SDD

*System design document for **Falling***

Version: 1.0
Date: 2016-05-29
Author: Group 18

This version overrides all previous versions.

1 Introduction

1.1 Design goals

Falling is a VR game for Android. It must therefore run on Android, be able to get input from user movement, and give visual and audible output optimized for VR. This includes getting the user's/VR-headset's orientation, listening for screen touches, outputting sound located in 3D-space, and displaying correctly distorted visuals for use with a VR-headset.

For development purposes the game must also be able to run on desktop computers. When running the game on a desktop computer the game obviously can't take input from orientation so another alternative must be available for testing purposes.

To make sure *Falling* is robust and resilient to future changes special caution must be exercised. These changes could be (but are not limited to):

- Project scope
- Supported platforms
- Game engine

To do this properly the game must be completely separated from its platform requirements. For the same reasons the game code must also be easily extendible.

1.2 Definitions, acronyms and abbreviations

- *Falling* – the subject game/software.
- *VR* – virtual reality, a computer generated and simulated alternative world, where users can immerse as a part of that simulated reality/world.
- *VR-headset* – a headset that is mounted over the user's eyes that help the user with the illusion of being in the virtual reality.
- *SLOC* – source lines of code. A unit of code quantity that counts the number of lines of source code.

2 System design

2.1 Overview

Falling, as a piece of software, is designed much like real time games have been designed for decades. This implies the use of several mature design/architectural patterns like:

- Game loop
- Scene graph (Wikipedia, 2016c)
- Entity component system (Wikipedia, 2016b)

Although the application doesn't have strict implementations of the above patterns the ideas can be seen throughout the application. For example: every game loop iteration calls update on the game which propagates the update through the scene graph and every entity in the world (like *AirPlane*) is composed of components (like *Model*).

Below are some cornerstones of the application detailed. Both the "whys" and "hows" are discussed for each section so that it's clear for developers how to further develop and/or support the application in the future.

2.1.1 Main development language

Since Android is a requirement for *Falling*, it was chosen that the application should be written in Java. Although it is possible to write Android applications using other programming languages, it's most convenient to do so using Java, since it's the native language for the operating system.

2.1.2 Game engine

It was decided that a cross platform game engine (at least supporting Mac OS X, Windows, and Android) was required for creating the game. The reason for this is that a game like *Falling* requires a lot of services like file loading, asset parsing, an accessible OpenGL (ES) API, a 3D audio API, and much more. The game engine that was chosen was libGDX (Badlogic games, 2016), since it fulfills all of the requirements (including being written in Java) and design goals.

2.1.3 Virtual Reality support

Google Cardboard (Google, 2016) was used to incorporate VR into the application. Google Cardboard is by far the most accessible VR API available for Android in terms of price. It also has a Java API which is suitable since the rest of the application is developed in Java.

To improve the interoperability between libGDX and Google Cardboard a plugin for libGDX was used. The plugin, named “Libgdx CardBoard Extension” (Wei Yang, 2015), connects some of the most common concepts in the Google Cardboard API to Java classes that connect to the libGDX API. An example is the *CardboardCamera* class that encapsulates everything related to a camera rotating with the user’s head, and exposes it as a subclass of the libGDX class *Camera*.

2.1.4 NotificationManager

Gameplay and logic is mainly handled through the *NotificationManager* class and the *FallState* implementations (detailed in section 2.1.5). The *NotificationManager* class manages the application-wide event monitoring system (Wikipedia, 2016a).

The class is essentially a very simple implementation of an event bus. The class can be instantiated, but it also supplies a default instance for convenient usage. The main advantage of doing it this way is that it’s very accessible from anywhere in the code, which is important for the gameplay scripting.

The *NotificationManager* is used by code across the application for most time based events. This includes events like alerting subscribers of collision events and alerting the platform of sounds that should be played.

2.1.5 FallState

Jumper, which represents the character performing the parachute jump, is controlled through the *FallState* component. In order to handle movement and other things differently depending on whether, for example, the parachute is pulled or not, the application makes use of the Strategy pattern. *Jumper* has a reference to the current *FallState*, which is an interface implemented by *FreeFallState*, *ParachuteFallState*, etc, which then have different ways of, for example, calculating movement.

A *FallState* implementation is also responsible for changing the current *FallState* to another when it finds it suitable. In other words the current *FallState* functions as a finite-state machine.

2.1.6 RenderQueue

The class *RenderQueue* enables different parts of the application to request objects to render. This is important to the application since it makes use of non-trivial rendering techniques, like distance culling, and has to have full control around how and when objects should be rendered.

The *RenderQueue* class is implemented much like the *NotificationManager* in that it is instantiatable but has a default instance available. Only the default instance is used for the libGDX-based rendering in *Falling*.

2.1.7 GameMode

The *GameMode* interface encapsulates everything that is required to implement a game mode. A game mode can't directly affect the supplied *World* instance, since the *World* interface doesn't expose any functionality for that. This might seem restricting, but there are functionally no hindering limits. Through using the *RenderQueue* and the *NotificationManager* it can listen to and send notifications and render its own objects to the world, to fulfill its game mode purposes.

At the time of writing only one game mode exist: *BalloonGameMode*. The game mode spawns balloons and mines that the jumper can collide with, and accumulates a score differently depending on the types of the collisions (mine, balloon, or other).

2.2 Software decomposition

2.2.1 General

Falling is decomposed into the following packages (Figure 1):

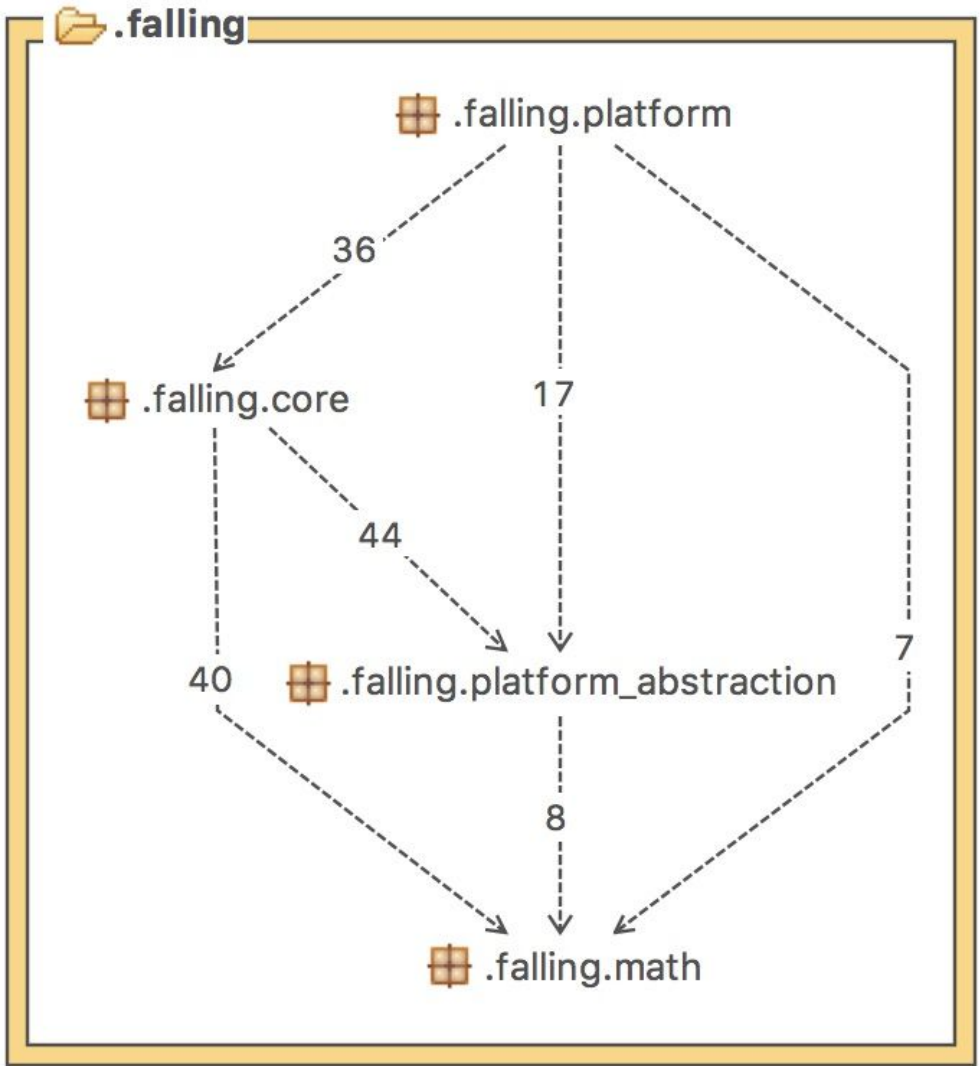


Figure 1 Package diagram for *Falling, including cross-package dependencies.*

The purpose of the packages are described in section 2.2.3 (Layering). UML class diagrams for each of the packages shown in Figure 1 can be found in the document “UML class diagrams” (see APPENDIX).

2.2.2 Decomposition into subsystems

Falling makes use of one service for all math related functionality. The service is entirely self-contained in the *math* package and is exposed through the *FallingMath*, *Matrix*, *Vector*, and *Rotation* classes. Since it is self-contained and is functional as a separate entity it can be seen as a subsystem of the application.

2.2.3 Layering

Falling is divided into three layers: a game layer, a platform layer, and an interface between them, the platform abstraction layer. This is done to make the application more robust and resilient (in line with the design goals).

The game layer is completely platform agnostic and completely separated from any platform dependent code. This layer takes up the majority of the application source code and consists of the top-level package *core*. In this layer *everything game related* is handled.

The platform layer is a minor part of the application (in terms of SLOC) in which all platform dependent code is defined. The platform layer defines the application entry point and from this calls into the game layer. As the game layer is platform agnostic and the platform layer defines the entry point, a great deal of flexibility is at hand for developers. For porting, or similar endeavours, vastly different platform implementations can be written to fulfill the platform needs without requiring any change at all in the game layer. The platform layer consists of the *platform* package. It is required that the platform layer fulfills all of its duties for the game layer (and in turn, the whole game) to function correctly. These requirements are defined in the document “Platform layer requirements” (see APPENDIX).

When the game layer communicates to the platform layer about what it wants (like assets that needs to be loaded, or audio that needs to be played, etc.) it makes use of the platform abstraction layer. The layer consists of the *platform_abstraction* package. Although this layer is platform agnostic –like the game layer–, its purpose is to mirror platform dependent components, much unlike the game layer, which only is concerned about gameplay.

2.2.4 Dependency analysis

A top-level package dependency graph is shown in Figure 1. There are no apparent issues with the internal application dependencies (like cyclic dependencies, etc.).

Other observations can be made from the figure: in the diagram it is clear that the entry-point platform layer only calls outwards, and services like the math package never calls out from itself. It also shows how the core and platform_abstraction packages are being used by the platform.

2.3 Concurrency issues

Not applicable: *Falling* does not make use of any type of concurrent processing.

2.4 Persistent data management

Not applicable: *Falling* does not store any persistent data.

2.5 Access control and security

Not applicable: *Falling* does not make use of or require internet access or special system access. All local files that are used within the application are distributed with the application itself.

2.6 Boundary conditions

Not applicable.

3 References

Badlogic Games (2016). *libGDX*. Accessed from https://libgdx.badlogicgames.com/ at 2016-05-25.

Google (2016). *Google Cardboard – Google VR*. Accessed from https://vr.google.com/cardboard/ at 2016-05-26.

Wei Yang (2015). *Libgdx-CardBoard-Extension*. Accessed from https://github.com/yangweigbh/Libgdx-CardBoard-Extension at 2016-05-25.

Wikipedia (2016a). *Event monitoring*. Accessed from https://en.wikipedia.org/wiki/Event_monitoring at 2016-05-25.

Wikipedia (2016b). *Entity component system*. Accessed from https://en.wikipedia.org/wiki/Entity_component_system at 2016-05-26.

Wikipedia (2016c). *Scene graph*. Accessed from https://en.wikipedia.org/wiki/Scene_graph at 2016-05-26.

APPENDIX

Platform layer requirements

For the application to function properly, the platform layer must perform some specific tasks and supply specific data to the game layer. These platform layer requirements are documented below in this document.

Requirements listed in ordered lists must be done in the specified order. Requirements listed in unordered lists can be done in any order.

At game setup

1. Construct a new *FallingGame* instance.
2. Load all resources specified by the current jump's *ResourceRequirements* instance. The platform layer is allowed to load resources any way it finds suitable. The only requirement is that it – at a later stage – can make use of the resources (see "While game is running" for information regarding how the resources should be used).

While game is running

The platform layer must maintain and run a *game loop*. The game loop is allowed to run at any speed, and can be either fixed-rate or variable-rate. For every iteration of the game loop the platform layer must do the following:

1. Collect user input however is suitable for the specific platform.
2. Set game input
    a. Call *setJumperHeadRotation(..)* on the *FallingGame* instance. The head rotation should be based on user input (collected in step 1).
    b. If the screen has been clicked (or similar), call *screenClicked(true)* on the *FallingGame* instance.
3. Clear the render queue: *RenderQueue.clear()*
4. Call *update(..)* on the *FallingGame* instance. The method argument should be a float *deltaTime* indicating how many seconds has passed since the method last was called: the frame time.
5. Perform rendering:
    a. If a PLAYER_HAS_CRASHED_EVENT_ID event has been received, clear the screen to black and perform all *GUITask*s.
    b. Else, clear the screen to the RGB color specified in *World.ATMOSPHERE_COLOR* and perform all *GUITask*s and all *RenderTask*s.

At any time while the game is running the platform must also listen to and react to notifications/events:

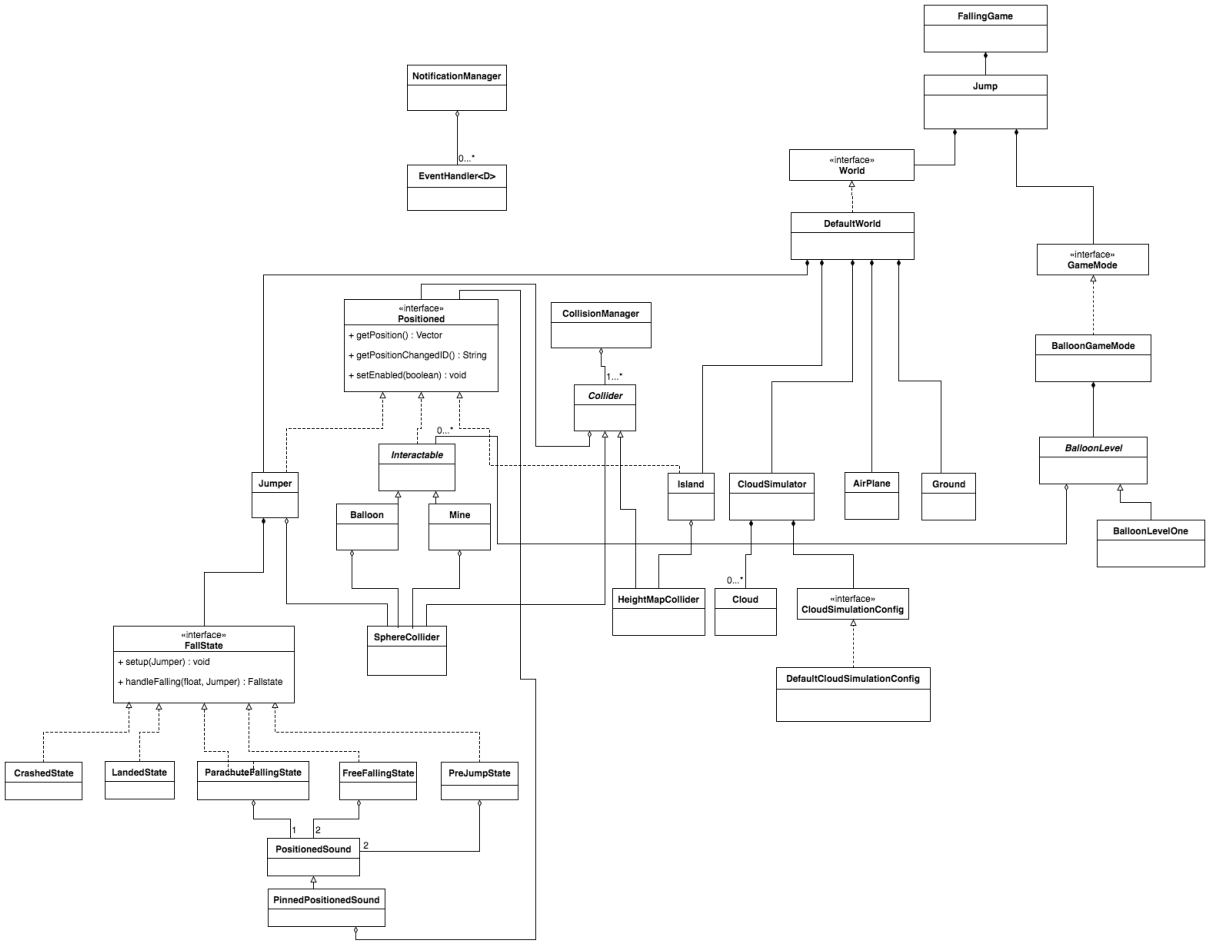
- PLAYER_HAS_CRASHED_EVENT_ID – Remember if this has been received. Its state is relevant to step 5 in “While game is running”.
- PLAY_SOUND_EVENT – Play the sound contained in the event data at the specified position and volume.
- LOOP_SOUND_EVENT – Loop the sound contained in the event data at the specified position and volume.
- STOP_SOUND_EVENT – Stop playing the sound contained in the event data.
- CHANGE_POSITION_SOUND_EVENT – Change the position of the sound contained in the event data.
- CHANGE_VOLUME_SOUND_EVENT – Change the volume of the sound contained in the event data.

At game cleanup

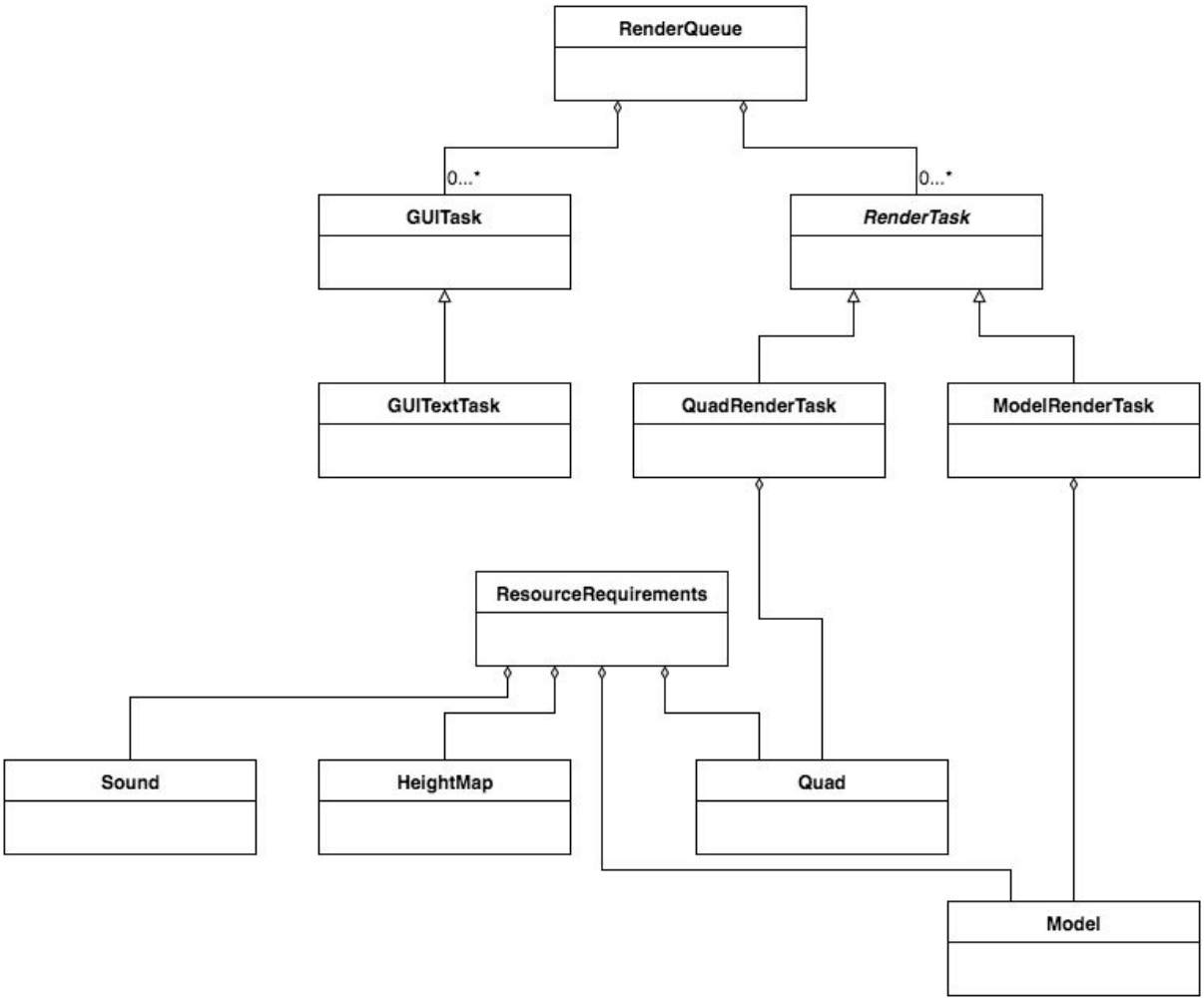
- Clean up any resources that have been loaded

UML class diagrams

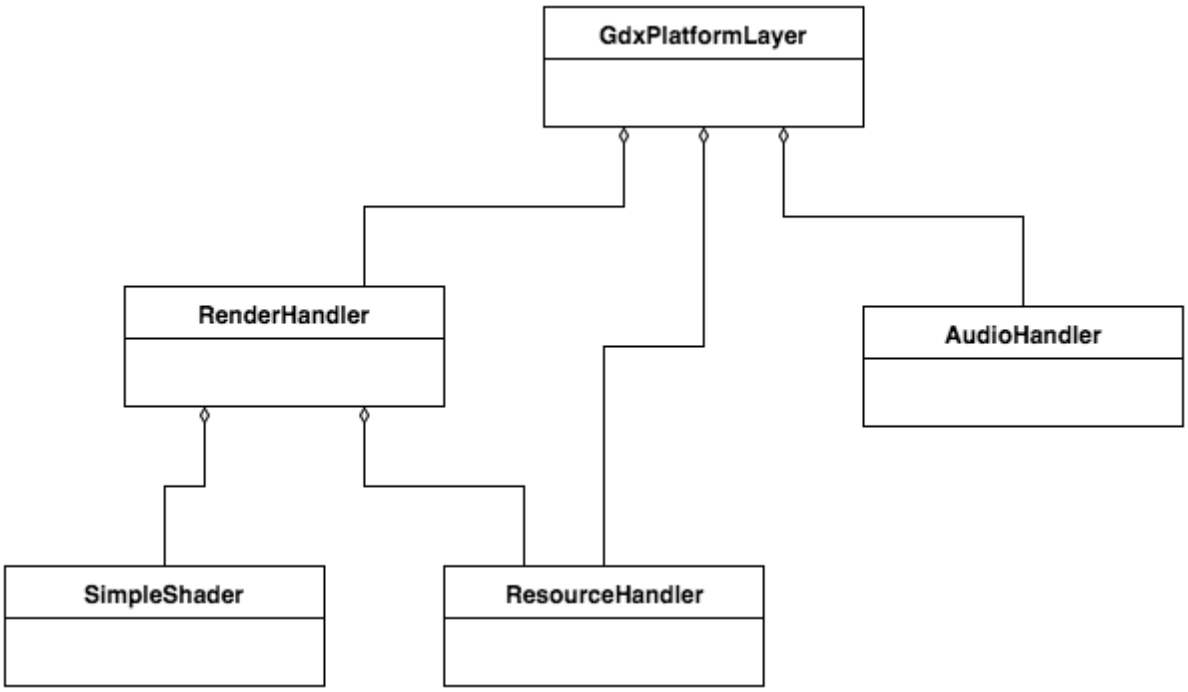
core package



***platform_abstraction* package**



platform **package**



math **package**