Falling

Utvecklingen av ett *VR*-baserat fallskärmshoppningsspel

Hannes von Essen
hvon@student.chalmers.se
Institutionen för Informationsteknik

Lisa Karlsson
karlisa@student.chalmers.se
Institutionen för Informationsteknik

Simon Moos
mooss@student.chalmers.se
Institutionen för Informationsteknik

Sabrina Samuelsson
sabsam@student.chalmers.se
Institutionen för Informationsteknik

Abstract

A *VR*-enabled skydiving game for Android was developed, aimed at creating an immersive and fun experience by incorporating realistic visuals, three-dimensional sound and a simulation based on real physics. Even though some compromises in realism were made in favor of gameplay, the final product does fulfill the project's purpose completely.

29 maj 2016
LSP310 V16
Kommunikation och ingenjörskompetens
Chalmers tekniska högskola

Innehåll

1 Inledning

Virtual reality (virtuell verklighet, hädanefter *VR*) är en *datorgenererad skenvärld i vilken användaren upplever sig vara och agera* (Nationalencyklopedin, 2016). En sådan värld kan simuleras på flera sätt och det har på senare tid utvecklats allt fler och enklare lösningar som gjort *VR* mer tillgänglig för gemene man. Så sent som 2014 släppte Google sin budgetvariant av *VR*-glasögon där användarens mobiltelefon används som skärm, något som möjliggör en betydligt lägre kostnad, och därmed större tillgänglighet, än vad som tidigare varit möjligt. Med en lättillgänglig teknik och en färsk marknad finns det stora möjligheter att vara innovativ.

En upplevelse som intresserar många är fallskärmshoppning, där fallskärmshopparen hoppar ut ur ett plan och faller genom himlen i 180 km/h. *VR*-tekniken möjliggör en djup inlevelse i den virtuella världen och kan därför tänkas ha bättre möjligheter att simulera känslan av ett fallskärmshopp än traditionella medier. Att upplevelsen är digital ger även frihet att frångå verkligheten för att skapa en mer spelliknande upplevelse. *VR*-tekniken har alltså potential att möjliggöra ett inlevelsefullt och roligt fallskärmshoppningsspel.

1.1 Syfte

Syftet är att ta fram ett *VR*-baserat fallskärmshoppningsspel för Android-telefoner. Applikationen ska vara *VR*-baserad på så sätt att den ger användaren en inlevelsefull känsla av att befinna sig i den värld som byggs upp av spelet. Utöver att ett fallskärmshopp simuleras ska även spelmekanismer införas för att göra upplevelsen mer utmanande och underhållande. Applikationen ska tas fram med smidig utbyggbarhet i åtanke.

1.2 Avgränsning

På grund av resursbegränsningar kommer applikationen endast vara en prototyp med en värld och ett spelläge, där det är möjligt att genomföra ett helt fallskärmshopp. Detta betyder däremot inte att koden ska vara undermålig eller hårdkodad; applikationen ska utvecklas med högsta

möjliga kvalitet.

2 Metod

Under utvecklingen av applikationen användes flera verktyg och tekniker för att stödja designprocessen, höja abstraktionsnivån och underlätta effektivt samarbete.

2.1 Domain Driven Design

I det inledande skedet av designprocessen användes arbetsmetoden Domain Driven Design (domändriven design, hädanefter DDD) för att ta fram klasser och den övergripande programstrukturen. DDD går ut på att programmets implementationsmodell ska vara starkt förankrad i motsvarande domänmodell (Evans och Fowler, 2003). För att möjliggöra denna starka förankring förepråkar DDD bland annat att programarkitekturen skiktas i flera abstraktionsnivåer där domänmodellen har ett eget lager. Detta för att lättare kunna fokusera på att uttrycka domänmodellen utan att behöva tänka på lägre nivåers ansvar. En sådan isolering användes i utvecklingsprocessen genom en skiktning i tre lager: ett spellager (domänlagret), ett plattformslager och en abstraktion av plattformen som ett gränssnittslager mellan dem (se bilaga ”SDD – System design document for Falling”).

Ett *Requirements Analysis Document* (kravanalysdokument, hädanefter RAD) skrevs för att assistera den domändrivna utvecklingen. Ett RAD är ett dokument där de krav som ställs på applikationen definieras innan utvecklingen börjar. I Falling-projektets RAD definierades de mest grundläggande kraven som applikationen skulle vara tvungen att uppfylla.

2.2 libGDX

Då projektet behövde utvecklas inom en begränsad tidsram (ungefär 8 veckor) var det behövligt att abstrahera bort vissa detaljer kring datorgrafik, platformsberoende, etc. Detta hanterades av kodbiblioteket

libGDX. Det finns flera anledningar till varför libGDX valdes över andra alternativ:

- Projektet krävde att Java skulle användas. libGDX är skrivet i Java, och är därmed kompatibelt
- Spelet behövde kunna köras på Android, men för en effektiv utveckling var det även viktigt att snabbt kunna testköra på datorn, något som libGDX möjliggjorde tack vare sitt multiplattformsstöd
- *VR*-teknik behövde kunna användas i enlighet med projektets syfte, vilket libGDX kunde interagera med på ett bra sätt (se sektion 2.3)
- Det är en välanvänd och váltestad produkt

Två möjliga alternativ som däremot ej valdes är jMonkeyEngine och Ogre3D. Den huvudsakliga anledningen till att libGDX valdes över dessa två alternativ var det smidiga *VR*-samspelet som libGDX möjliggör (se sektion 2.3).

2.3 Google Cardboard

För att kunna uppfylla syftet krävdes någon typ av *VR*-teknik. Sådan finns i flera prisklasser med olika uppsättningar av funktioner, men för optimal tillgänglighet för användare valdes Google Cardboard. Det som krävs för att köra ett program genom Google Cardboard är en mobil, samt ett Google Cardboard *VR*-headset som kan införskaffas för under hundralappen. Andra alternativ som Oculus Rift, Oculus Gear VR, och HTC Vive kostar alla tusentals kronor vilket var utanför budgeten för projektet samt riskerade att göra spelet mycket mindre tillgängligt för de flesta användare.

För att kunna använda Google Cardboard erbjuder Google ett gratis API (Google, 2016a). För detta projekt användes mer specifikt ett instickspogram till libGDX vilket kombinerar Google Cardboard-API:t med libGDX. Detta var viktigt för utvecklingsprocessen då det ytterligare kunde förkorta utvecklingstiden. Google Cardboard-API:t möjliggör registrering av användarens huvudrotation, beräkning av korrekt bild för vardera öga samt uppspelning av tredimensionellt positionerat ljud. Google Cardboard registrerar endast huvudets rotation; ingen information om positionen i rummet finns tillgänglig.

2.4 Versionshantering

För att effektivt kunna skriva kod användes Git som versionhanterare. GitHub användes som värd för Git-repository:t (projektförrådet) och dess verktyg användes för att stödja utvecklingsprocessen. Dessa verktyg inkluderar statistiska analyser och verktyg för att göra så kallade *pull-request*s lättare.

Arbetsflödet *Feature Branch* (Atlassian, 2016) användes för att smidigt kunna arbeta på flera funktioner samtidigt. Arbetsflödet innebär att varje ”feature” (ny funktionalitet, buggfix, eller liknande) löses i en separat gren i repository:t vilken senare, när den är färdigställd, sätts ihop med huvudgrenen igen. Detta arbetsflöde gör det lätt för utvecklare att arbeta med något specifikt i en separat miljö, utan att kompromissa samarbete och utan att förvirra Git-historiken.

3 Teori

Som teoretisk bakgrund till projektet användes mönster för spelutveckling, Google's egna riktlinjer för utformning av *VR*-applikationer, litteratur om de olika faserna i ett fallskärmshopp samt fysikaliska formler relevanta för hoppet.

3.1 Datorspel med realtidssimulation

Alla datorspel har olika sätt att driva simulationen framåt, rent tekniskt. För ett turbaserat spel, till exempel, är det vanligt att använda ett händelsebaserat system. För spel med realtidssimulation är det däremot nödvändigt att kontinuerligt uppdatera tillståndet hos världen och rita upp det nya tillståndet, ofta så snabbt som möjligt. Denna kontinuerliga uppdatering sköts normalt av en så kallad *game loop* (Nystrom, 2014, s. 123-138).

För varje iteration av en game loop utförs vanligtvis följande steg:

1. Hämta och samla all relevant indata (*user input*)

2. Be alla komponenter som ska simuleras att genomföra nästa uppdatering med hjälp av indatan

3. Rendera världen

Indatan samlas vanligtvis på någon central plats där den är lättåtkomlig för alla som vill läsa av den. Något som kan vilja läsa av informationen är ett spels karaktär, vilken förmodligen (under uppdateringssteget) rör sig utifrån användarens knapptryckningar.

3.2 Virtual Reality (*VR*)

VR är en datorgenererad och -simulerad virtuell verklighet som användare känner sig som en del av. Då *VR* är mer av en filosofi än en specifik teknik finns det flera olika sätt för utvecklare att göra detta på. En av de mer populära teknikerna är att använda ett så kallat *VR-headset*. Ett *VR*-headset monteras på användarens huvud likt glasögon och möjliggör användaren att se den simulerade världen från en unik synvinkel för vart öga, vilket ger illusionen av 3D. Spelaren kan ”se sig omkring” i den virtuella världen genom att *VR*-headset:et mäter hur huvudet rör sig.

I *VR* är det lätt att användaren upplever så kallad *simulator sickness* (simulatoråksjuka) om inte speciellt fokus läggs på att undvika det. Simulator sickness kan uppstå när det finns skillnader mellan vad användaren upplever och gör i verkligheten och vad hen ser i den virtuella världen (Google, 2016b). En vanlig orsak är att det tar för lång tid att rendera den virtuella världen, vilket leder till att rörelser blir ojämna trots att användaren rör sig stadigt i verkligheten. Därför är en hög bildfrekvens särskilt viktig för *VR*-applikationer.

3.3 Fallskärmshoppning

Fallskärmshoppning görs normalt ur ett flygplan på uppemot 4000 meters höjd. Hopparen faller fritt genom luften och når efter ungefär 10 sekunder en konstant maxhastighet runt 60 m/s (Illustrerad Vetenskap, 2001).

De främsta fysikaliska krafter som verkar under hoppet är tyngdkraft och luftmotstånd. Tyngdkraften accelererar hopparen nedåt tills dess att

luftmotståndet $F_d = \frac{1}{2} C \rho A v^2$ (Ekholm, Fraenkel och Hörbeck, 2013) blir lika stort som tyngdkraften, och därmed resulterar i en konstant hastighet. Luftmotståndet beror på luftmotståndskoefficienten (C), luftens densitet (ρ), tvärsnittsarean (A), samt fallhastigheten (v).

På grund av luftmotståndet har hopparen stor möjlighet att påverka sin rörelse under fallet. Minsta ändring av kroppens lutning förändrar luftmotståndets fördelning på kroppen och kan skapa hastigheter på över 10 m/s i alla sidoriktningar (Nyqvist, 2009, s. 141).

Vid 700-1200 meters höjd, efter ett frifall på ungefär en minut, fälls fallskärmen ut och får genom det ökade luftmotståndet hopparen att retardera till 3-4 m/s nedåt med en mer konstant framåthastighet på 8-10 m/s (Nyqvist, 2009, s. 59, 69). Kontrollen över rörelsen genom luften ser nu annorlunda ut; till skillnad från att som i fritt fall ha relativt omedelbar rörelsefrihet i alla riktningar rör sig hopparen nu åt det håll fallskärmen för tillfället är riktad, och kontrollerar denna riktning stegvis genom att hålla nere fallskärmens vänstra eller högra bromshandtag. Eftersom en del av lyftkraften går åt till svängen ökar då sjunkhastigheten (Nyqvist, 2009, s. 71).

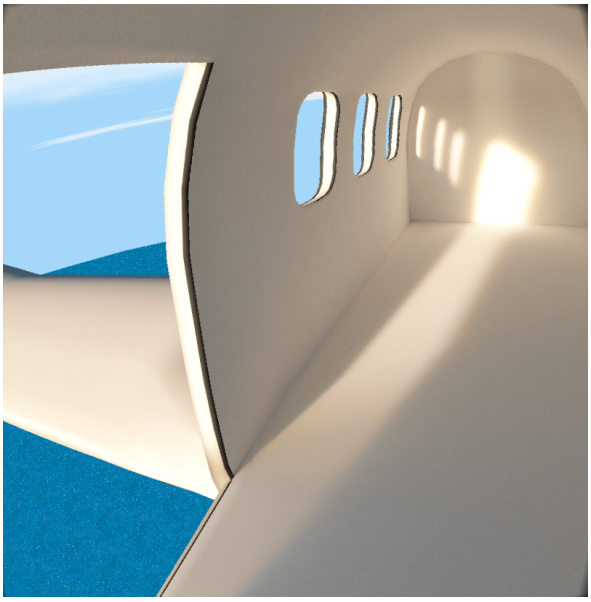
Hoppet avslutas med att fallskärmshopparen landar genom att dra i båda bromsarna strax ovanför marknivå (en s.k. *flär*) vilket bromsar upp fallet och ger en mjuk landning (Nyqvist, 2009, s. 87).

4 Resultat

Huvudprodukten är den Android-applikation som utvecklats. Det finns även en skrivbordsversion som användes för att underlätta testning under utvecklingsprocessen.

När spelaren startat applikationen står hen i ett flygplan (se figur 1), och kan ”hoppa” ut ur det genom att luta sig nedför kanten. Då börjar själva fallet och spelaren rör sig nedåt genom världen. I luften finns ballonger som spelaren ska samla på, minor att undvika, samt dekorativa moln. Alla dessa objekt fungerar dessutom som indikatorer på hur spelaren rör sig genom rummet.

Medan spelaren faller nedåt kan hen förflytta sig horisontellt genom att se



Figur 1: Spelet startar i ett plan över havet.

sig omkring, då hen får en acceleration i den riktning hen ser åt tills en maxhastighet har nåtts.

Fallskärmen kan dras genom ett tryck på mobilens skärm. Hastigheten nedåt minskar då och spelaren börjar röra sig i den riktning hen såg åt när fallskärmen vecklades ut. Riktningen kan justeras genom att luta huvudet mot vardera axel, och det är även möjligt att se sig omkring utan att styrningen påverkas, precis som i verkligheten.

När spelaren når ön, om fallskärmen är utvecklad, landar hen och spelet är slut. Efter en mjuk inbromsning visas spelarens poäng upp på skärmen tillsammans med ett meddelade om att spelet kan startas om.

4.1 Simulering

Beräkningarna av fallhastighet och -acceleration görs enligt de verkliga fysikaliska principerna för tyngdkraft och luftmotstånd med konstanter satta till typiska värden från verkligheten. Dock fick vissa av konstanterna justeras för att sluthastigheten för fritt fall skulle hamna nära den vanliga maxhastigheten av 60 m/s. Det ökade luftmotståndet när fallskärmen fälls ut simuleras genom att tvärsnittsarean ökas, vilket orsakar en kraftig

inbromsning av fallet. Den ökade sjunkhastigheten vid svängar simuleras genom en lätt minskning av tvärsnittsarean. Styrningen under fritt fall är skapad för att vara lättförståelig och intuitiv, men har ingen stark förankring i ett faktiskt fallskärmshopp.

För att spelaren ska få en känsla av att hen rör sig genom rummet lades moln till. När spelaren står i planet rör sig molnen för att ge en illusion av att planet rör sig. Denna lösning gör att det virtuella rummet kan förbli relativt litet då spelarens position inte ändras innan hen hoppat ur planet.

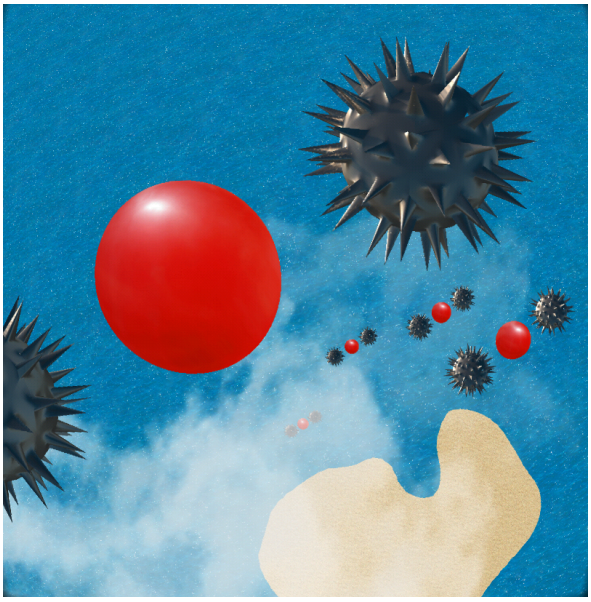
Spelet är tänkt att spelas med hörlurar för full inlevelse och innehåller flera 3D-positionerade ljudeffekter från den omkringliggande miljön, såsom flygplanets brummande på insidan av planet, den starka vinden när spelaren lutar sig ut, och ett dynamiskt vindljud under det fria fallet som blir skarpare ju mer spelaren svänger.

Då synintryck är en stor del av *VR*-applikationer lades det fokus på att få 3D-modellerna i spelet att se trovärdiga ut. Flygplanet som spelaren hoppar ut från är baserat på ritningar för en PAC P-750 XSTOL (Pacific Aerospace, 2016), ett vanligt fallskärmshoppningsplan som bland annat används av Fallskärmsklubben Cirrus Göteborg.

För att få en realistisk ljussättning av 3D-modellerna och samtidigt behålla en god prestanda användes *baking*, en slags förhandsrendering som innebär att ljussättningen av en modell beräknas på förhand och sparas i en bildfil. För att rita modellens färg och ljussättning använder sig programmet sedan av informationen i bildfilen istället för att beräkna den i realtid, något som möjliggör fler bildrutor per sekund. För flygplansmodellen utförde en fysikbaserad renderare beräkningar av ljusstrålar i flera timmar för att få en så realistisk ljussättning som möjligt, och fånga känslan av solen som skiner ovanför molnen.

4.2 Spelmekanismer

Applikationen innehåller ett spelläge, där spelaren samlar på ballonger och undviker minor (se figur 2). Så länge spelaren inte rör vid någon av minorna byggs en kombo upp för varje ballong som plockas. En högre kombo ger fler poäng för varje ballong som plockas upp. För första ballongen hen samlar får spelaren 100 poäng och med varje ytterligare ballong hen samlar ökar poängen hen får med 100. Om hen åker in i en



Figur 2: Spelelement ger hopparen en utmaning på vägen ner.

mina blir detta nollställt. För att indikera hur lång kombo spelaren har samlat på sig, och därmed hur mycket poäng hen får, används ljud; en ton hörs när en ballong samlas, och ju högre kombo spelaren har desto ljusare blir tonen. Om spelaren åker på en mina hörs en explosion och tonen för nästa ballong blir återigen mörk för att visa att kombon är bruten.

För att klara av spelet måste spelaren landa på ett korrekt sätt: på land efter att ha dragit ut fallskärmen. Om spelaren når marken innan hen dragit fallskärmen så krashar hen och förlorar spelet, detsamma om hen landar i vattnet. Om detta sker blir skärmen svart, spelaren får reda på vad som gått fel och får möjligheten att starta ett nytt hopp.

För att få applikationen att fungera som ett underhållande spel för användaren var en del anpassningar nödvändiga. Storleken av de objekt som finns att interagera med under hoppet är överdimensionerade för att de ska vara enklare att se från ett större avstånd och vara lättare att träffa.

Flygplanets höjd över marken sattes med hänsyn till längden på spelet snarare än vilken höjd ett faktiskt fallskärmshopp skulle ske från, och höjden som fallskärmen måste dras på är mycket lägre än i verkligheten för att göra spelet mer förlåtande och förkorta den relativt händelsefattiga fallskärmsdelen.

4.3 Vidareutveckling

Applikationen arbetar mot gränssnitt vad gäller spelläge (GameMode) och värld (World) för att underlätta utökning av spelet.

I det existerande spelläget, BalloonGameMode, beror utplacering av interagerbara objekt (Interactable), såsom minor och ballonger, på vilken BalloonLevel som valts, då detta sköts av abstrakta metoden create() i BalloonLevel. Således kan nya nivåer göras då varje konkret implementation av metoden kan skapa en egen konfiguration av objekt. När dessa skapas ska de läggas till i en lista med Interactables för att kunna ritas ut, vilket gör att fler typer av objekt, med olika utseende och/eller effekter, kan skapas och läggas till så länge de är Interactable.

Koden har en tydlig separation mellan spellogiken och den plattformsspecifika koden, vilket gör att det skulle vara relativt lätt att byta ut detta plattformslager för att stödja en annan *VR*-teknologi.

5 Diskussion

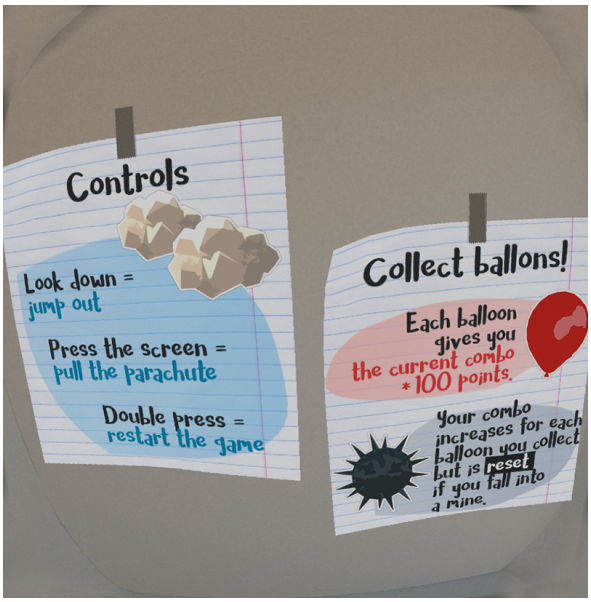
I enlighet med syftet är det intressant att se närmare på vilka anpassningar som gjorts för att uppnå en bra balans mellan realism, spelmekanismer och prestanda för att skapa ett spel som är inlevelsefull och utmanade. Idéer kring vidareutveckling är också av intresse.

5.1 Anpassningar gällande simulering och spelupplevelse

För att kunna göra ett inlevelsefullt och underhållande *VR*-spel och undvika *simulator sickness* krävs bra prestanda. För att uppnå detta har mängden grafiska element som visas samtidigt fått hållas nere, till exempel genom att låta spelelement dyka upp på ett visst avstånd istället för att vara synliga under hela spelet. Detta gör visserligen spelet mindre detaljerat, och att ha ett större antal moln hade kunnat bidra till en tydligare fartkänsla. Begränsningen i hur långt fram i nivån spelaren ser kan på samma gång ses som en spelmekanism som ökar spänningen, och vaksamheten hos spelaren, eftersom nya föremål kan dyka upp när som helst.

Det är också viktigt att rörelser känns naturliga och intuitiva för användaren. Därför togs extra hänsyn vid utveckling av styrkontrollerna till att när spelaren svänger, under fallskärmsdelen, ska kameran inte svänga mer än vad som känns bra. Testning gjordes löpande, och saker som hur snabbt kameran svänger, till vilken grad, och hur länge den fortsätter svänga anpassades utifrån tanken att det skulle vara bekvämt för användaren. Sättet att svänga valdes för att efterlikna verklig styrning av en fallskärm.

En viktig faktor för illusionen om att den simulerade världen är verklig inte bryts är användningen av ljud som indikator på vad som händer (både upplevelsemässigt och vad gäller spelmekanismer), istället för att visa text på skärmen. Att använda text som sådan indikator är vanligt i många spel, men i den virtuella verkligheten skulle text direkt på skärmen bryta illusionen av en riktig värld. Därför togs beslutet att göra spelinstruktionerna till fysiska papper i världen, istället för att ha dem som skärmtext (se figur 3).



Figur 3: Spelets instruktioner sitter upptejpade på en av planets väggar.

Det har alltså dels gjorts kompromisser hos simuleringens verklighetshetstrogenhet till förmån för en bättre spelkänsla och bekvämlighet hos användaren, och dels gjorts anpassningar av traditionella spelelement för att bättre passa *VR*-formen.

5.2 Fortsatt utveckling

Det finns mycket som kan läggas till och förbättras med applikationen – många av de idéer som funnits under arbetets gång kunde på grund av tidsbrist inte genomföras.

Till exempel kan det implementeras ett nivå-system där spelaren kan låsa upp fler nivåer genom att klara av de redan upplåsta, och där varje nivå är annorlunda på något sätt. De kan se olika ut, med olika utplacering av ballonger, eller ha fler sorters objekt med nya funktioner. Det skulle också kunna finnas andra spellägen, där ett exempel kunde vara att spelaren får poäng baserat på precision vid landning. På grund av hur koden runt GameMode och World strukturerades skulle dessa variationer vara enkla att göra.

Med ett nivåsystem skulle det också vara passande med något slags meny, där spelaren kan välja nivå, se över sina framsteg, och vilka poäng som fåtts på de olika nivåerna. För detta behövs möjlighet att spara data mellan spelsessioner.

En annan intressant funktion skulle vara ett flerspelarläge, där flera personer med varsina *VR*-glasögon skulle kunna spela tillsammans och se varandra falla i luften för att till exempel göra formationshopp.

Möjligheten att lätt byta ut den plattformsspecifika koden kan tänkas vara särskilt användbar för just en *VR*-applikation eftersom marknaden för *VR*-tekniker är väldigt snabbföränderlig; den i projektet använda Google Cardboard-tekniken kommer förmodligen vara utdaterad om bara några år. Att denna återanvändbarhet av koden uppnåtts kan nog till stor del tillskrivas den valda utvecklingsmetoden DDD och dess principer om en skiktad programarkitektur, något som hjälpt till att tidigt i projektet separera spellogiken från plattformen.

6 Slutsatser

Det har utvecklats ett Android-spel som spelas med ett *VR*-headset, där spelaren hoppar fallskärm ut ur ett flygplan. Diverse effekter för att göra det mer inlevelsefullt finns, såsom simulerat vindljud då spelaren faller, och banan med ballonger ger spelaren en utmaning att arbeta med och

underhållas av. Koden är designad på ett sätt som möjliggör smidig vidareutveckling och överförning till framtida *VR*-tekniker.

Trots att vissa kompromisser gjorts kring simuleringens verklighetsanknytning är projektets krav på inlevelsefullhet, spelmekanik och utbyggbarhet väl uppfyllda och syftet kan därför anses vara uppnått.

Referenser

Atlassian. (2016). Comparing workflows. Hämtad 25 april 2016, från https://www.atlassian.com/git/tutorials/comparing-workflows

Ekholm, P. U., Fraenkel, L. & Hörbeck, S. (2013). *Formler & tabeller i fysik, matematik & kemi för gymnasieskolan*. Konvergenta.

Evans, E. & Fowler, M. (2003). *Domain-driven design: tackling complexity in the heart of software 1st edition*. Prentice Hall.

Google. (2016a). Cardboard. Hämtad 26 maj 2016, från https://developers.google.com/cardboard/

Google. (2016b). Designing for google cardboard. Hämtad 26 maj 2016, från http://www.google.com/design/spec-vr/designing-for-google-cardboard/

Illustrerad Vetenskap. (2001). Hur snabbt faller man i fritt fall? Hämtad 26 maj 2016, från http://illvet.se/fysik/naturlagar/hur-snabbt-faller-man-i-fritt-fall

Nationalencyklopedin. (2016). Virtuell verklighet. Hämtad 26 maj 2016, från http://www.ne.se/uppslagsverk/encyklopedi/l%C3%A5ng/virtuell-verklighet

Nyqvist, A. (2009). *Hoppa fallskärm*. Himmelsdyk.

Nystrom, R. (2014). *Game programming patterns*. Genever Benning.

Pacific Aerospace. (2016). Specifications. Hämtad 27 maj 2016, från http://www.aerospace.co.nz/aircraft/p-750-xstol/specifications

Bilagor

SDD

*System design document for **Falling***

Version: 1.0
Date: 2016-05-27
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 to 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 connects 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 of 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 controll 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):

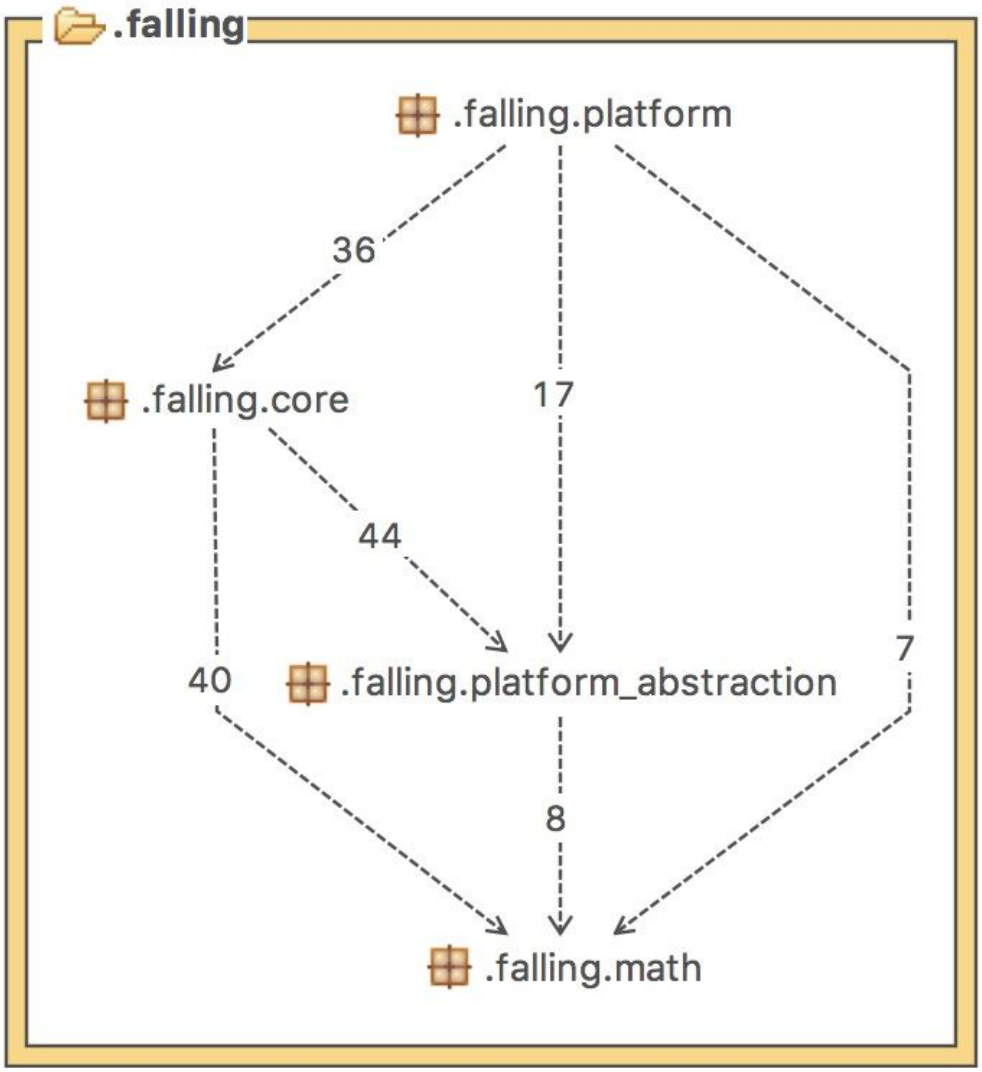


Figure 1 Package diagram for *Falling, including cross-package dependencies.*

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.