible for handling all the levels in a game and also has a connection to the *GameSetup* for changing a game's data. Note that both, loading a game and creating a new one start this activity.

3.3.3 Game Design

In order to create the graphical user interface and activity flow of the game several other games in the Google Play Store have been analysed, such as "Temple Run", "Angry Birds" and "Inotia 3". However, harmonisation of an universally usable and customisable design could not be put into practice due to time limitations. Because of that, the design has been based on activity diagrams.

The statechart diagram in figure 3.4 - statechart diagrams are a graphical representation for process organisation and part of the **U**nified **M**odeling **L**anguage (UML) - illustrates the structure and program flow for every created game. In the main menu the user can choose between three options:

- The "About" item allows him to take a look at the description of the game, the developer and other info,
- The "Exit" item quits the application,

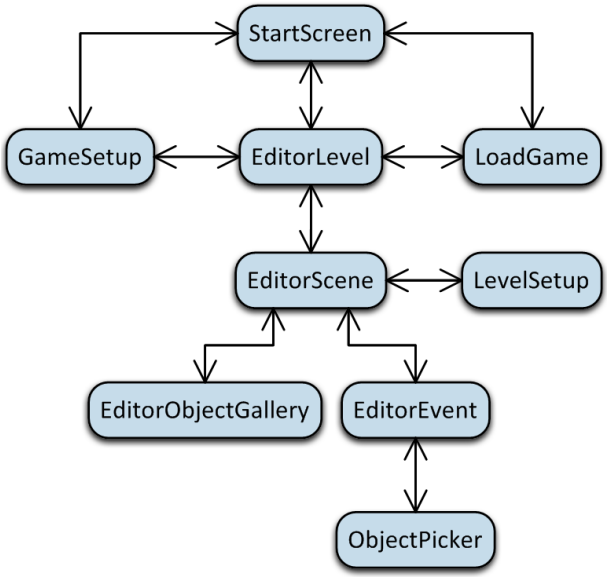


Figure 3.3.: Game editor layout and interaction between the components of the editor.

- The "Start Game" item initialises a new game and starts it at the beginning.

The most interesting part is certainly the "Start Game" option as it results in a more complex activity flow than the other options. All actions and events in the game can be specified in the game creation process of the editor and allow varying level implementations, thus instead of going into detail only a rough structural overview of the activity flow is presented in figure 3.4.
Once a game is started, the application loads all necessary game data and the user can start playing. If he does not want to continue playing, it is possible to return to the main menu with the device's back button, otherwise he has to play until he reaches the specified level goal. When he completes a level a score screen is shown, indicating how much points have been scored in the last level. Tapping on the screen proceeds the game to the next level loaded by the application or, in case the end of the game is reached, a highscore screen is displayed. Of course a game can also end prematurely, for example if the player dies to often or all attempts are consumed. In this case a game over screen is shown and tapping on it will return to the main menu.

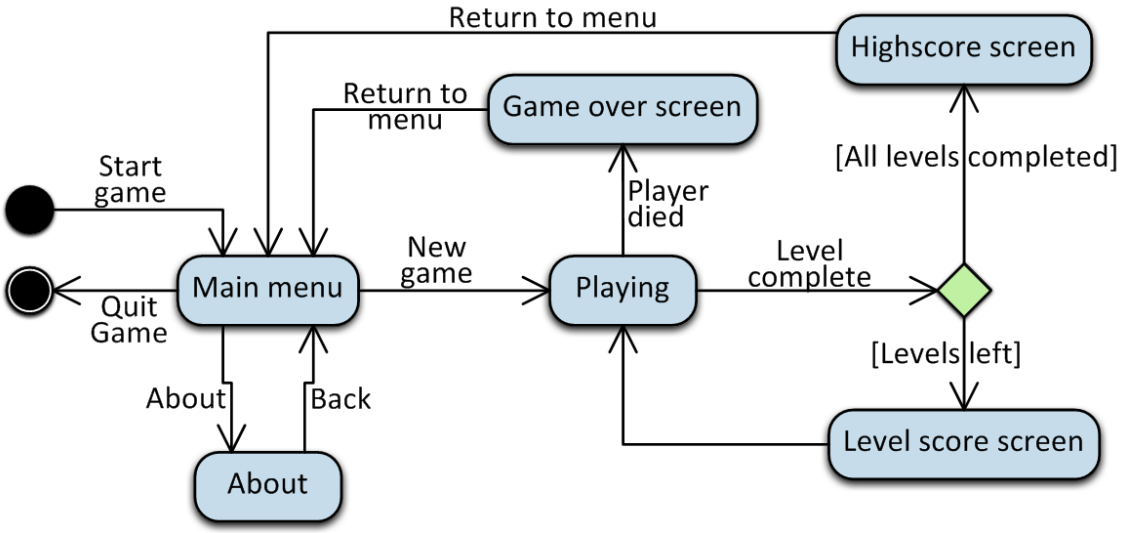


Figure 3.4.: Complete state machine diagram of every game created by the editor.

Implementation

4.1 Graphics engine

The graphics engine is a fundamental part of each game and can be arbitrarily complex. With respect to the short processing time of this bachelor thesis, writing an own implementation of a complete graphics engine would go over the top. One possibility to work around this problem would be to integrate an already existing engine for Android platforms, such as AndEngine, or to implement only the absolutely necessary parts of a 2D engine with minimal graphics settings. By weighing the pros and cons, the following section gives a short overview over both approaches and points out, why an own implementation of the engine has been realised.

4.1.1 Complex versus lightweight implementation

Integrating a premade graphics engine is accompanied by large benefits, especially regarding complexity and adjustability. The already mentioned AndEngine for example ships with support of automatical device scaling, Box2D physics extension, particle system controller and much more. But on the other hand, the complexity is also the biggest problem of this approach. Most of the mentioned features are not needed or supported, neither in the game editor, nor in the created game. The functionality overhead also results in greater training time and therefore decreases the time advantage compared to the proper implementation.

Developing a solution of one's own which is broken down to only the absolutely necessary features, such as displaying two dimensional objects with an orthographic view, can be implemented nearly as fast as learning to handle foreign code and additionally provides better overview and control.
Finally, device limitations are not negligible and tip the balance towards a self-created graphics engine. At first glance an already existing engine seems to satisfy perfectly the requirements for a fast and efficient solution, but this is only true for executing a compiled application. The game editor as well as the created game, considered in isolation, match this scenario, but the problem lies in between. Due to the memory limitation of an application it is not possible to link and compile projects on the device itself, which exceed a certain dimension. Chapter 4.8.3 Memory limitation on Android devices on page 63 goes into more detail about this issue on mobile devices.

All in all the chosen approach is a more reasonable way to combine fast accessibility, sufficient settings and above all, lightweight functionality for mobile devices with limited resources.

4.1.2 Render settings

The largest part of the view component is implemented in the graphics class of the project and is responsible for rendering all objects currently in the game. Like every other controller, the graphics also uses the singleton design pattern and thus only allows a single instance active at any time. Graphics class only contains one noteworthy member variable, the camera. Its functions are explained in the next section.
By implementing the Renderer interface from the OpenGL graphics library, the three main components *onSurfaceCreated(GL10 gl, EGLConfig config)*, *onDrawFrame(GL10 gl)* and *onSurfaceChanged(GL10 gl, int width, int height)* are provided for implementing the drawing logic.

OnSurfaceCreated(...) is called once at surface creation and responsible for all permanent rendering settings like depth testing, shader model and lighting mode. Since editor and game are only designed to be a two dimensional orthographic view, only a few settings are sufficient to initialise the OpenGL interface. Perspective correction is set to the GL_NICEST filter function which results in perspective correct interpolation of textures and colors and therefore minimizes computed fragments. Additionally, blending is enabled and the blend function is defined as GL_SRC_ALPHA (src), GL_ONE_MINUS_SRC_ALPHA (dest) to simulate correctly alpha blending.

After the surface is created and each time the surface changes, for example when the device is rotated or the surface view changes its dimensions, *onSurfaceChanged(...)* is called. Since the global coordinate system has its origin in the bottom-left corner, rotating the device as well as manipulating the size of the surface view will change width and height. Therefore, all camera settings have to be changed. This includes adjusting the viewport size to match the surface size, plus specifying the viewing frustum. As the physics engine is based on the metric system and uses meters as a unit of length, this is adapted here and frustum dimensions are also represented in meters. Conversion between pixels, which is the input from surface size, and meters is calculated by the formula: $Meter = Pixel \ / \ MeterToPixelFactor$. The only change to the OpenGl state machine in this method is setting the viewport to the newly given width and height of the surface.

All remaining functionality of this method serves the purpose of displaying the screen size of the target device in the scene editor and will not be explained in detail.

Now that all adjustments to the OpenGL interface are made it is time to draw something. The *onDrawFrame(...)* method runs on its own thread and is called multiple times per second. In each drawing step all camera settings have to be loaded, as described in section 4.1.3 on page 35. To avoid thread exceptions when objects are added to the game object list, this is because OpenGl runs on a different thread as the main UI thread and is not synchro-

nised, all objects have to be stored in a temporary buffer and then loaded into the actual game object list in the OpenGl thread, which is done by calling *managePendingObjects()*. Thread safety is always a big problem when OpenGL functions are called from other threads or when objects in two threads interact. Another problem when using OpenGL on mobile devices is, that the context of OpenGL is only valid if one of the three main methods of the OpenGL interface are executed. Accessing OpenGL functionality from outside these methods will always lead to an exception and crash the application. There are two different approaches to deal with these problems: it is either possible to add a Runnable object to the respective thread with a *Handler* and its *.post(Runnable r)* method and specify all actions there, or to provide flags for all actions which then trigger the addressed functions in the next iteration step of the thread. Sometimes even both options have to be used in order to work correctly. An example is the *takeScreenshot()* function in the graphics class, which takes a screenshot of the currently rendered image in the surface view of the editor. First of all, the context has to be active, so the screenshot flag in graphics has to be set. In the next rendering iteration, the *onDrawFrame(...)* function then takes the screenshot and a new *Runnable* which finishes the scene editor activity is posted to the main UI thread.

The actual drawing code can be divided into two parts, drawing with textures and drawing just the outer bounds of a shape, their bounding boxes. For both scenarios the OpenGl state machine has to be adjusted to draw with the correct settings. Before drawing either of them, the appropriate options are set once to avoid constant rearranging of the graphics pipeline and therefore optimise GPU and memory usage.

Listing 4.1: Setting options for drawing objects with textures.

```
1 gl.glEnable(GL10.GL_TEXTURE_2D);
  gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
3 gl.glEnableClientState(GL10.GL_TEXTURE_COORD_ARRAY);
  gl.glColor4f(1f,1f,1f,1f);
```

Code example 4.1 illustrates textured drawing. At first, OpenGl is configured to draw two-dimensional textures and is also told to accept vertex arrays and texture coordinate arrays when drawing objects with *glDrawElements*. Additionally, the blending color for all following drawing calls is set to solid white which is necessary to disable external color influence to the object texture. Each pixel color of the texture is multiplicated component-by-component with the color set by *glColor* or variations of this method. White as four-dimensional vector (1,1,1,1) therefore does not alter the texture and keeps every pixel color as is:

$$C = (1 * r, 1 * g, 1 * b, 1 * a) \tag{4.1}$$

Next, a loop which iterates over all game objects calls the rendering code of each object separately. The *drawObjectWithAnimationOrTexture(DrawableObject d)* function shown in listing 4.2 on the next page is responsible for object positioning, scaling and rotation and arranges each object correctly in the world space. For this purpose OpenGL provides special functions to manipulate the active model-view-projection matrix.

Listing 4.2: Translating rotating scaling and finally drawing of a texture.

```
private void drawObjectWithAnimationOrTexture(DrawableObject d){
    gl.glPushMatrix();
    gl.glTranslatef(d.getPosition().x, d.getPosition().y, 0);
    gl.glTranslatef(d.getSize().x / 2, d.getSize().y / 2, 0);
    gl.glRotatef(d.getRotation(), 0, 0, 1);
    gl.glTranslatef(-d.getSize().x / 2, -d.getSize().y / 2, 0);
    gl.glScalef(d.getSize().x, d.getSize().y, 0);
    d.draw(gl);
    gl.glPopMatrix();
}
```

To ensure that every object has exactly the same base matrix, the standard model-view-projection matrix created by the camera is pushed onto the matrix stack and therefore stored for the next objects without any change. After that, all following matrix operations are only applied to one single object. In general, matrix multiplication is not commutative and therefore all operations have to be applied in a strict order. Note that all operations have to be called in reverse order to produce the correct output. Since the bottom left corner of each shape is defined as object origin, it can be directly scaled to its size (the original size of each object is *Vec2(1,1)*) . Next up, the object's center is translated to the world space origin by subtracting half its size from the current position. Now that world and object origin share the same point, rotation around the z-axis spins the object right in place without any offset. Then, the translation mentioned two steps earlier has to be reverted. This is absolutely necessary because it only serves the purpose of rotating the object around its center point. A last translation moves the object to its position and finishes all matrix manipulations. Last but not least, the objects drawing method is executed.

Even though the project underlies the MVC pattern, it is only applied weakened here. Instead of realising the drawing in the graphics class, the OpenGL context is passed to the currently processed object which then sends all necessary data to the graphics pipeline itself. Every object consists of a shape with customised code to match perfectly the texture with its boundary. For more information about object and shape organisation see chapter 4.6.6 Object editor, on page 55. Listing 4.3 shows the drawing code for the rectangle shape.

Listing 4.3: Drawing method of the rectangle shape.

```
public void draw(GL10 gl)
{
    //bind texture
    gl.glBindTexture(GL10.GL_TEXTURE_2D, textures[0]);
    //specify the location and data format for rendering and draw
    gl.glVertexPointer(2, GL10.GL_FLOAT, 0, vertexBuffer);
    gl.glTexCoordPointer(2, GL10.GL_FLOAT, 0, textureBuffer);
    gl.glDrawElements(GL10.GL_TRIANGLES, indices.length, GL10.
        GL_UNSIGNED_SHORT, indexBuffer);
}
```

Data passed to OpenGL has to match the previously activated states and therefore include texture binding, specifying all vertices and texture coordinates. Finally, OpenGl is instructed to draw all vertices which are defined as triangles in the indices array. At this point, textured drawing is complete and all activated OpenGL options can be deactivated again. Only vertex array drawing is left active for rendering bounding boxes and screen rectangles.

In code example 4.2 one essential line was left out. To restore the model-view-projection matrix before manipulating it, *glPopMatrix()* has to be called. It pops the top of the matrix stack and makes sure that all matrix operations of the following objects have the same base matrix.

Up to now, only the term bounding boxes have been mentioned, although there are actually no bounding boxes in the scene editor. This is because the object's bounding box is not used as such, but utilised as selection indicator. Just like textured rendering, drawing the bounding boxes is realised by the exact same matrix operations. The only difference lies in the rendering part. Bounding boxes do not have a texture but only consist of colored lines. Instead of defining a color for these lines in every rectangle and passing these additional values to the graphics pipeline, the current drawing color is set to a specific value, in case of the bounding boxes completely red. Multiplying the set color with the standard color of a vertex, which is always white, will result in displaying the set color (see equation 4.1). To draw the box, the object's vertex buffer pointer has to be passed to OpenGl again. Rather than creating a box out of triangles, this time a line loop is used. Thus, it is possible to use the exact same vertex array as for textured drawing. The function *glDrawArrays(GL10.GL_LINE_LOOP, 0, 4)* then tells the renderer to create a box out of four elements.

The remaining part of the *onDrawFrame(GL10 gl)* function is less interesting, because it only creates two additional bounding boxes which are stored in the graphics class. These indicate the level boundaries and the camera rectangle size in scene editor. The lastly mentioned one shows the camera clipping of the target device. Information about level size is stored in the *LevelInstance* class, target device size and camera rectangle position are defined in the camera class and explained in the following subsection.

4.1.3 Camera

After defining all render settings a camera has to be created. As specified in the requiremets (REQ-15.1 on page 24), the camera is limited to a two dimensional orthographic view and contains position, zoom factor, viewport and frustum size as well as information about the camera rectangle which indicates the screen size of the target device.

All camera settings have to be loaded in every draw cycle, for that reason the camera class provides the *loadCameraSettings(GL10 gl)* function which handles all necessary actions to prepare the OpenGL interface for drawing with the correct settings. First of all the depth and color buffer are cleared and the backbuffer color is set to black. The backbuffer

is part of the framebuffer and instead of drawing to the screen directly which would cause flickering, everything is first drawn into the backbuffer then displayed when the rendering and drawing is finished. After that the viewport, this is the part of the screen where the drawing happens, is set. Next, the projection matrix is set and initialised with the identity matrix. It is crucial to not combine previous transformations matrices with the current one.

Finally, the call of *glOrthof(...)* produces a parallel projection where the position of the camera is the center of the viewing frustum. Depending on the zoom factor the left, right, top and bottom clipping planes increase or decrease the size of the viewing projection. Near and far clip do not change and are set up to show everything between -1 and 1 on the z axis. These settings result in a two-dimensional side view, where the z-coordinate of a polygon has no effect on the size of the displayed projection. Especially side-scrolling and platform games prefer this setup.

Listing 4.4: Loading the camera settings for drawing with a two dimensional parallel projection.

```
1  public void loadCameraSettings(GL10 gl) {
       //Clears the screen and depth buffer:
3      gl.glClear(GL10.GL_COLOR_BUFFER_BIT | GL10.
           GL_DEPTH_BUFFER_BIT);
       gl.glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
5
       //set iewport, reset projection martrix, select mode matrix
7      gl.glViewport(0, 0, (int)viewportWidth, (int)viewportHeight);
       gl.glMatrixMode(GL10.GL_PROJECTION);
9
       gl.glLoadIdentity();
11     gl.glOrthof(position.x - frustumWidth * zoom / 2,
            position.x + frustumWidth * zoom / 2,
13          position.y - frustumHeight * zoom / 2,
            position.y + frustumHeight * zoom / 2,
15           -1.0f, 1.0f);
       gl.glMatrixMode(GL10.GL_MODELVIEW);
17 }
```

Section 4.1.2 already mentioned the screen rectangle indicator of the target device. All information about this rectangle is stored in the camera class. The target device size is set in the game settings and optimised for small smartphone screen resolutions. Additionally, the size depends on the currently active screen orientation of the game (a size of (480, 800) in portrait mode is changed to (800, 480) if landscape mode is activated). The extra position variable for the screen rectangle is crucial for correct positioning, depending on the level bounds. The camera is configured to move freely through level space without any barrier, but this is not the case for the screen rectangle which is kept inside the level boundaries at any time.

4.2 Physics engine

A physics engine introduces the laws of physics to the application with the purpose to appear more realistic. As with the graphics engine, a careful consideration concerning the implementation approach was made and resulted in choosing an already existing physics engine, the JBox2D physics extension. The complex physical calculations which are hard to implement in a short period of time, as well as its ease of use and fairly good performance where crucial for this decision. However, when this thesis was nearly completed it turned out that there are problems with heap space when compiling on the device. Since this physics engine is very large compared to implementing an own solution, it is necessary to overhaul the physics part if the application gets improved in the future to make sure the device provides enough resources. This issue is addressed further in section 4.8.3 Memory limitation on Android devices on page 63.

JBox2D is a Java port close to the the popular C++ based Box2D physics engine available for multiple platforms including iOS, XBox360 and Windows Phone. Basically, "Box2D is a rigid body simulation library for games" [7]. The source code is provided as .jar file and imported to the project as external library. Additionally, the SLF4J Android logging framework, also provided as .jar and imported as external library, has to be included in order to make JBox2D work correctly.

In the following the main components of the engine and their used functionality are described briefly to provide basic knowledge of the physics engine. For further explanation see the reference manual [7].

World The world is a collection of bodies, fixtures, constraints and joints. All objects in this container class can interact with each other. Running and managing the simulation is also handled by the world with parameters such as step duration and position iterations per update.

Shape A shape describes the basic collision geometry and is used for collision detection. There is only one shape template for a circle but it is also possible to create own shapes like rectangles or other multi-polygon shapes.

Fixture To attach a shape to a body, the fixture class is used. It also holds additional non-geometric data such as friction, density and restitution to make the simulation of an object more realistic.

Body Simulating object movement in the world is possible with the body class since it holds position and velocity data. Applying forces, torque and impulses to a body is also possible. There are different types of bodies, two of them are actually used in the game editor: On the one hand there is the static body which does not move and has a fixed position, on the other hand the dynamic body is fully simulated. It can collide with all body types and moves according to the applied forces.

Contact For every two overlapping axis-aligned bounding boxes (AABB) - these correlate to their fixtures - a contact exists and holds all contact points. The calculation of the

contact points is based on the shapes of the involved fixtures (for example circle-circle or polygon-polygon collisions). This functionality is the essence of the contact class, please see the manual for further description.

Based on this short introduction into the terminology of the physics engine, the essential parts of the *Physics.java* class are outlined. Instead of only using the physics engine in an exported game it is also implemented in the game editor. Due to the fact that both of them require a representation of physical objects and customisation of different solutions is very time consuming, the physics engine is also activated in the game editor but simulation is paused to prevent objects from moving (4.5).

Listing 4.5: Creating the physical world with paused simulation state and specified gravity value.

```
boolean doSleep = true;
world = new World(getGravity(gravityIndex), doSleep);
```

The implementation of the JBox2D physics engine was extended with basic functions for creating rectangle and circle structures with different body types. Listing 4.6 shows an example implementation for a circle - also known as ball - with special physical properties.

Listing 4.6: Sample code for creating a physical circle object.

```
public Body addBall(Vec2 position, float radius, BodyType type) {
    // Create Dynamic Body
    BodyDef bodyDef = new BodyDef();
    bodyDef.type = type;
    bodyDef.position.set(position.x, position.y);

    // Create Shape with Properties
    CircleShape circle = new CircleShape();
    circle.m_radius = radius;

    // Create a fixture for ball
    FixtureDef fd = new FixtureDef();
    fd.shape = circle;
    fd.density = 0.9f;
    fd.friction = 0.3f;
    fd.restitution = 0.6f;

    // Assign shape to Body
    Body b = world.createBody(bodyDef);
    b.createFixture(fd);
    b.setUserData(b);

    return b;
}
```

In line 3 and 12 definitions for the body and fixture are created and their attributes are set. Line 8 shows the instantiation of the underlying shape and finally, in line 18 and 19, the actual body is constructed with the specified data. This procedure is the base of every body creation process and has to be repeated for all objects.

In contrast to the graphics engine which runs on its own dedicated thread, the physics engine's update function must be called from within the gameloop. Besides, the *update()* function is also responsible for applying user input to the player object as illustrated in listing 4.7.

Listing 4.7: Managing the simulation process and control input.

```
//apply jumping force to player
if(p != null && p.getRemainingJumpSteps() > 0){
    float force = p.getJumpForce();
    p.getBody().applyForce(new Vec2((p.getRemainingJumpSteps() ==
        0)? p.getBody().getLinearVelocity().x * 7: 0f, force), p.
        getBody().getWorldCenter());
}
//simulate world
world.step(timeStep, iterations, iterations);
```

4.3 SuperController

The *SuperComtroller* is part of the MCV pattern design and primarily important for the game, despite also implemented in the editor. It controls the program flow of the game and keeps track of the game state and all objects in a single level. Moreover, it supervises the *GameLogic*, *Physics* and *StorageManager* components.

Once the *SuperController* is created its *initialize()* method instantiates the above mentioned controllers and loads all necessary game data into the LevelInstance and starts to drive the simulation with the UpdateThread classes.

4.3.1 LevelInstance

The *LevelInstance* represents the actual game and holds the currently active objects, references to the player and level information. Only one instance of the class can be active at a time and is referenced from the *SuperController*. This design allows minimal memory usage while playing the game. After a level is finished - either by completing it or exiting the game - the level instance is disposed by its *dispose()* method that frees all allocated memory. Loading the next level creates a completely new *LevelInstance*, ready to be filled with the latest level data.

Listing 4.8: LevelInstance holds all game specific data like all objects in the scene or the current Score.

```
1 public LevelInstance(Vec2 levelSize, int levelNumber) {
      this.levelSize = levelSize;
3     this.levelNumber = levelNumber;
      gameObjects = new ArrayList<DrawableObject>();
5     addObjects = new Vector<DrawableObject>();
      removeObjects = new Vector<DrawableObject>();
7     collectibleObjects = new ArrayList<Collectible>();
  }
```

All of its data is also accessible from the outside via special getters and setters. This functionality is mostly used by the *SuperController*, when it comes to calculations based on the level size or dynamically removing or adding objects to the game.

4.3.2 UpdateThread

Unlike rendering the graphics which does not need any configuration by the developer as it is handled by internal Android components, continuously executing the logic of the game requires an external thread. For this particular purpose, the *UpdateThread* - also referred to as "gameloop" - class was implemented. It is started by the *SuperController* and runs in a fixed time step, meaning that the time between an update is (nearly) the same at any time. The fixed time step simulation was preferred to a variable time step due to the following reasons:

1. Since physical calculations should rely on a fixed time step (for reproducible and deterministic execution - if this is supported by the engine), it is easier to use the same timer for logic procession. [7]

2. In addition to the physics, nothing but the *GameLogic* is updated by the gameloop and therefore interpolation between the updates is not necessary to smoothen the game experience.

In order to make sure every simulation step consumes exactly the same amount of time, the thread sleeps dynamically between two steps. The sleeping time is calculated depending on the processing time of the last step and the fixed time step duration, compare line 3 of listing 4.9.

Listing 4.9: Frametime (1/60 second) minus time consumed by the update.

```
long diff =  System.nanoTime(); - beforeTime;
2 sleepTime = (frameTime - (diff)) / 1000000L;
```

It is mandatory that the thread only runs if the application is active (cf. Android's running activity state). Thus, executing the thread can be stopped with a pause flag. As listing 4.10 demonstrates, the thread's run method is executed while the *paused* flag is set to *false*.

Once the flag is changed, for example by exiting the game the *SuperController* sets the flag to *true*, the gameloop finishes its current loop and stops running it.

Listing 4.10: The gameloop's run() method.

```
public void run(){
    while(paused == false){
        //do something
    }
}
```

4.4 GameLogic

The *GameLogic* works much like a referee which controls and guides the game flow. The class is not implemented in the editor but only part of its created games. There are three main tasks which are handled by the *GameLogic*. It provides methods to...

1. ...manipulate and store player data such as scored points and lives.
2. ...process detected collisions and check event trigger.
3. ...update the graphical user interface (HUD) depending on changed data values.

Instead of storing the player specific data in the player object of a level, this data is managed by the *GameLogic*. Due to the fact that the game completely drops a level with all its objects and creates a new one from initial values when a new level is loaded, the player's score and lives would be lost. Therefore, this approach was chosen to avoid constantly copying data from one level to another. Moreover, the *GameLogic* class checks the winning and loosing condition and finishes the game if the player has lost all his lives.

Most functionality of this class is processed by the update function which is called multiple times a second by the gameloop thread via the *SuperController*.

Listing 4.11: The update function controls the game logic.

```
public void update() {
    checkCollectibles();
    processCollisions();
    processEvents();
}
```

Listing 4.11 illustrates the main functions of the *GameLogic* class which are now explained in more detail:

checkCollectibles() This function checks if the player picked up a *Collectible*. Since this object does not have a physical representation in the world of the game, a collision with the player cannot be recognised. Because of that, *checkCollectibles()* checks if the player object and any *Collectible* overlap and triggers the corresponding events, for instance adding points or lives for the player.

processCollisions() As already mentioned in the physics engine section, all collisions between *ActiveObject*s are stored in a list in the *GameLogic*. The *processCollisions()* method now checks all collision pairs in this list and triggers events which have been defined by the developer with the event editor. Listing 4.12 is an example for a populated *processCollisions()* method which demonstrates its intention.

processEvents() The *processEvents()* function completes the detection of events. Every trigger which does not depend on collisions is processed by this method and immediately handles the attached functionality. An example implementation is demonstrated in figure 4.13.

Listing 4.12: Excerpt from the *processCollisions()* function illustrating how the triggering works.

```
int playerID = SuperController.getInstance().getLevelInstance().
    getPlayer().getID();
if(idA == playerID || idB == playerID){
    switch (Const.currentLevelID) {
        case 1:
            if(idA == 4 || idB == 4){
                killPlayer();
            }
            break;
    }
}
```

Listing 4.13: Excerpt from the *processEvents()* function with two levels and their respective events.

```
switch (Const.currentLevelID) {
    case 1:
        if(score >= 25){
            gotoNextLevel();
        }
        break;
    case 2:
        if(score >= 150){
            SuperController.getInstance().removeGameObject(33);
        }
        break;
}
```

Note the additional switch statement in figure 4.12 and 4.13. It subdivides the events for each level of the game and is explained in section 4.8 on page 59, concerning the export of a game with the export manager.

Updating the graphical user interface is not part of the gameloop since changes to this

data is only applied occasionally. The respective methods are not called in every update step, but rather on demand when the data is actually changed. Changing the players score for example will directly affect the HUD.

4.5 EditorGestureListener

Gestures play a very important role as they greatly improve the user experience. The implemented gestures facilitate every interaction with the level and its objects. The Android libraries provide different kind of pre-built gestures which can be recognised, but they have to be adjusted to work correctly. The *EditorGestureFilter* class implements all desired interfaces and processes the gestures which are then forwarded to a custom interface. This interface is implemented in the scene editor and the integrated callback methods are triggered, every time the gesture listener invokes them.

Most of the custom interface gestures are an extended version of their base implementation, but with modified parameters, such as *onClick(int, int, View)* or *onLongClick(int, int, View)*. The given interface is designed to interact with the *SurfaceView* and its displayed objects, as it provides additional information about the touch coordinates. For example, clicking on an object cannot be recognised with a preset *OnClickListener*, because it is not part of the hierarchical view structure, thus the surface has to provide the touch point which can then be used to take further actions.

The custom interface implements three other interfaces - *OnLongClickListener*, *OnTouchListener* and *OnClickListener* - to react to standard actions. Every interaction with the View the interface is registered to, calls the *onTouch(...)* function, which is the basis for every other gesture. It is possible to extract the absolute interaction point with the surface. Since *onTouch()* is the basis of every more or less complex gesture, it is always called first. The coordinates of the touch point are stored and therefore available until other gestures are triggered. Since the *EditorGestureFilter* implements interfering interfaces - *onScroll(...)* and *onLongClick(...)* can be triggered simultaneously - the involved methods have to set an enum lock, so that only the gesture first started can proceed. The realised gestures and their corresponding functionality are described as follows:

onScroll(MotionEvent, float, float) This fundamental gesture already exists in Android's *SimpleOnGestureListener* interface and is completed by simply adding a scrolling lock. The *MotionEvent*, which contains the pointer coordinates, starting the gesture and the method's x- and y-distance values are sent to the custom interface function.

onScrollEnded() Each time the *onTouch()* event handler is triggered by an "ACTION_UP" event, this means a pressed gesture is finished, it checks the gesture lock state and calls *onScrollEnded()* if the lock is set to "SCROLL".

onPinchZoom(double, Vec2) The pinch zoom gesture is more complex. It is composed of two independent motion events which come after each other. If the event is of type "ACTION_POINTER_DOWN", the application checks the distance between primary and secondary pointer on the surface and, if the distance is above a defined threshold,

initialises the pinch zoom by setting the "ZOOM" lock. From now on, every move event checks for the zoom lock and calculates the x- and y-distance between the update steps. In the end, the traveled distance for both directions and the squared distance are passed to this interface function.

onLongClick(int, int) Nearly the same as *onScroll()*, *onLoncgClick()* additionally takes the coordinates of the touch event.

onClick(int, int, View) Completely equals the *onLongClick()* functionality, but is triggered by another event.

onUpEvent() Extends the *onScrollEnded()* gesture so that every other finished gesture is perceived.

4.6 Game editor components

In this chapter, all editor activities are described in detail and a connection to the MVC components of the project is established. All activities are defined in the package *de.tum.gamestudio.Activities* and provide the structural basis for the game editor.

4.6.1 Start screen

The StartScreen class is the entry point of the whole application. It handles all necessary start configurations on first launch and offers the user the possibility to create a new game or resume previously created games. Every time the application is started, it checks whether the object database for the editor already exists or not. For this purpose a new DatabaseLoader is instantiated which then checks the existence of the object database and, if necessary, copies the pre-populated object database to the memory card of the device.

The DatabaseLoader class inherits from SQLiteOpenHelper, "*a helper class to manage database creation and version management*" [18] which is provided by the Android OS. Checking for installed databases is not natively supported by SQLiteOpenHelper, but can be simulated with other functions. To check for the database a new file is created with the database file as target and the *dbFile.exists()* methods then returns true if the database already exists. Based on this knowledge, *createDataBase(...)* either returns or begins to copy the data from assets directory to a path specified in *Const.java*. The default application directory is not used as described by Google, because this directory is private and other applications can not access it. This would lead to problems when compiling the game since the external compiler tool has to access the copied libraries. Therefore the "AndroidGameStudio" folder on the SD-Card will not be uninstalled with the application, so the user has to do this on his own.

Input- and *OutputStreams* are created with the data file as input source and directory path, plus filename as destination file. The functionality to copy files from an *InputStream* to an *OutputStream* is very general and therefore defined in the static *Functions.java* class to make it accessible from anywhere in the project. Both streams are passed to that function,

which then partially buffers the input and writes it to the output file. Finally, it also flushes the streams and closes them to avoid memory leaks. After this procedure, the database is ready to be use until user data for the application is wiped, or the game editor is uninstalled.

The activities' graphical user interface offers buttons to create a new game, resume a previous game or load a saved game. Rather than processing the corresponding functionality directly, a new activity is started.

4.6.2 Game setup

The GameSetup activity is prepared to handle two different scenarios. In scenario one, the user wants to create a completely new game, hence all input fields are left empty and the "Create game" will trigger the creation of a new game. Scenario two is based on an already existing game, but the user wants to change, add or delete data which has been entered earlier.

The different usage is identified by a single boolean value passed with the intent. A true init value will cause the activity to store all entered data in a database. For every game the user wants to create he has to enter some game specific data. This includes information about the game, as well as author and is specified in the *GameSetup.java* class. At least the game's name must be entered and it is also necessary to select the screen orientation. Possible screen orientations are *vertical* and *horizontal*, which are equivalent to Android's standard orientations, *portrait* and *landscape*. To create a new game, this information is sufficient, but there are also other additional input fields. It is up to the user if he wants to fill in his name and a description for the game right away. Besides that, the user is also asked to select an image or icon for his game.

Listing 4.14: Starting the *MediaStore* content provider to select a gallery image.

```
Intent i = new Intent(Intent.ACTION_PICK, android.provider.
    MediaStore.Images.Media.EXTERNAL_CONTENT_URI);
2 startActivityForResult(i, 1234);
```

On click, the image selector button creates a new Intent which opens a gallery where the user can select any image stored on the device. As code example 4.14 shows, the image is selected by taking advantage of the *MediaStore* content provider of Android OS. The intent tells the content provider to return the selected image in the gallery by setting the ACTION_PICK flag of the intent. The result is then handled in the activity's *onActivityResult(...)* method. With the function's result code, the success of the operation can be checked. If the code of the received data equals the pre-defined RESULT_OK value, the path to the image and its filename are extracted from the received Intent by querying the *_data* rows of the *MediaProvider* and selecting its first entry. Image filename and directory path are temporarily stored for further treatment and the image is decoded and converted to fit the image button's dimensions. *DecodeImage(String path)* takes a file path as input, creates a new bitmap by utilising the *BitmapFactory* function *decodeFile(String s)*

and re-calculates width and height of the target image. Depending on the current size a scale value is calculated which, multiplicated by the source image size, limits width and heigth to a maximum size of two hundred device independant pixels (DIP). This function's purpose is to match image to button size and properly display the result to the user.

By hitting the back or cancel button, all entered data is discarded and the editor returns to the main menu. To finally set up a new game, the user has to click on the "Create game" button.

The *DatabaseController.java* class handles the access to the database and implements the functionality mentioned above. *CreateNewGame(...)* for example takes all information from the setup activity, encloses them in a *ContentValues* instance and puts the values into *GameData* table. Name, author, description and image path are stored as strings, the orientation is an integer value. A significant part of this function is the return value of the insert command. It returns "*the row ID of the newly inserted row, or -1 if an error occurred*", as described in JavaDoc. Since the row ID is the primary key of the table, it identifies the game uniquely and can be used for further interaction with other tables, for instance the the subsequent creation of a new level in the game. For the users' convenience, right after the game also a level is created and linked with it. The game ID, standard level number and name are passed to *addLevelToGame(...)* which then constructs a new level and stores it in the database. Code example 4.15 depicts this functionality.

Listing 4.15: Creating a new level and storing it in the database.

```
public Level addLevelToGame(int gameID, int levelID, String
    textureNamePI){

    //insert data
    ContentValues args = new ContentValues();
    args.put("Game", gameID);
    args.put("Level", levelID);
    args.put("PreviewImage", " ");
    args.put("LevelName", "Level " + levelID);
    args.put("Gravity", 0);
    int combinedID = (int) db.insert(Const.tableSavedGames, null,
        args);

    return new Level(levelID, combinedID, textureNamePI, "Level "
        + levelID);
}
```

A level consists of a name, preview image, gravity index, game ID, level ID and, most important, a combination of game and level ID. Section 4.6.4 (see page 50) goes into more detail about the gravity index. Surprisingly, level ID is not marked as a primary key of the table, but the *_id* field is. This is because enumeration of the levels must be the same in each game, to ensure correct sorting order. It starts at one and increases by one with every new

level. The additional *_id* field is not absolutely necessary but without it, more tables would need additional fields to clearly join data between the tables. *LevelObjects* and *Events* table make use of this design. The return value of *addLevelToGame(...)* is not processed here.

Once the level is ready for use, the *createNewGame()* method now returns the id of the current game which is then stored in the static class *Const.java*. This is an appropriate way to make it available from everywhere in the editor and simplify access to often used variables. Otherwise, the value would have to be passed to every single activity and sub-method. Finally, a directory on the SD-Card is constructed with game ID as name. Yet, the folder is not used, but it is required to store screenshots for all levels. To proceed towards the next step, the *EditorLevel* activity is started. To make sure that the user cannot navigate back to the setup activity, the next command in the program flow is a *finish()* statement. It exits the activity immediately by popping it of the activity stack and returns to the main menu.

The second application scenario describes altering an existing game. Both scenarios are very similar but yet different. If the boolean *init* variable is not true, this means it is no new game but the user wants to change data, all data of the game is loaded. Additionally, the "Create game" button is renamed to "OK" and will now update the data in the database instead of inserting new values. By calling *getGameData(...)* with the ID of the game, a query is executed on the *GameData* table. The resulting cursor holds all required data which is then displayed to the user. The records can now be changed as usual and by clicking the "OK" button, their database fields are updated. The *DatabaseController* function *updateLevel(...)* is configured to modify the game according to the active game. Changing the game's name with an already existing .apk file or installation on the device may lead to problems, especially duplicate installations with different names are a problem. The game setup takes care of these problems and renames already existing .apk files together with the database entry and also takes care that the currently installed game has to be uninstalled in order to rename the game.

4.6.3 Level editor

Every time the user creates a new game, resumes or loads an old one, the *LevelEditor* activity is called. Its main goal is to display all existing levels in the selected game and provide functionality to manage the game and its levels. This activity also offers buttons to export a game and play an already existing game.

In order to make use of the *GridLayout* of the GUI, a *LevelAdapter* is instantiated and set as the grid's data adapter. As mentioned in the previous section, often used variables are stored in a static class to access them quickly. However, this design also entails risks. It is possible that the context of the class gets lost when the application runs in background for a long time and the garbage collector clears the data of the static class. In order to load the game and its levels correctly, the currently active game ID is additionally written to the shared user preferences every time he enters the *LevelEditor* activity. When the activity is started or resumed, it checks the content of the static variable and, if it is not valid, reloads

the game ID from shared preferences. Another check of the game ID is performed and, in case it is valid, starts loading the levels.

To display all the levels in the game, the program has to process three different tasks. The level data has to be queried from the database, translated into a suitable data representation for object oriented implementation and then loaded into the grid adapter. All three steps are explained in more detail in the following section.

The first task is made up of a simple database query, executed on the already opened *DatabaseController*. It requests the level's name and image, as well as its ID and, of course, the combined ID by selecting all levels, where game ID in the table and in *Const.java* class are the same. Since the query command of every *SQLiteDatabase* is able to order the selected rows, the resulting Cursor object will hold the data ordered by level ID in ascending order. Accordingly, the grid will always show the correct sort order of the levels.

This very simple step is followed by the preparation of the Cursor data. An array list of levels forms the basis for the adapter. All data is read from the Cursor row by row and translated to a level object. The object does not have any functionality, but only serves the purpose to encapsulate the data and make it available to represent it in the array list.

Now that the levels are prepared for further actions, the application can proceed with the last step. By subclassing an *ArrayAdapter*, a concrete implementation of *BaseAdapter*, a creation is constructed to use custom objects in a more complex layout. All essential functionality is inherited from its parent, like *getCount()* and *getView()* and are overridden to match the layout's requirements. The level list is passed to the newly created level adapter and can now be used by the adapters methods. As described in the Android API documentation, *getView()* creates a view that displays the data at the specified position in the data set. The standard implementation is only applicable for a single text view, however, the used layout file consists of an image and text. The extended *getView()* method is configured to inflate a specific layout for the grid item if the view does not already exist and, if possible, reuse an old view to save resources. In both cases, the level's name is set to the *TextView* and the image is loaded from SD-Card and set as content of the *ImageView*. The adapter is now ready to display an item in any list or grid.
A missing piece is the adapter's *getCount()* method. It has to return the total number of elements stored, which in this case, equals the number of elements in the *ArrayList* of levels. This will be of more significance when the user deletes a level and is therefore explained later in this section.

To interact with the items in the grid, two different kinds of listeners are registered on the grid and the interface with the logic is implemented in the activity. Simple clicks on an item in the grid are handled by an *OnItemClickListener*. As could be expected, a click on a level opens the scene editor with the corresponding level. The triggered callback method contains the logic behind this. Right after the opened level and combined ID is written to *Const.java* class the scene editor is started. All information about the scene editor are explained in the next section.

Figure 4.1.: Level editor with sample levels.

To delete a level from the game, the user has to click on a grid item and hold it down for a while. Pressing the confirm button in the delete dialog will then delete the level not only from the game, but also from the database. As already mentioned, it is crucial that all level numbers of a game are consecutive, so deleting a level other than the last one will result in an invalid state. Luckily, this problem can be circumvented with a simple SQLite statement. The database controller decreases the level ID of every level in the game, with a level number greater or equal to the deleted one, by one. As a result, the required order is maintained. Additionally, the level is then removed from database and adapter. Wasting resources is prevented by iterating over the adapter and manually decreasing the level ID similar to the database operation. The leading thought behind this procedure is, that the system does not have to reload the whole adapter. Instead, only the data is touched which is not visible to the user and therefore it is not necessary to update the layout.

The three-button layout at the bottom of the screen provides functionality to create a new level, edit game information and, most importantly, compile the game. After compiling a game two additional buttons are added to this layout, which offer the functionality to start the newest version of the exported game and share the created game via email.

The most basic part is triggered by the “New level” button. Section 4.6.2 already introduced the *addLevelToGame()* method, but this time the return value of the function comes in handy. The created level is written to the database and then directly translated into a Level object. Adding it to the level adapter finishes the instruction.

The edit game functionality has been mentioned in section 4.6.2, too, and refers to the

second scenario. The game setup intent is started without extra information about the *init* state, thus interpreted as "false" by the *getExtra()* method by default.

The "Publish game" button triggers a routine which outputs a complete playable game. Publishing requires the "JavaIDEdroid" application to be installed, thus the level editor checks if this requirement is met by calling the *PackageManager's getApplicationInfo(...)* method with the package name of the application. If the application is not installed the user is asked to install it. See section 4.8 for more information about the *ExportManager*.

Other than the already mentioned buttons, the fourth button is not visible all the time, but only when the created game is already installed or an .apk file exists to install the game. To check the installation status of the game, the *PackageManager* is utilised once again. Additionally, the versions of the currently installed game and the newest export are compared. If the version numbers match, the game is started but not installed again, otherwise the installation is started. To compare the version numbers, an intent filter is registered the manifest file of the exported game, a version number is written in to the *Const.java* class of the game and the main activity is capable of receiving intents. The game editor then sends an intent with the newest version number stored in the database to the game, which then checks if the installed version is up-to-date and simply keeps running if this is true. In the event that they do not match, the result code "RESULT_CANCELED" is returned to the caller and triggers the installation of the game. Listing 4.16 shows the decision process in the main activity of the game.

Listing 4.16: Comparing the game versions.

```
1 if (action.equals(Intent.ACTION_SEND)) {
      if(intent.getIntExtra("Version", 0) == Const.versionNumber+1)
3         return true;      //versions match -> play game
      else{
5         this.setResult(RESULT_CANCELED, intent);
          this.finish();
7         return false;
      }
9 }
```

The "Share game" button creates a new intent and puts all necessary information for an email to the intent. Only the email address is left out and has to be handled by the email application. To send an APK file, the MIME type of the file has to be set to *application/vnd.android.package-archive* and the file path has to be parsed as an Uri.

4.6.4 Level settings

Each level has some additional information which can be adjusted in the *LevelSetup* activity. This part of the editor only has minimal settings. It is possible to change the name of the level and select the gravity and control methods for the game. Gravity and control method are interdependent and offer three settings to choose from:

Standard gravity The game is configured to have a side view to the scene, therefore the earth's gravity is applied and all physical objects with a mass are attracted to the bottom of the screen. The player's movement is controlled by the orientation sensor but limited to the x coordinate. Jumping is initiated by tapping the screen anywhere, which applies an impulse to the player's body inverse to the gravity.

No gravity There is absolutely no gravity applied in any direction. Forces applied to an object make it travel endlessly without reducing its speed. Player controls are based on the orientation manager and enabled for x- and y-direction. This setting is similar to a top-down view.

Sensor gravity The sensor gravity option is also a top-down view, but instead of controlling the player's movement the orientation sensor controls the gravity of the level, thus influences all dynamic physical objects (Terrain is not influenced for instance, as its fixed flag of the physical body is set).

Selecting multiple options is prevented by the *RadioGroup* the items are arranged in. Checking a *RadioButton* in the radio group deselects any other checked option in the radio group. Initially, the standard gravity option is checked and enabled by default for every level.

4.6.5 Scene editor

The core of the whole application is the scene editor which allows the user to modify a level and also serves as interface between event and object editor.

Graphical user interface

The GUI can be divided into two base components. The *LinearLayout* on the left is designed for object and level manipulation and also acts as object gallery. The large *ViewGroup* on the right implements the graphics engine and shows the level with all its objects as well as level and camera bounds. Depending on the mode the editor is in, the layout holder on the left either shows the user's object favorite list, or it displays properties of level and selected objects.

Placing mode

The first of the modes mentioned is the placing mode. It allows the user to select objects from his favorite list and place them in the level. The favorite list is positioned in the holder on the left and shows all favorite objects in a specific category, e.g. "Background" objects. All data for the list is loaded in the *loadObjects()* function. Similarly to other load functions in the different application activities, a cursor with the data is requested from the database, which is done by *getObjectGalleryFavoriteData(...)*. It is sufficient to load name, image path and ID of the *DrawableObject*, since it is only designed to be displayed in the favorite list and not yet for usage within the MVC part of the editor.

The *DrawableObject* class is able to create subclasses of its own by passing the proper

parameters. The objects resulting of this procedure are then added to an array list and finally, the adapter for the list is created out of them. The adapter used to display data in the favorite list equals the adapter used in the level editor. Instead of Level objects it holds instances of *DrawableObjects* and is based on a different layout for its views, although this only differs in its size. However, as both adapters are nearly the same, details are not explained here any more.

The user is able to switch through all existing categories by clicking the "Change category" button on the bottom of the holder. The displayed information in the holder are then changed to an overview of all possible categories. After selecting an entry in the *ListView*, the favorite list is reinitialised with the corresponding data. The adapter is populated with the data right after the favorite list by calling *fillFavPicker()* which creates a list of Hash-Maps as basis for the list adapter. The Hash-Map allows to map object information to a key, in this case the values for the text and the path to the icon of every list entry are stored. The complete array list of the Hash-Maps is then used to provide the data for a new *SimpleAdapter* instance. Together with the layout for a list entry and the resource IDs of the views the data is now accessible by the list.

In placing mode, most of the interaction with the surface is not enabled. It is only possible to change the camera position and zoom level, object manipulation is completely deactivated in this mode. However, the user can select an object in the favorite list, which is then highlighted, and place them everywhere inside the level bounds by clicking on the surface view.

Although moving objects with a gesture is deactivated, it still shares the same function as scrolling. If placing mode is active, *onScroll(...)* changes the position of the camera by one per cent of the traveled distance and also considers the orientation of the device to make sure x- and y-coordinates are recognised correctly, as illustrated in listing 4.18.

Listing 4.17: Moving around the camera.

```
1 if(this.getResources().getConfiguration().orientation ==
    Configuration.ORIENTATION_LANDSCAPE){
     distanceY *= -1;
3 }
renderer.getCamera().changePosition(distanceX / 100, distanceY /
    100, false);
```

Both, zooming and placing objects also share the triggering callback methods with other functionality in editing mode. Zooming requires a cleared object selection. Only if no object is selected, the zoom of the camera is changed by the incoming value. Object placement is handled in the activities *onClick(int, int)* method. A click coordinate has to be transformed from screen to world space, because the screen coordinates are always values between zero and the length of the OpenGLView in x- and y-direction. Therefore, *getWorldPos(int, int)* includes the current camera position, to obtain the correct interaction point in world space. A last check makes sure that the object positioned at the pointer is completely inside the

level bounds and finally, a new *DrawableObject* is created and added to the game object list. Name, texture path and type of the object are copied from the the *DrawableObject* instance in the favorite list, the position is given by the world coordinates calculated in the last step.

The size of the new object is based on a standard value for every object type and afterwards adjusted to width and height of the texture so that non quadratic texture dimensions are transfered to the object and do not deform it.

Listing 4.18: Moving around the camera.

```
if(mode == EditorMode.Placing){
    Vec2 pos = getWorldPos(posX, posY);
    if(selectedItem < 0 || !superController.isCenterInGameArea(
        pos, new Vec2(1,1)) ) return;

    DrawableObject item = adapter.getItem(selectedItem);
    pos.subLocal(new Vec2(0.5f,0.5f));

    DrawableObject d = DrawableObject.createCenteredFromValues(
        item.getName(), item.getTextureName(), pos, new Vec2(1,1),
         0, item.getType());
    d.adjustSizeToTexture();
    superController.addGameObject(d, false);
}
```

Editing mode

To switch between the editor modes, the user has to click on the mode selector button on the top left corner of the render frame. In order to create a great game, the standard attribute values for rotation and size are in most cases not very appropriate. Therefore, the editing mode makes it possible to modify all attributes of a placed object. To do so, it is necessary to select the object in the surface view. A chosen object is indicated by a red bounding box. It is not possible to select more than one object at the same time. Since many objects can overlap each other, a selection menu is shown to the user, when he taps the overlapping spot. He can then choose the intended object.

In editing mode, the holder view is an interface to change both, object and level properties. When an object is selected, the holder integrates a layout which allows to change the object's size, position and rotation. Plus and minus buttons for size and position change the object's width and height, respectively move the object around in the world. The equally labeled buttons for rotation apply counter- and clockwise rotation.

Additionally to this interface, it is possible to modify object's via gestures on the OpenGL surface. Assumed that the object is selected, its size can be changed with the pinch zoom gesture. Moving it around requires the user to start the gesture on the object. Scrolling

around on the surface then lets the object follow the path of the finger.
The paste bin button in the top left corner, which is only visible in editing mode, provides functionality to remove a selected object from the level.

With the level size feature already addressed, the user can change the size of every level individually. The layout of the main View is included in the holder on the left and always visible in editing mode. The vertical axis of the layout correlates to the height, the horizontal axis to the width of the level. To modify the level boundaries, the user simply has to press the plus and minus buttons provided by the layout. There is no limitation for the size of a level, except the fact that it cannot be smaller than the target platforms camera rectangle. In this case, the level bounds and the camera rectangle share the same dimensions and overlap completely. Every time the size of the level is changed, the editor checks the camera rectangle's position and adjusts it to the new level bounds if it intersects the level bounds rectangle. For example, decreasing the level width with the camera rectangle nearby will change the x position of the rectangle and therefore keep it inside of the level.

What's more, objects outside of the level bounds are not allowed either, so every interaction with an object has to guarantee that this requirement is satisfied. The editor makes sure that placing an object, changing its position, resizing the object and also rotating it will always keep the object in a well-defined state. Indeed, decreasing the level width or height can lead to objects which are no longer inside the level bounds, but the editor also takes care of this circumstance and deletes all affected objects.

Autosafe

The user does not have to worry about saving his level, as the editor takes care of that automatically. Every object that is placed or removed, is immediately written to or removed from the database. Furthermore, all object manipulations are identified and the application updates the object data when *onDestroy()* is triggered. Certainly, this could also be done while the user is manipulating an object, but this would lead to excessive use of the memory card. Especially gestures used to resize the object or change its position, which update the object multiple times per second, would greatly affect system performance and quickly lead to "**A**pplication **N**ot **R**esponding" (ANR) errors.

Popup menu

In order to switch to *EditorObjectGallery*, *EditorEvent*, *EditorLevel* or *LevelSetup* activity, an instance of *MenuPopupWindow* is created and shown to the user. It inherits from the already described *CustomPopupWindow* class and additionally implements the *OnClickInterface* to detect button clicks. The design of the class allows to set a special layout for every specific implementation, in this instance the "menubar_popup.xml" resource is used which has a button for every of the named activities. Activity and button are then associated with each other through an intent. To show the pop-up menu, the user has to press and hold the OpenGLView. With the coordinates passed from the gesture listener, the menu is then opened and centered at the pointer coordinates.

4.6.6 Object editor

A game editor without objects would be pretty useless. Therefore, a gallery provides the user with objects in many different categories he can place in a scene. The layout of *EditorObjectGallery* consists of a list on the left and a *TabHost* on the right. The *TabHost* itself is composed of a tab for every category and shows all objects a particular category within a grid. The object classification does not completely match the *DrawableObject* pattern, but every listed object is translated to a subclass of *DrawableObject*. Some of the object names clearly describe their main feature, such as *Collectible*, though not every object alias is that obvious. The following section gives an overview of all categories:

Background *Background* objects are designated for embelleshing the level. They do not have a physical representation, thus the user can not interact with them.

Collectible As its name implies, a collectible can be picked up by the player. It then increases the player's score by a specific amount. The analogue *DrawableObject* subclass shares the name and is inherited from *PassiveObject*.

Moving Physical moving objects are called *Moving*. It is inherited from the *ActiveObject* class, which is the parent class for all objects that provide a body and are able to collide with each other.

Player A *Player* is a type of *ActiveObject* and can be controlled by the user. It is part of the physical world simulation, but user input can apply forces to it depending on the chosen control functionality. It is not possible to place more than one object of this type in a level.

Terrain The *Terrain* is a type of *ActiveObject*, but it cannot move freely all over the world. Instead, the Terrain has a fixed position and is therefore suitable for creating the ground on which the player can move around. This object also has some special textures which allows tileable patterns (edges of tileable textures do not mismatch and create the illusion of a continous large texture).

The list makes use of the same objects as the grid. It is connected to the currently active category and shows the user's favorite objects. The intention behind the favorite list is, that not every single object is shown in the scene editor, but only a small part of it. This design is more comfortable for the user, as it allows faster and clearer navigation. To add an object to the favorite list, clicking on it in the tab is enough. It is then added to the adapter underlying the favorite list, as well as stored in the database so that the user does not have to reselect his favorites every time. Removing an object from favorite lists works similarly. A click on the object in favorite list removes it from both, adapter and database. The background of the adapters is not addressed anymore, because it is already described for the level adapter.

So far, the mechanics behind the *TabHost* and its content have not been mentioned. At first glance it would be convenient to load the object data for every tab during the start of the activity. Nevertheless, this design is very memory intensive for large lists and should be avoided. To be sparing with the memory of the device, there is only one

adapter for the favorite list, and one for the tabs which is reused on every tab change. The *OnTabChangeListener* added to the *TabHost* triggers the *loadObjects(...)* method. It creates the adapters data representation layer, clears and re-initialises the adapter. Admittedly, following this structure causes the application to reload the data on every tab change, but since calculation time is rarely a limited resource on mobile devices, this design is preferred to memory intensive preloading.

4.6.7 Event editor

Now that the level environment is set with all its objects it is time to specify the logic and events which make a game more interesting. Based on this idea the *Event editor* was introduced to create events with a trigger and functionality.

The event editor is mainly built upon the *Event* class which holds the definition of *Trigger* and *Functionality*, provides functions to create text for its individual components and translate between the enum and integer representation. Generally, the identifier for the component type is stored as an enum value, but *ListView*s cannot handle these and therefore the translation methods are absolutely necessary. Furthermore, the *Event* class is able to create itself from previously stored database values.

Creating a new event is pretty simple: A click on the "New event" button is enough and a custom dialog box pops up with templates for triggers and functionality. They are arranged in two columns each one holding a *ListView*, one with the trigger templates, the other with the functionality templates as depicted in figure 4.2. It is mandatory to select one item in each *ListView*, otherwise the event cannot be created. Pressing the "OK" button adds the event to the overview page and stores it in the database with initial values, including the type of the trigger and functionality.

Note the purple question marks in figure 4.2. A text part highlighted in this colour signals interaction possibility for the user. The question marks in this case additionally indicate that the standard values have not been changed yet. Once a value is assigned, the question mark is exchanged by the chosen option but still highlighted in purple. The values which can be chosen are bound to a component type, meaning that the user can select *ActiveObjects*, numbers or areas. Manipulating event values directly updates the database representation to ensure data consistency between the editor's displayed and the stored information.

Choosing the event options is implemented in different ways depending on the data type that is used for the functionality or the trigger. For a integer data type for example, a simple dialog is shown with an input field for a number. Creating an area or selecting an object in the game is a bit more tricky. For these options the *ObjectPicker* activity is started which simply shows a *SurfaceView* with all the objects in the game. To select an *ActiveObject* the user has to click on an object displayed in the surface view. The activity then returns the object's id to the *EventEditor* which then links the object with the event.

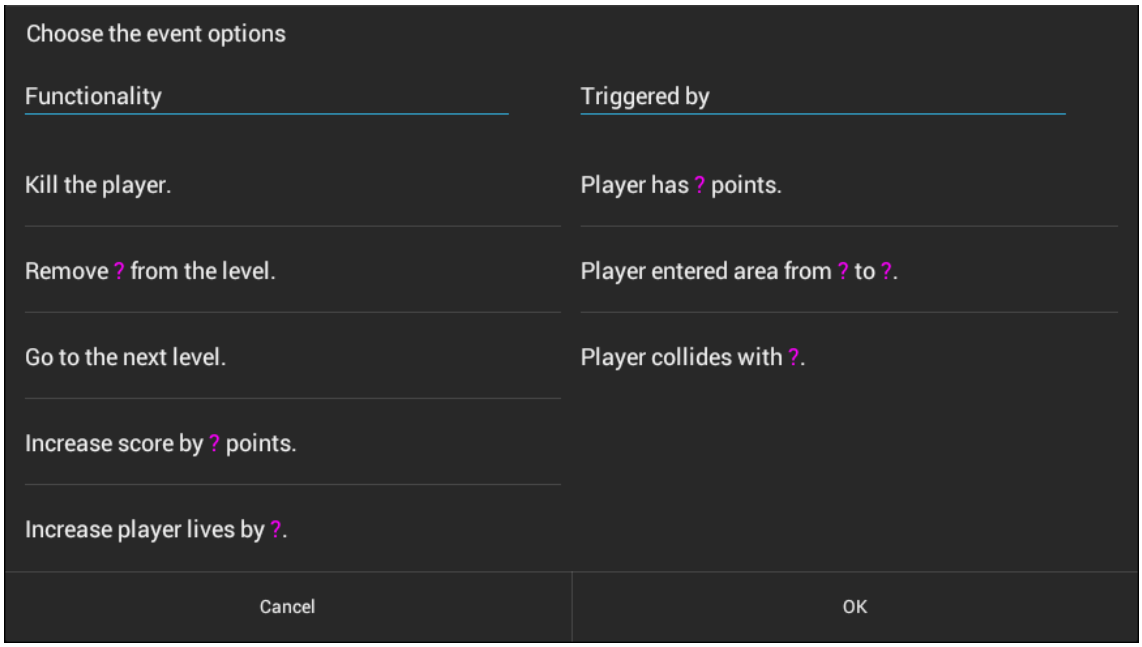


Figure 4.2.: Dialog for choosing event functionality and trigger.

Instead of selecting an existing object, the user has to create a new one when he wants to define an area. Clicking anywhere on the surface of the *ObjectPicker* creates a new rectangle with base values (position at the interaction point and size of one times one meters). Clicking somewhere else, outside of the rectangle's bounds will the currently created rectangle to the new position. Furthermore, the user can change the size of the area and refine its position by clicking on the existing rectangle. This is then highlighted by a bounding box and the user can manipulate it just like all other objects. To complete the editing part an accept the rectangle, pressing the "OK!" button is enough. Otherwise the user can cancel the picker activity by pressing the "Cancel" button. When a valid rectangle is created, its position and size is put into an intent and returned to the calling activity which then updates the corresponding event in the database and updates the adapter display.

In the following there is a short description of the components already mentioned, "Trigger" and "Functionality", as well as a list of all their templates and respective value types.

Trigger As its name implies, a trigger defines the activating mechanism of an event. Most triggers are preconfigured to involve the player. It would also have been possible to remove the player from the trigger and replace it with an arbitrary object which can be selected by the user, but this approach was not implemented to simplify the user interface. Table 4.1 lists all available triggers.

Functionality The *Functionality* is activated by a trigger and defines the performed action. They affect various different game mechanics. Table 4.2 lists all available options.

Trigger	Value type
Player has ? points.	Integer
Player entered area from ? to ?.	Integer Array
Player collides with ?.	ActiveObject

Table 4.1.: All available "Triggered by" options.

Functionality	Value type
Kill the player.	Void
Remove ? from the level.	DrawableObject
Go to the next level.	Void
Increase score by ?.	Integer
Increase player lives by ?.	Integer

Table 4.2.: All available "Functionality" options.

4.7 DatabaseController

Requirement REQ-13 in section 3.1.1 describes the possibility to store persistent data for a game which is created by the editor. The *DatabaseController* class complies this demand and handles all storage operations initiated by a subcomponent of the game editor (cf. section 4.6 concerning the game editor components). It extends the *SQLiteOpenHelper* for managing the database.

Basically, this controller is used for processing SQLite queries. It provides basic functions for inserting, deleting and updating object data. The code in listing 4.19 is an example for deleting a *DrawableObject* from a level. Furthermore, all events associated with this object are updated too, to avoid issues with inconsistent data stored in the database.

Listing 4.19: Database instructions for deleting a *DrawableObject* in a level and updating affected event data.

```
1 public void removeGameObjectFromLevel(DrawableObject d){
      db.delete(Const.tableLevelObjects, "_id=?", new String[]{
          String.valueOf(d.getID())});
3
      //update events where the object played a role
5     ContentValues args = new ContentValues();
      args.put("object1", -1);
7     db.update(Const.tableEvents, args, "object1=?", new String[]{
          String.valueOf(d.getID())});
      args.clear();
9     args.put("object2", -1);
```

```
        db.update(Const.tableEvents, args, "object2=?", new String[]{
            String.valueOf(d.getID())});
11  }
```

The *DatabaseController* only serves one additional purpose which is to extract the game data of a specific level for exporting the game. This is not absolutely necessary, precisely because copying the whole database is easier and faster but produces large file outputs if there are many games stored in the editor. Similar to selecting the games textures for exporting (this surely has a greater impact on the resulting game file size), this approach was also realised here.

4.8 ExportManager

The *ExportManager* bridges the gap between editing a game and actually making it playable with an APK file. To start the export process, which consists of two independent main tasks, the user has to click the "Publish game" button in the level editor. First of all, the entire game data of the currently opened game is prepared by the editor - this is explained in the next section 4.8.1 - then, the compilation process is started and forwarded to an external tool which is illuminated in section 4.8.2. In addition to its blocking job, the progress dialog also serves the purpose to inform the user about the export and thread status.

4.8.1 Exporting a game

Running the export is a major operation and processing it on the UI thread results in bad performance and ANR errors. Therefore, it is encapsulated in a custom implementation of *AsyncTask* which "allows you to perform asynchronous work on your user interface" [16]. Nevertheless, the GUI is locked by a progress dialog until finished and no user interaction is possible to avoid data consistency violations when changing game content while exporting it. Exporting a game is divided into four main parts, three of these - all except compiling the application - are handled by the *ExportManager* class:

1. Copy the game template with its Java classes and the required libraries.
2. Insert customized game data including events, textures and game objects.
3. Compile and sign the game.
4. Delete all copied and temporary files except the output application file.

The first part, recursively copying the data is performed by the *copyStandardFiles()* and its subsequent *addRecursively(String path, String assetPath)* and *copyFilesInDir(String path, String assetPath)* functions, copies the entire "ProjectTemplate" directory and all of its subfolders and files from the editor's assets, to a destination directory on the SD-Card. Game and layout files are part of this template, as well as bash files for the compilation process and much more. In case the copied file is of type XML, BSH or Java, its content is searched for the keyword "GameName" and each appearance is substituted with the name

of the exporting game. Additionally, the folder names containing "GameName" are also renamed the same way. This is necessary to adjust the package names and compiler options.

The next step is to insert the user customised game data into the exported game. Controlled by the *insertCustomizedData()* function, first the game logic is processed, then the required textures are identified and replicated. Exporting the game logic requires to translate the specified events in the event editor to a text representation. This is handled by the *exportGamelogic(String gameName)* method. A switch statement encapsulates the logic, which holds all events of a single level in one execution path indicated by the level number. With this design, only events of the currently played level have to be checked and errors with undefined objects can be avoided. Moreover, especially for large numbers of events, the performance is quite better. The base for translating the events is created by the *createEvents(Cursor c, boolean rawData)* function in the event class, which takes a *Cursor* with game data as input and return an array list of *Events*. Again, the event data is then translated by the *eventToString(Event e)* method, which combines an if-clause for the trigger and formatted functionalities as execution code. Since different methods are responsible for checking general events and collisions (they are not trigger based, but stored in a list), both of these functions are manipulated during the export process. However, the translation process is the same for both functions.

Copying the database and object textures is also part of this step. Instead of simply copying the entire database, the editor's *copyGameData(String outFileName)* extracts the objects and game data of the chosen game. Furthermore, all used textures are identified and their names are returned in a String array to copy only the necessary textures with the *copyTexturesToAssets(ArrayList⟨String⟩ textures)* function of the *ExportManager*. Particularly for games with few different objects this effort pays off, as the game editors texture archive may be very large.

Building the game is handled by an external tool explicitly explained in section 4.8.2, Compiling a game and finally all copied game files except the created *.apk are deleted.

4.8.2 Compiling a game

Writing a compiler for Android applications which runs on the same platform is a massive task to be carried out. The required working hours are probably enough to write a separate thesis about this topic. This problem was remedied by using the external tool "JavaIDEdroid" which handles the entire compilation and signing process. The Google project page describes the purpose of the application:

> *JavaIDEdroid is an integrated development environment which runs on Android and allows to create native Android applications without the need to use the Android SDK on Windows or Linux. It should run on Android 1.6 or higher.* [3]

Besides, the integrated development tools are listed in the online documentation, also briefly described here and additionally illustrated in figure 4.3:

aapt tool The aapt compiler takes the project's resource files and compiles them. [14]

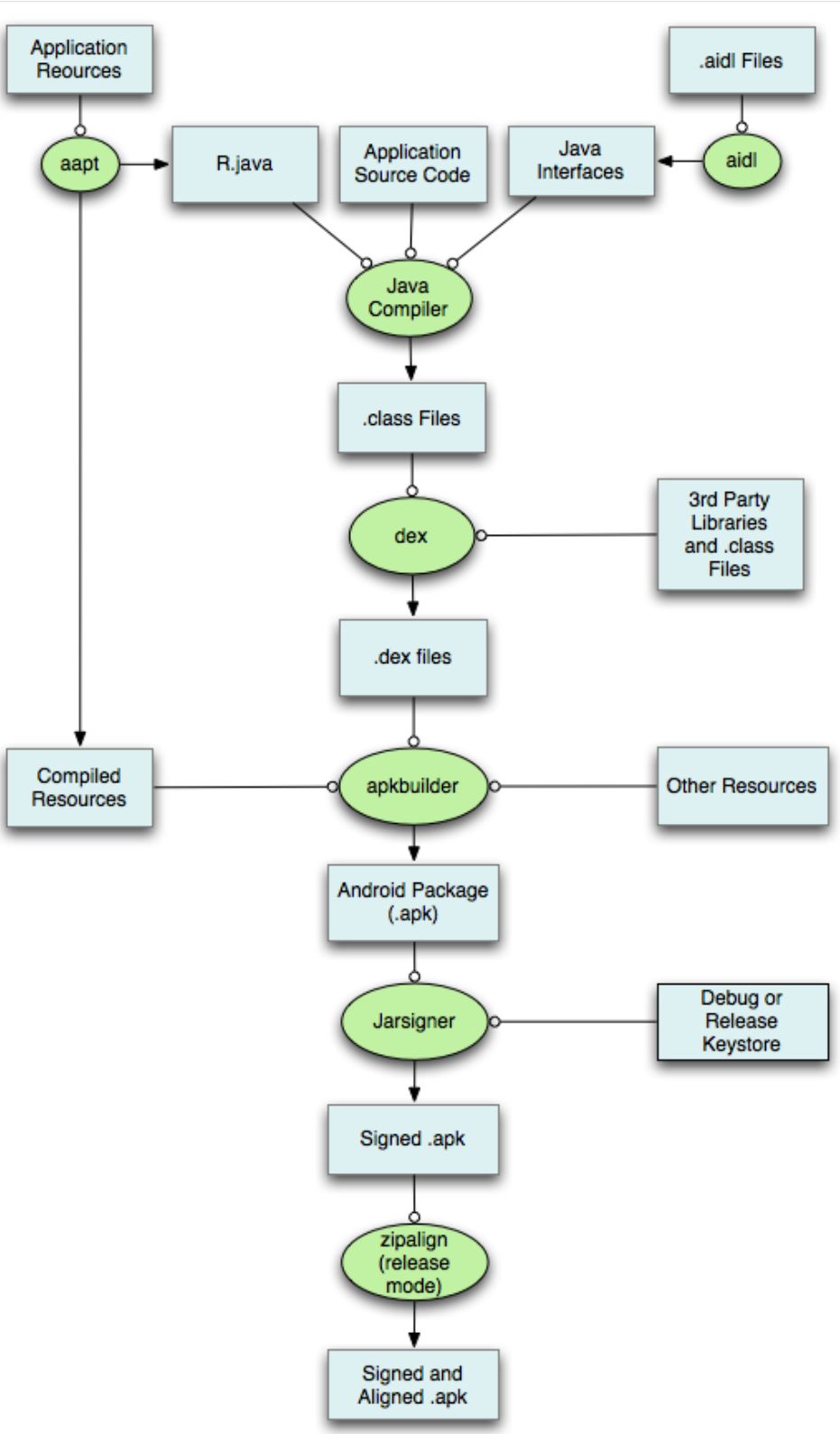


Figure 4.3.: Outline of a typical build process of an application. [14]

Java compiler The applications source code, the resource file and all Java interfaces are compiled by the Java compiler and output as .class files. [14]

dx tool The dex compiler converts the Java .class files into .dex files, destined for the DVM. Third party libraries such as the JBox2d physics engine are converted to .dex files, too. [14]

ApkBuilder All compiled and non-compiled resources as well as the .dex files are packaged by the apkbuilder to a single .apk file (only unsigned APKs). [14]

zipsigner-lib Installing the application on a device requires it to be signed by a debug or release key. This is done by the zipalign-library. [14]

BeanShell Interpreter Interpreting and executing the shell scripts which start the individual tools and compilers is done by the BeanShell Interpreter. It simplifies and automates the build process. [3]

JavaIDEdroid supports Android's ACTION_SEND functionality and with the optional *setComponent(ComponentName component)* statement, the component to handle the intent is explicitly defined. The automated compiling process is started by the *buildGame()* method as shown in listing 4.20.

Listing 4.20: Sends the compile request to JavaIDEdroid application.

```
1 private void buildGame(){
    Intent intent = new Intent(Intent.ACTION_SEND);
3   intent.setComponent(new ComponentName("com.t_arn.JavaIDEdroid
      ", "com.t_arn.JavaIDEdroid.MainActivity"));
    intent.putExtra("android.intent.extra.ScriptPath", Const.
      projectPath + gameName + "/0_build-prod.bsh");
5   intent.putExtra("android.intent.extra.ScriptAutoRun", true);
    intent.putExtra("android.intent.extra.ScriptAutoExit", true);
7   intent.putExtra("android.intent.extra.WantResultText", true);
    activity.startActivityForResult(intent, 123);
9   publishProgress(6);
}
```

An essential component of the compilation process is the Android library, Android.jar. This library file is available through the Android SDK and integrated in the editor application. The copied bash scripts for JavaIDEdroid are customised and reference this system library which is necessary for compilation.

Once exporting and building is finished, the user can finally play his game. The provided APK file is ready for deployment on all devices running Android versions greater than 2.3.3 alias "Gingerbread". Since it is only signed with a debug key, publishing the created game in the Google Play store is not possible [17]. The game can only be shared via email and installed as third party application.

4.8.3 Memory limitation on Android devices

As the paper "Memory management for Android apps" [8] from Google I/O 2011 states, every application has a maximum amount of memory it can allocate. Once the application hits this limit during execution, an "OutOfMemoryError" is thrown and the application crashes or exits. Most standard applications will not exceed this limit, but the compilation process is very memory intensive and the garbage collector can not free memory as fast as it is allocated by the compilation process. The heap limit is not a problem for the game studio application however, "JavaIDEdroid" suffers from this issue. There is an existing bug report on the project page since October 2011 [2], but it has not been solved until now. An appropriate solution to this problem would be to set the "largeHeap=true" option in the manifest file of the application. Unfortunately this is not implemented in the application hosted in the Google play store since this fix is only available for Android 3.x and newer.

The dimensions of the game editor are likely to not hit the maximum heap size most of the time. However, under certain constellations it is possible that the compilation process fails during execution. This can be caused by running many different applications concurrently which allocate too much memory and therefore the compilation process lacks the required resources. Furthermore, too many triggers and functionalities in a game can be a problem as they increase the *GameLogic.java* class file and therefore extend the compilation process. Besides, the maximum heap size is device dependent, it ranges from 16 MB to 48 MB, so the application may cause problems on devices with a lower limit than this maximum. The device used to test the application while developing the game studio, an Archos 101 G9, is limited to 50331648 Bytes or exactly 48 MB (50331648 / 1024 / 1024). This limit can be invoked by the following code part:

Listing 4.21: Requesting the maximum heap size of the executing device.

```
Runtime rt = Runtime.getRuntime();
long maxMemory = rt.maxMemory();
Log.v("onCreate", "maxMemory:" + String.valueOf(maxMemory));
```

Rooting the device is a possible way to circumvent problems with application memory as there are tools that allow to increase the maximum heap size of the device. An example is the VMHeapTool [23] from the Google Play store.

As the potential target group of "Android Game Studio" are persons with no programming skills, most of the users will not have the knowledge to use these tools, so the problems with heap space can not be prevented inherently. Hopefully, the developer of "JavaIDEdroid" will fix this problem in the future, but until then the game editor will be limited in its range of functions and implementing additional features like animations, sounds or particle systems will not be possible.

Part III.

Quality Management

Software Testing

5.1 Types of Software Testing

Software testing is an essential part of every software project to prevent faults and errors of the application. However, as stated by E. W. Dijkstra:

> *"Program testing can be used to show the presence of bugs, but never show their absence!"* [5]

This means that software testing is absolutely necessary to find bugs, but since not every can be found, the costs of testing has to be weight up against its benefits.

Maybe one of the most famous accidents of computer related failures are the Therac-25 accidents which claimed the live of at least five patients. The Therac-25 is a linear accelerator for treating cancer patients which had some severe software bugs, leading to an overexposure to radiation. Investigations of the accidents revealed that no or minimal software testing was performed and only the hardware of the device was checked [22]. Usually, software systems and especially Android applications are not life-threatening, but at least the image of the developer will suffer if bugs lead to unusable software or lost user data and fixing these problems afterwards is very expensive. Therefore, most companies have separate departments for software testing and employ many "Software Testers", which has become a full-time profession over the last years. The following subsections go more into detail about the testing process and illuminate the performed tests to ensure the correctness of the "Android Game Studio" application.

Software testing can be divided into four major categories, as figure 5.1 shows. The following overview describes the test components and goals of each individual category:

Unit Testing is performed to verify the correctness of a specific code section, meaning that it carries out the intended functionality. The test runs at the class level and includes testing the constructors and destructions of a class. [5]

Integration Testing verifies the software's interfaces against its subsystem design. The complete system can be tested at once, however, usually only small groups are tested to quickly identify defects based on the locality principle. [5]

System Testing determines if the system meets the specified functional and non-functional requirements. In this step, the entire system is tested. [5]

Acceptance Testing is performed by the customer after the software was delivered and may include typical transactions. This step demonstrates that the software meets the requirements is ready to be used. [5]

In large software projects it is crucial to extensively perform all of the mentioned tests, but since this project is carried out by a single person in a short time frame of three months, most of the testing was performed during the development. Furthermore, manual testing was preferred over automated testing. Indeed, there are many tools and frameworks designated for automatic testing such as "MonkeyRunner" or "JUnit", but as it is also very time consuming to write the test cases for these tools, it was decided to only verify the correctness of the application by manually testing it.

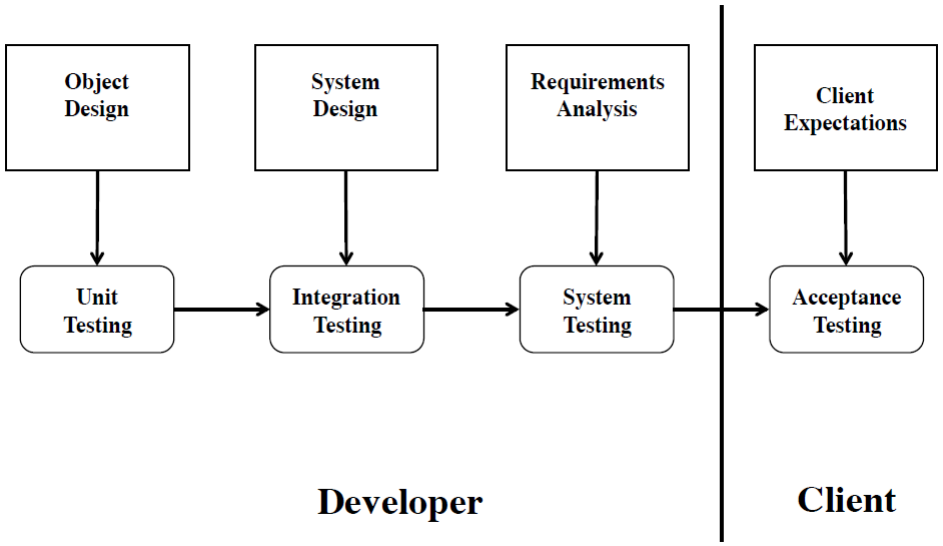


Figure 5.1.: Testing Activities and Models. [5]

5.2 Test Results

The following sections describe the performed tests and also present their results. Both tests, running a static code analysis tool and a user test, were executed in the final stages of this thesis to guarantee the correctness of the implemented project.

5.2.1 Code Analysis Tool

In order to identify potential types of errors the static code analysis tool "FindBugs 2.0" is used as Eclipse plugin. The program searches for bug patterns which are likely to indicate errors. The tool allows to set a rank report level which rangers from one to twenty (twenty being least severe) and only shows possible error sources above this threshold. With the standard setting of fifteen, which finds bugs "of concern", nearly no bugs are found. Only

activities writing to a static class are marked, but this can be safely ignored. More severe errors of the types “troubling”, “scary” and “scariest” were not found in the editor project [1].

To make sure not only the editor is free of bugs but also the created games, another test had to be run to approve this. Although the game is part of the project, the bug finder test can not be run on the asset folder. Therefore, a new Android project based on the game data was set up and then the test could be executed. The test did not find any serious bugs here, too.

5.2.2 User Test

To validate the correctness of “Android Game Studio” beta testers were asked to try the application and additionally, a test plan with precise instructions was created to test the same functions on different devices. The test plan includes different scenarios based on hypothetical user stories, which test possible sources of error in the application. All evaluated scenarios and their results are presented in the following tables:

Table 5.1.: Test scenario 1 - Creating a new game

Tested functionality	Device	Passed / Annotation
• create a new game • place object of every category • create all combinations of triggers • export and play game	Archos 101 G9, Asus Transformer TF101	Yes, working as intended

Table 5.2.: Test scenario 2 - Renaming and extending an existing game

Tested functionality	Device	Passed / Annotation
• rename the game • create new levels in the game • delete a level in the list of levels • export and play game	Archos 101 G9, Asus Transformer TF101	Yes, working as intended

Table 5.3.: Test scenario 3 - Deleting a game

Tested functionality	Device	Passed / Annotation
• delete a game • resume the game in main menu • load another game • wipe user data • resume game in main menu	Archos 101 G9, Asus Transformer TF101	Yes, working as intended

Table 5.4.: Test scenario 4 - Test events with exceptional values

Tested functionality	Device	Passed / Annotation
• create a game • add events and do not assign values • add events with coinciding functionalities and triggers • export and play game	Archos 101 G9, Asus Transformer TF101	Yes, working as intended

Table 5.5.: Test scenario 5 - Test functions with exceptional input

Tested functionality	Device	Passed / Annotation
• create a game with an already existing game name • add 50+ physical objects to the level • increase the level size to (100, 100) • export and play game	Archos 101 G9, Asus Transformer TF101	Yes, working as intended

Table 5.6.: Test scenario 6 - Background work

Tested functionality	Device	Passed / Annotation
• randomly go to an activity in the editor or game • open other applications and do something in background • wait several minutes • resume the editor or game activity	Archos 101 G9, Asus Transformer TF101	Yes, working as intended

Additionally, the created games were sent to different Android devices, including a Samsung Galaxy Wi8150 and a Google Nexus One. On both devices, no errors occurred and the games ran without problems.

Part IV.

Conclusion and Outlook

Conclusion

The goal of this thesis was to implement an easy to use game editor which allows users with actually no programming experience to create games and even share them with others. It was intended that the editor is similar to a construction kit and allows the user simply to click games together with different templates provided by default. Android was chosen as development platform since it is very open and offers a large variety of functions. Especially when delving deep in system functions like compiling and singing on the device, Android is the best and perhaps even the only choice of realising these goals.

Due to the great extent of the application - designing and implementing the editor and the created games as well as connecting them results in enormous outlay - not every function was implemented in detail, but rather targeted as a prototype. The best example for this issue is the object editor which holds a small amount of objects for testing in each category instead of providing the ability to dynamically create new objects with own settings like texture, physics options or points for collecting an item. However, all main functions have been realised and the editor is fully operable.

The thesis itself gives an overview about the design and implementation process of the application. It presents details about game programming in common and transfers this knowledge to the developed app. The research part at the beginning introduces the necessary background knowledge to understand the Android operating system and also introduces the basics of OpenGL ES and graphics programming on mobile devices. Furthermore, the design part explains details of the model-view-controller pattern, an essential design pattern for games and other software. Finally, the implementation part describes the main components of a game, such as graphics and physics engine, and illuminates the logic background of the editor and all of its components.

Finally, to guarantee and confirm the failure free operation of the implemented game editor functionalities, a user test was held.

Outlook

As the previous chapter concluded, the “Android Game Studio” application implements all planned functionality. Nevertheless, there are many possible improvements to enrich the editor and create great games with it.

First of all, there are some functions to simplify the way of dealing with the editor. A good example is to replace the buttons in the scene editor, which are responsible for object and level manipulation, by sliders so that large changes do not require the user to press a button multiple times, but instead control the same functionality with a slider. Another possible feature is a snap-to-grid functionality, which aligns objects with each other when they are closer together than a specific threshold. This will make level creation much more easy, because the user does not have to manually align the object to get a considerable output with no gaps between tiles.

There are also many possible extension for the game, for example an animation manager which allows to play animations and therefore create a vivid game environment. Moreover, implementing sounds could also be one of the next steps to improve the game editor, as well as adding a simple and basic artificial intelligence for opponents.

However, to make all these improvements possible, the compiler used has to be fixed by the developer. Since this is not likely to happen in the near future, it is more appropriate to run another path and replace the large physics engine with a less powerful implementation to decrease the lines of code and the code complexity of the project. The savings achieved are then available for additional features in order to release the application in the Google Play store.

Appendix

A Storage Medium

This medium contains the source code of the “Android Game Studio” application including all databases, bash files and assets.

Bibliography

[1] FindBugs™ - Find Bugs in Java Programs. http://findbugs.sourceforge.net/. [Retrieved: 09-October-2012].

[2] T. Arn. Issue 21: Out-of-memory condition when running dx on large projects. http://code.google.com/p/java-ide-droid/issues/detail?id=21. [Retrieved: 06-October-2012].

[3] T. Arn. JavaIDEdroid. http://code.google.com/p/java-ide-droid/. [Retrieved: 25-September-2012].

[4] H. Barra. 500 Million. https://plus.google.com/u/0/110023707389740934545/posts, September 2012. [Retrieved: 20-September-2012].

[5] B. Bruegge and A.H. Dutoit. *Object-Oriented Software Engineering: Using Uml, Patterns, and Java*. Prentice Hall, 2010.

[6] E. M. Buck, D.A. Yacktman, and R. Engel. *Cocoa Design Patterns*. mitp-Verlag, 2010.

[7] E. Catto. Box2D v2.2.0 User Manual. http://box2d.org/manual.pdf. [Retrieved: 09-September-2012].

[8] P. Dubroy. Memory Management for Android apps. http://static.googleusercontent.com/external_content/untrusted_dlcp/www.google.com/de//events/io/2011/static/notesfiles/MemoryManagement.pdf. [Retrieved: 06-October-2012].

[9] R. Eckstein. Java SE Application Design With MVC. http://www.oracle.com/technetwork/articles/javase/index-142890.html. [Retrieved: 28-August-2012].

[10] M. Gargenta. *Learning Android*. Oreilly Series. O'Reilly Media, Incorporated, 2011.

[11] Google. Android NDK. http://developer.android.com/tools/sdk/ndk/index.html. [Retrieved: 16-August-2012].

[12] Google. App Framework. http://developer.android.com/about/versions/index.html. [Retrieved: 17-August-2012].

[13] Google. Application Fundamentals. http://developer.android.com/guide/components/fundamentals.html. [Retrieved: 17-August-2012].

[14] Google. Building and Running. http://developer.android.com/tools/building/index.html. [Retrieved: 12-September-2012].

[15] Google. OpenGL. http://developer.android.com/guide/topics/graphics/opengl.html. [Retrieved: 03-September-2012].

[16] Google. Processes and Threads. http://developer.android.com/guide/components/processes-and-threads.html#Threads. [Retrieved: 13-September-2012].

[17] Google. Signing Your Applications. http://developer.android.com/tools/publishing/app-signing.html. [Retrieved: 11-October-2012].

[18] Google. SQLiteOpenHelper. http://developer.android.com/reference/android/database/sqlite/SQLiteOpenHelper.html. [Retrieved: 17-September-2012].

[19] Khronos Group. OpenGL ES Reference Manual. http://www.khronos.org/opengles/documentation/opengles1_0/html/. [Retrieved: 23-August-2012].

[20] Khronos Group. OpenGL ES Version 3.0. http://www.khronos.org/registry/gles/specs/3.0/es_spec_3.0.0.pdf, August 2012. [Retrieved: 23-September-2012].

[21] Tom Krazit. Google's rubin: Android 'a revolution'. http://news.cnet.com/8301-1023_3-10245994-93.html, May 2009. [Retrieved: 20-September-2012].

[22] N. Leveson. Medical Devices: The Therac-25. http://sunnyday.mit.edu/papers/therac.pdf. [Retrieved: 06-October-2012].

[23] Martin. VM Heap Tool (root only). https://play.google.com/store/apps/details?id=com.martino2k6.vmheaptool&hl=de. [Retrieved: 06-October-2012].

[24] Microsoft. Model-View-Controller. http://msdn.microsoft.com/en-us/library/ff649643.aspx. [Retrieved:02-September-2012].

[25] Microsoft. Publish/Subscribe. http://msdn.microsoft.com/en-us/library/ff649664.aspx. [Retrieved: 02-September-2012].

[26] H. Mosemann and M. Kose. *Android*. Hanser, 2009.

[27] A. Munshi, D. Ginsburg, and D. Shreiner. *OpenGL ES 2.0 Programming Guide*. OpenGL Series. Addison-Wesley, 2009.

[28] PricewaterhouseCoopers. Milliardenspiel – Hart umkämpftes Wachstum auf dem deutschen Videogames-Markt. http://www.pwc.de/de/pressemitteilungen/2012/milliardenspiel_hart_umkaempftes_wachstum_auf_dem_deutschen_videogames_markt.jhtml, August 2012. [Retrieved: 20-September-2012].

[29] M. Smithwick and M. Verma. *Pro OpenGL ES for Android*. Apressus Series. Apress, 2012.

[30] M. Zechner and R. Green. *Beginning Android 4 Games Development*. Apressus Series. Apress, 2011.

List of Figures

Listings

List of Tables