CHALMERS

Kandidatrapport
Unfair

- Ett actionorienterat 2D-plattformsspel skapat i Unity Personal

Anton Hallin
Maria Larsson
Johannes Magnusson
Jonathan Nilsson
Henrik Ståhlsparre
Mattias Åkesson

Kandidatarbete DATX02-15-26
Chalmers Tekniska Högskola
Institutionen för Data- och Informationsteknik
Göteborg, Sverige 2015

Anton. Hallin
Maria. Larsson
Johannes. Magnusson
Jonathan. Nilsson
Henrik. Ståhlsparre
Mattias. Åkesson

Examiner: Arne. Linde

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden May 2015

Förord

Rapporten behandlar ett kandidatarbete på Chalmers inom data- och informationsteknik. Arbetet är utfört av sex studenter som går sitt tredje och fjärde år på civilingenjörsutbildningen inom datorteknik och informationsteknik. Kandidatarbetet har utförts under en termin och omfattar 15 hp.

Vi vill tacka vår handledare Josef Svenningsson för vägledning och stöttning genom hela arbetet. Vi vill även tacka Linnéa Harrison på DICE för tips och rådgivning. Ett stort tack skänkes också till Fackspråk för språkliga verktyg och handledning samt till de personer som testat och utvärderat produkten.

Sammandrag

Spelindustrins marknadsandel har under de senaste decennierna upplevt en stadig tillväxt och inga tecken visas på att avbryta den trenden. Med expanderande målgrupper och ökad förekomst världen över kan man med säkerhet säga att den blivit en betydande aktör på den globala ekonomiska marknaden. Detta projekt ämnade besvara frågan: Hur kan spelmekaniken för ett actionorienterat 2D-platformsspel implementeras i utvecklingsmiljön Unity Personal. Detta åstadkoms genom att utföra en studie i speldesign och Unity-baserad spelutveckling och applicera resultaten av dessa i skapandet och implementationen av ett spel. Resultatet är en prototyp av ett simpelt 2D-actionplatformsspel med exempel på hur ett antal olika spelmekaniska moment kan implementeras. Den mest berikande erfarenheten från projektet är förståelsen för den komplexitet som återfinns i utvecklingsprocessen av ett spel. Många misstag kan begås i såväl planeringen som utförandet av en sådan process; några av dessa belyses i den här rapporten.

Abstract

Over the past decades the gaming entertainment market has been growing steadily and is showing no signs of discontinuing this trend. With expanding target demographics and increased gaming prescence all over the world it is safe to say it has become a significant actor on the global economical market. This thesis aimed to answer the question: How can game mechanics for a 2D action platform game be implemented in the development platform Unity Personal. This is accomplished by performing a study in game design and Unity based game development, and using the results of these to create and implement a game. The result is a prototype of a simple 2D platformer with examples of implementation methods for a number of game mechanics. What became the most important learned experience from this project is the intricacy of the game development process. Many missteps can be made in both the planning and execution of such a process, some of which will be elucidated in this paper.

Innehåll

1 Inledning

Det finns inga säkra siffror att tillgå för att fastställa den ekonomiska omfattningen av dator- och tv-spelsindustrin. I en marknadsundersökning utförd av Newzoo uppskattades den globalt omsätta 81.5 miljarder amerikanska dollar (USD) 2014 [1]. Gartner beräknar samma siffra till 101.6 miljarder USD [2] och Reuters siffra för 2013 var 66 miljarder USD [3]. Oberoende av variationerna konstateras spelindustrin vara en enorm ekonomisk aktör. En industri som växt kraftigt de senaste åren och visar ingalunda tecken på att bromsa. I en rapport från 2014 från ESA (Entertainment Software Association) konstateras att den årliga tillväxttakten för USA:s spelindustri mellan åren 2009-2012 (9.7%) var mer än dubbelt så hög som den rapporterade tillväxttakten för hela landets ekonomi under samma period (4.2%) [4]. Newzoo uppskattar en årlig marknadstillväxt på 8.1% mellan åren 2012-2017 [5], och Statista beräknar samma siffra mellan 2013-2018 till 6.2% [6].

Det ekonomiska läge som marknaden upplever påvisar också hur spelande blivit en stor del i många människors liv. Detta stöds också i en undersökning av Nielsen, där framgår att genomsnittstiden som personer som spelar spel spenderar på spel i veckan ökat med 24% mellan år 2011-2013 från 5.1 till 6.3 timmar[7]. En vanlig villfarelse beträffande spelindustrin är vem som konsumerar dess produkter. ESA beräknade i en undersökning att genomsnittsåldern för spelkonsumenter i USA år 2014 var 31 år och att 48% av dessa var kvinnor [8]. Industrin är långt förbi dess yngre år, då spel ansågs vara något huvudsakligen pojkar underhöll sig med under deras uppväxt. Idag sträcker marknaden sig till en mycket bredare demografisk grupp och är något vem som helst kan engagera sig i.

En klassisk spelgenre som uppskattas av många är 2D-plattformsspel. I spelindustrins yngre dagar, när spelandet flyttade från arkadhallar till konsoller i hemmet, dominerade actionorienterade 2D-plattformsspel på marknaden med spel som Super Mario Bros(Nintendo), Metroid(Nintendo), Mega Man(Capcom) och Sonic the hedgehog(Sega) [9]. Med 3D-motorernas framfart under mitten av 90-talet avtog genrens popularitet, både bland konsumenter och producenter. Men med den kraftiga tillväxt spelindustrins indiemarknad, ofta kännetecknad av sin inspiration från de äldre spelgenerationenrna, har genren återfått sin relevans [10]. Spel som Super Meat Boy(Team Meat) och Braid(Number None Inc.), båda 2D-plattformsspel, är exempel som varit essentiella i indiemarknadens tillväxt [11][12][13]. Karaktäristiskt för dessa är ett högt tempo, pussel och utmanande svårighetsgrad.

Indiemarknadens uppgång har bidragit kraftigt till utvecklingsmiljön Unitys ökade popularitet. Samtidigt har Unity varit essentiell i samma uppgång [14]. Unitys användarbas övergick två miljoner 2013, en fördubbling gentemot 2012 [15]. En stor faktor för den här utvecklingen är erbjudandet av gratisversionen, Unity Personal, vilket gör utvecklingsmiljön lättillgänglig för aspirerande spelutvecklare. Som kontrast kostade en licens för Unreal Engine 4, före 2 mars 2015, $19 per månad samt 5% av intäkterna från spel utvecklade i den miljön [16]. Efter detta datum valde utvecklarna dock att ta bort månadsavgiften, troligtvis som svar på Unity's framgång. En annan fördel med Unity är den stora mängd undervisningsmaterial som finns att tillgå både genom Unitys onlinedokumentation och från andra aktörer som exempelvis videoströmmningshemsidan Youtube. Många framgångsrika spel har utvecklats med motorn vilket stärker dess relevans på marknaden. Exempel är Guns of Icarus Online(Muse Games), Rust(Facepunch Studios), Wasteland 2(inXile Entertainment), Kerbal Space Pro-

gram(Squad), Never Alone(Upper One Games) [17] och AAA utgivaren Blizzard Entertainments kortspel Hearthstone [18].

1.1 Syfte

Mot given bakgrund ämnar projektet besvara följande frågeställning: Hur kan spelmekaniken i ett actionorienterat 2D-plattformsspel implementeras i utvecklinsmiljön Unity Personal? Detta görs genom att designa och implementera en prototyp av ett spel i genren, vilket är projektets slutprodukt. Utöver huvudsyftet finns också två delsyften.

Ett delsyfte är att göra en studie i speldesign. Speldesignen är det mest centrala elementet i ett spel och är därför essentiell för kvaliteten på slutprodukten. Speldesign och spelmekanik är dessutom inte helt skiljbara eftersom spelets design ligger som grund för den spelmekanik som implementeras.

Eftersom spelet utvecklas i Unity Personal krävs en del efterforskning i hur utvecklingsmiljön fungerar samt hur den kan användas i produktionen av spelet. Det formar också den arbetsprocess som används i arbetet. Det sista delsyftet blir därför att redogöra för och utvärdera den arbetsprocess som brukas i projektet med förhoppning att detta kan ligga som grund för framtida försök att göra ett liknande arbete.

1.2 Mål

Målet med projektet är att i kod implementera spelmekanik i ett actionorienterat 2D-plattformsspel. Spelet ärver många av sina spelmekaniska egenskaper från projektets primära inspirationskälla, Super Meat Boy [19] och liksom Super Meat Boy skall spelet vara enkelt att spela men svårt att avklara, dvs. spelet skall ha låg komplexitet men hög svårighetsgrad. Hur de olika spelmekaniska inslagen är designade och implementerade är avgörande för kvaliteten på slutprodukten. Detta övergripande mål har vidare delats upp i två delmål. Vilket pressenteras nedan.

1.2.1 Minimum Viable Game: Ett första implementationsmål

Inom produktutveckling finns ett koncept som på engelska heter Minimum Viable Product (MVP). Begreppet definierar den tidigaste versionen av en produkt som uppfyller ett syfte för konsumenter. En MVP utvecklas med så få resurser som möjligt och används för att utvärdera existensen av en tillräckligt omfattande målgrupp samt att appellera investerare [20, s.23] . Detta tankesätt går att applicera på spelutveckling genom det som kallas Minimum Viable Game (MVG) [21, s.53] . En MVG ämnar dock inte släppas som produkt till konsumenter utan används för utvärdering av grundkonceptet samt kontrollera att dess funktioner är tillfredsställande. Det första målet är därför att ta fram en MVG för den slutprodukt projektet ska resultera i. Följande spelmekanik omfattas i delmålet:

Spelaren skall genom spelkaraktären kunna:

- Gå
- Springa
- Hoppa
- Utföra hopp mot väggar

Spelvärlden skall innefatta:

- Mark som spelaren kan stå på
- Väggar som spelaren kan hoppa ifrån
- Fiender och hinder
- En målgång för att ta sig vidare till nästa bana

Ytterligare ett krav är att spelvärlden konstrueras så att spelaren får möjlighet att testa på samtliga av spelkaraktärens förmågor i samband med olika hinder. Vid förlust skall spelaren omgående få ett nytt försök på banan.

1.2.2 Expanderande implementationsmål

Det andra delmålet är att implementera ytterligare funktionalitet till den grundläggande spelmekaniken i projektets MVG. Tilläggen är både nya spelmekanikiska inslag samt avancerade versioner av de tidigare MVG-målen och omfattar följande mål:

Spelaren skall genom spelkaraktären kunna:

- Utföra ett extra hopp i luften
- Kontrollera sin rörelse horisontellt även när karaktären är i luften
- Utföra en snabbdykning, rakt nedåt

Spelvärlden skall innefatta:

- Olika subvärldar med ökande svårighetsgrad
- En hub-bana som leder till de olika subvärldarna
- Plattformar i rörelse som spelaren kan stå på
- Plattformar som förstörs efter spelkaraktären rört dem
- Samlingsbara objekt och hemligheter
- Ljudeffekter som betonar spelmekaniska moment, samt musik till spelets banor

- Följsam, samt avgränsad kamera
- Fiender som interagerar med spelvärlden och spelaren
- En vattenvärld där spelkaraktären kan simma
- Objekt som ändrar spelkaraktärens gravitation

Det finns även ett mål att implementera en sparfunktion med tillkommande förmåga att starta ett nytt spel samt ladda ett sparat spel. Även en meny där inställningar kan göras skall implementeras samt en utmärkelselista där spelaren ska kunna se sina framgångar.

1.3 Avgränsningar

För att det skall vara möjligt att producera ett spel samt författa all projektrelaterad dokumentation inom den givna tidsramen har vissa avgränsningar gjorts. Grundläggande spelmekanik som avser spelarens rörelser avgränsades till en imitation av den i spelet Super Meat Boy med vissa expansioner som beskrivs under avsnitt 1.2. Visuell och auditiv återkoppling är viktiga medel för att hålla spelaren informerad om händelser i spelvärlden samt ge spelaren subtila signaler som gör handlingar mer instinktiva [22, s111, s364]. På grund av tidsbegränsning läggs dock minimal vikt för implementationen av dessa element. Grafik och ljud implementeras därför i största utsträckning endast då det krävs av den planerade spelmekaniken. Av samma anledning implementeras ingen eller minimal handling i speldesignen. Således undviks omfattande design av icke-spelarkaraktärer, handlingsdrivna skriptade event och filmiska scener.

Mängden spelkonsumenter är stor och ett spel kan omöjligen tillfredsställa alla. Det är därför viktigt att rikta spelet till en specifik målgrupp. Målgruppen för spelet är personer som söker engagerande spelmekanik och hög svårighetsgrad. Super Meat Boy riktades till den här målgruppen och bevisade både dess existens, samt att ett spel riktat till den kan bli en massiv succé [12][13].

1.4 Metod

Projektet inleddes med litterära studier inom speldesign. Med dessa studier som underlag genomfördes en brainstorming. Utifrån brainstormingen, samt tidigare nämnda avgränsningar (avsnitt 1.3) för projektet, diskuterades olika spelelement för fastställa en slutgiltig design.

Då speldesignen började ta form påbörjades även en kort inlärningsperiod för att bekanta sig med huvudverktyget för projektet, Unity Personal. Träningsmaterial fanns att tillgå på Unitys officiella hemsida, vilket gav en övergripande förståelse för programmets funktioner och komponenter. Under inlärningsperioden utvärderades möjligheten att implementera projektets olika mål med den givna tidsrestriktionen. De inslag som ansågs för krävande ändrades eller dömdes ut.

Implementationen av spelet har skett parallellt med inlärningen och har delvis skett genom parprogrammering. Kortfattat innebär parprogrammering att två programmerare arbetar tillsammans på samma skärm, där en har möjlighet att göra inmatningar medan den andra aktivt

granskar och kommer med idéer [23]. Målet var att öka kvaliteten på koden, underlätta problemlösningen samt undvika att för mycket tid spenderades på ett problem [24]. En annan fördel med parprogrammering är tillförlitlighet. Om någon skulle bortfallit under arbetet på grund av exempelvis sjukdom, så fanns det ändå alltid någon som var insatt i varje kodstycke.

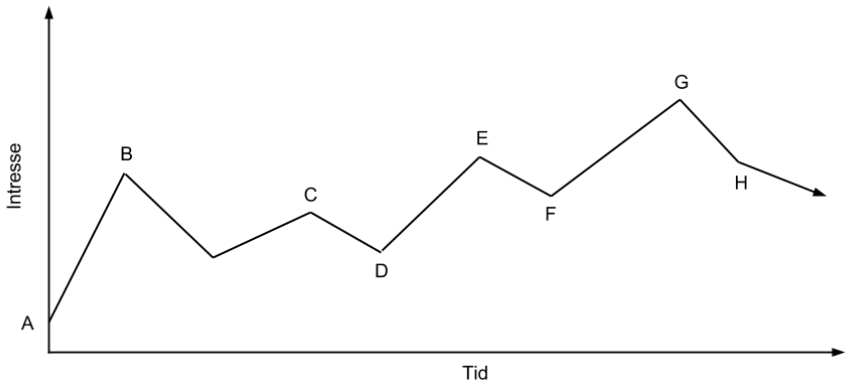
Tidigt under implementationen byggdes en gemensam programgrund som det fortsatta arbetet utgick ifrån. I programgrunden ingick spelkaraktärens samtliga förmågor och spelvärldens grundläggande funktionalitet. Den här arbetsmetoden gjorde projektet skalbart och tidseffektivt i en mindre grupp. Produkten anpassades och begränsades successivt efter projektgruppens förmågor. En viktig aspekt av arbetsmetoden var att olika spelmekaniker kunde implementeras parallellt i olika subvärldar utan att skapa konflikter med varandra.

2 Speldesign för 2D-actionplattformsspel

En av de viktigaste delarna vid konstruktion av ett spel är dess design. För att kunna göra välgrundade val i detta anseende utförs en mindre studie i speldesign. Det blir också ett naturligt steg på vägen till att uppfylla projektets mål eftersom välfungerande spelmekanik föds ur genomtänkt speldesign. I detta avsnitt presenteras ett par viktiga koncept gällande speldesign, designen för det producerade spelet, dess stöd utifrån dessa koncept samt gruppens egna idéer.

2.1 Intressekurvan

Intressekurvan är ett etablerat koncept inom speldesign [25]. Kurvan beskriver nivån av intresse spelaren har för spelet över tid. När en sådan kurva ritas bör utseendet bestå av en initialt hög topp då spelet ämnar att fånga spelarens intresse, som på engelska kallas the hook"(kroken), följt av en nergång då tempot sänks och spelaren tillåts bekanta sig med spelet. Det följs sedan av en serie toppar och dalar som gradvis ökar i storlek för att slutligen kulminera i finalen, vilken måste vara den högsta toppen, efter vilket spelet är över. I figur 1 visas ett exempel på en optimal sådan kurva där B markerar spelets krok, C, D, E och F markerar toppar och dalar för olika händelser, G markerar spelets klimax och H spelets slut.



Figur 1: Exempel på hur en optimal intressekurva kan se ut

Avvikelser från kurvan kan få konsekvenser, exempelvis om den initiala kroken är för kraftig kan resten av spelet kännas underlevererande. Är den emellertid för svag fångas inte spelarens intresse nog för att fortsätta spela. På ett liknande sätt kan resten av spelet kännas underlevererande om någon av de tidigare topparna skulle vara större än de nästkommande. Är topparna för utspridda kan upplevelsen kännas innehållsfattig och uppkommer de för tätt blir spelaren desensibiliserad. Slutligen, om intressetoppen för spelets sista akt inte är större än de andra kommer slutet kännas otillfredsställande. Detta koncept går att applicera på mer än bara spelets övergripande design; varje subvärld, bana och utmaning kan också gynnas av att följa den här modellen. I spelet som produceras är banlängden emellertid så kort att varje bana får ses som själva utmaningen. Således sammanfogas de två senare kategorierna [26, s.247].

2.2 Val av genre

Vid utveckling av en produkt eller tjänst är det viktigt att veta vilken målgrupp den riktar sig till. Ingen produkt kan vara perfekt för alla. Faktorer som personliga preferenser, ålder, socio-ekonomiska förutsättningar och i somliga fall särskilda behov (exempelvis nedsatt syn) påverkar vad som eftersöks i en produkt eller tjänst. Detsamma gäller för digitala spel, där ett sätt att delvis definiera målgruppen är genom valet av genre. De flesta spelkonsumenter föredrar vissa genrer över andra då det blir lättare för dem att hitta produkter som sammanfaller med deras preferenser. Samtidigt vill spelutvecklare hitta konsumenter för deras produkter så kategoriseringen generarar positivt utfall för båda parter.

Olika spelgenrer kräver olika mängder resurser och tid att utveckla. På grund av tidsramen för projektet var det viktigt att välja någon av de enklare genrer i detta avseende. I webbserien Extra Credits, frontad av den professionella spelutvecklaren James Portnow, benämns 2D-Plattformers som ett exempel på en genre som, i relation till andra, är enkel att skapa en MVG för [27]. Därför valdes att implementera ett actionorienterat 2D-plattformsspel.

2.3 Svårighetsgrad och komplexitet

Filosofin för den övergripande designen av spelet är att det skall vara svårt och actioninriktat men inte komplext. Genom en välformad inledning i spelet som beskrivs i avsnitt 2.8 skall kontrollerna och spelkaraktärens förmågor lätt gå att lära sig och svårigheten skall istället återfinnas i precisionsbaserad ban- och världsdesign. Detta är en filosofi som återfinns i många framgångsrika spel i genren, såväl nya som gamla, exempelvis Super Mario Bros (Nintendo), Volgar the Viking (Crazy Viking Studios), 1001 Spikes (Nicalis) och Super Meat Boy.

Detta tillvägagångssätt gör det enkelt att lära sig kontrollerna och snabbt göra framsteg i spelet, vilket är i enlighet med intressekurvans initiala topp. Är den inledande pedagogiska sektionen för lång kan spelets inledning kännas trög. Gränsen för vid vilken tidpunkt detta händer tror projketgruppen kan variera från genre till genre och vara särskilt kort för 2D-actionplattformers eftersom spelgenren inherent inte är särskilt komplex jämfört med exempelvis rollspel eller realtidsstrategispel [27].

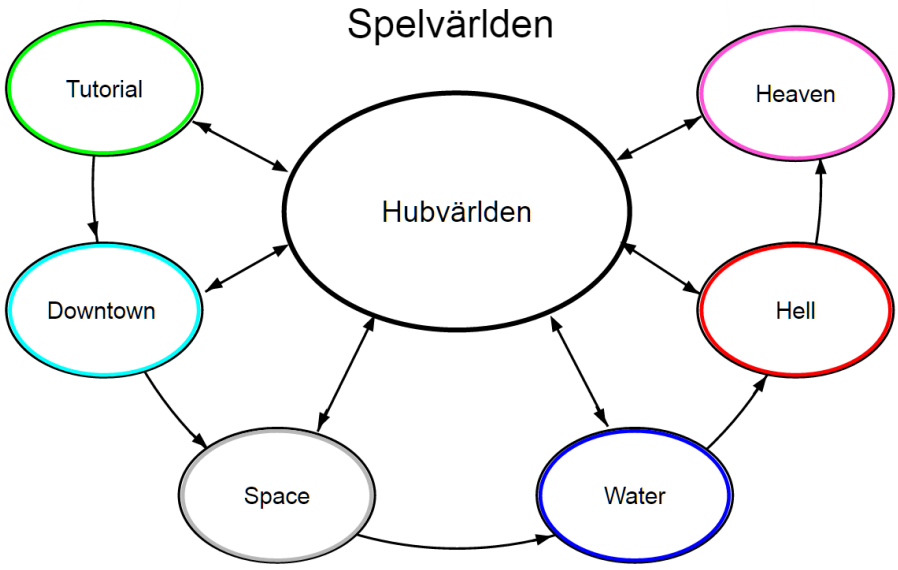
Jeff Schell beskriver i sin bok The Art Of Video Game Design att nivån av utmaning som ett spel presenterar för spelaren är en god indikation för hur intressant spelet är [26, s. 251]. Detta faktum utnyttjas i designen av spelet och dess svårighetsgrad. De mellanliggande topparna i intressekurvan för den övergripande speldesignen representeras av olika subvärldar. En subvärld motsvarar en ordnad följd av banor med gemensamt tema. Banorna för var subvärld blir gradvis svårare, men svårighetsgraden för den sista banan i var subvärld skall vara högre än första banan i nästkommande subvärld. På så vis sänks initialt tempot och ger spelaren tid att absorbera estetiken av den nya subvärlden samt lära sig de nya spelmekaniker den presenterar.

Utöver svårighetsgradens fortlöpande måste den ges en initial nivå. I detta spel har det gjorts ett aktivt val att sätta den nivån högt. Det sammanfaller med projektgruppens egna preferenser och är den typ av speldesign som främsts vill undersökas. Idén är att en svårare utmaning ger större känsla av bedrift när den avklaras. Detta koncept utnyttjas till exempel i de populära spelen från Fromsoftware: Demon Souls, Dark Souls och det nyligen utgivna

Bloodborne. Detta tillvägagångssätt presenterar dock sina egna problem. Om svårigheten överstiger rimliga nivåer, eller hanteras felaktigt, finns risken att spelaren blir för frustrerad och slutar spela. Projektgruppens tanke kring frustrationen är att den skall uppstå på grund av bristande skicklighet snarare än en känsla av att spelet är orättvist designat, omöjligt att klara eller på grund av dåliga kontroller.

2.4 Världar och banor

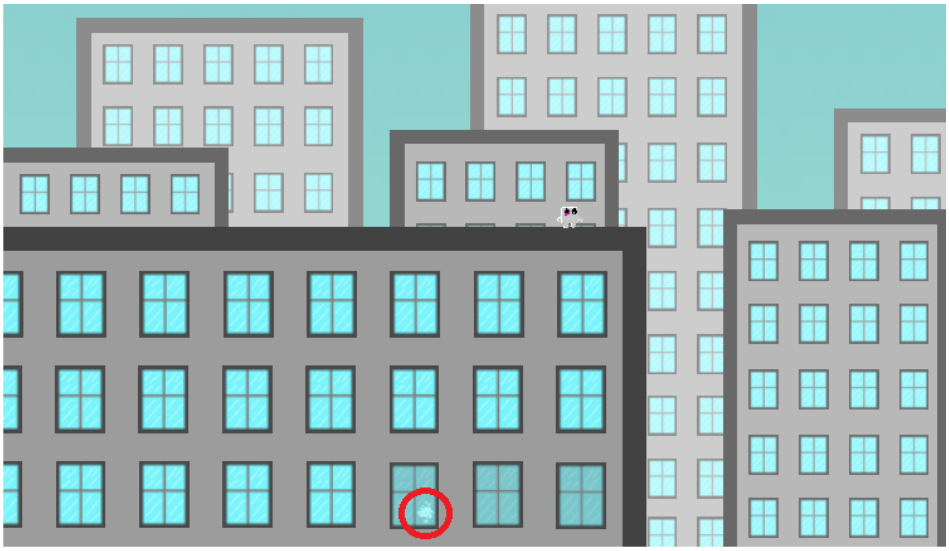
Spelvärlden är uppdelad i sex olika subvärldar: Tutorial, Downtown, Space, Waterworld, Hell och Heaven samt en hub-bana som knyter samman dessa, vilket illustreras i figur 2. När man klarat en subvärld låses en ny upp enligt angiven ordning. Det finns flera anledningar till att spelet delats upp på det här sättet. Varje subvärlds svårighet kan sättas för att representera de mellanliggande topparna i intressekurvan, vilket behandlas i avsnitt 2.3. Samtidigt tror projektgruppen att uppdelningen ger spelaren milstolpar i spelandet och på sätt förstärker känslan av framgång. Att dela in banorna i olika subvärldar öppnar även upp för möjligheten att låta varje subvärld vara unik, dels genom ett eget designtema, men även genom de olika spelmekaniska element som förekommer på dess banor. Även detta är ett redskap som bidrar till upprätthållningen av intressekurvans form. Om samma grundläggande bandesign och estetik återfinns i hela spelet tros den bli ointressant för spelaren efter en viss tid. Den gradvisa introduktionen av spelmekaniska element tvingar dessutom spelaren att lära sig nya moment under spelets fortlöpande och bidrar således till ett varierande spel. Detta sättet att hantera subvärldar återfinns i en mängd framgångsrika spel som Super Mario 64(Nintendo), Spyro the Dragon-serien(Insomniac) och Crash Bandicoot-serien(Naughty Dog).



Figur 2: En visuell avbildning av spelvärlden och hur den är ihopkopplad.

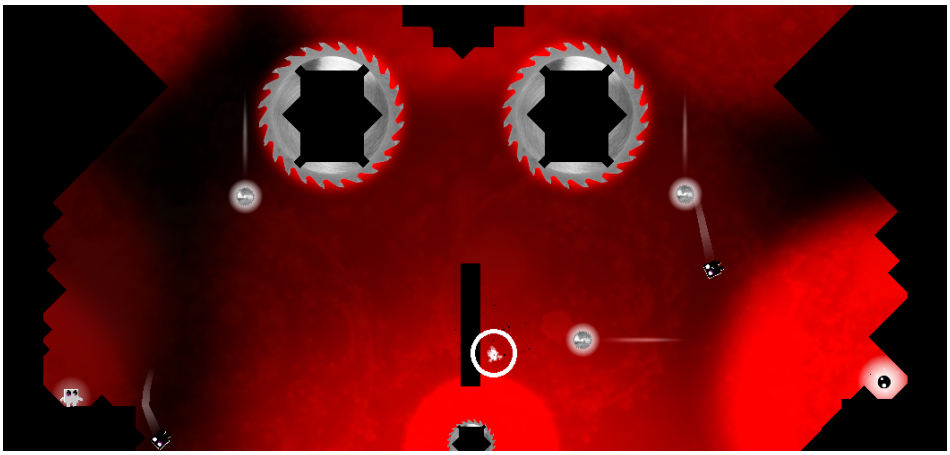
Grunden för en bana är mark, väggar, startposition, målgång samt en serie hinder. Vid designen av banor har det tagits ett par medvetna beslut. För att målgången skall vara lätt att urskilja är den färglagd svart och vit samt animerad med en partikeleffekt som cirkulerar runt den. Tiden det skall ta att klara av en bana för en erfaren spelaren bör inte överstiga en minut. På så sätt begränsas antalet hinder spelaren måste övervinna för att återvända till

samma hinder för ett nytt försök. Banorna skall vara kompakt designade, det vill säga hindren skall komma tätt inpå varandra så att spelaren slipper gå längre sträckor för att nå dem. Det bidrar till en snabbare och mer actioninriktad spelkänsla. Det finns lågt underhållningsvärde i händelsefattiga transportsträckor [22, s.115], vilka bidrar till att göra mellanrummet mellan topparna på intressekurvan för breda.



Figur 3: Gömt samlingsobjekt, markerad med röd cirkel

Vidare skall varje bana innehålla ett samlingsbart objekt [22, s.79]. Samlingsbara objekt är något som används av många plattformsspel för att uppmana spelaren till att utforska spelvärlden eller utmana sig själv. Genom att gömma den, som i figur 3, tvingas spelaren vara extra uppmärksam och genom att placera den strategiskt, som i figur 4, kan man låta spelaren själv öka svårighetsgraden hos en bana genom att försöka ta den. Andra spel som nyttjar det här tricket är Donkey Kong Country-serien(Rare), Crash Bandicoot-serien(Naughty Dog) och Rayman-serien(Ubisoft).

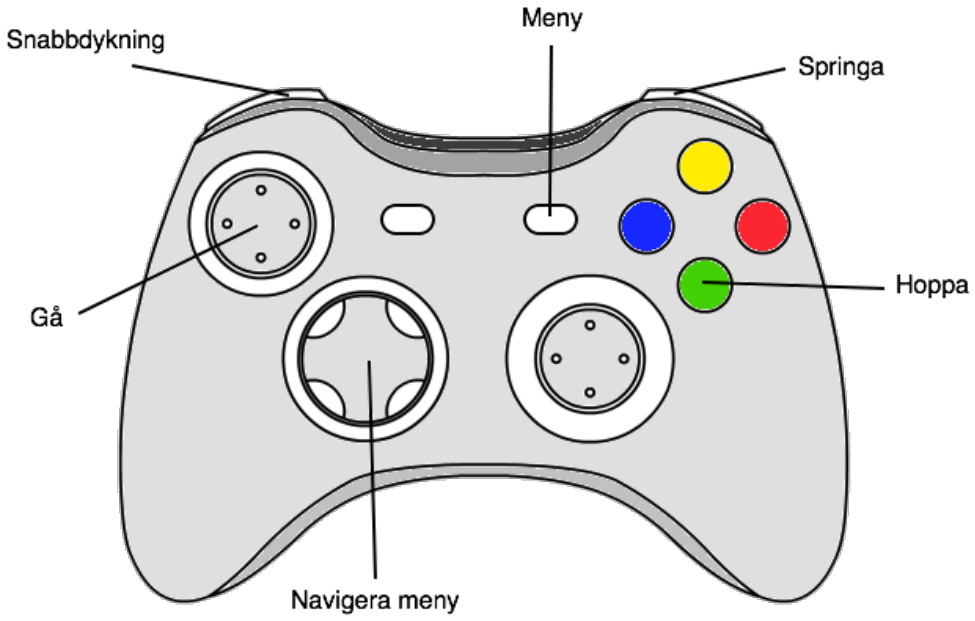


Figur 4: Ett samlingsobjekt som är svårt att ta, markerad med vit cirkel

För att dessa objekt skall locka spelaren att plocka dem bör det resultera i en belöning [26, s.33]. Detta kan göras genom en utmärkelselista, från engelskans achievements, där en utmärkelse tilldelas spelaren när en eller flera uppgifter avklaras. Exempel på en sådan uppgift är att samla alla samlingsobjekt från en subvärld. Detta förändrar inte spelet men ger spelaren ytterligare ett mål att uppfylla.

2.5 Kontroll

Det finns tre huvudsakliga faktorer som bör övervägas när kontrollen för spelkaraktären tilldelas. Primärt bör de existerande konventioner för den givna genren studeras för olika funktioner. Personer som spelat många andra spel kommer ha förväntningar på hur spelkaraktären styrs och om dessa inte uppfylls känns spelet icke-intuitivt. För det producerade spelet har främst kontrollerna för Super Mario-serien och Rayman-serien studerats. Nästa faktor som bör övervägas är hur väl tilldelningen stämmer med vad som händer i spelet. Trycker spelaren på neråtknappen förväntas en rörelse i samma riktning. Slutligen bör ergonomin beaktas, för att göra det möjligt att spela spelet en längre tid utan smärtor och obehag. Lyckligtvis täcks båda dessa faktorer oftast upp av konventionerna såvida man inte har väldigt annorlunda förmågor eller spelstil i sin design [22, s.164]. Spelet stödjer även användningen av en Xbox-kontroller. På Xbox-kontrollen används höger pekfingerknapp se fig 5, från engelskans "right trigger", för att springa vilken är den enda tilldelning som avviker från de konventioner som finns. Spelet kräver mycket hoppande och springande på samma gång och för att kunna springa även när tummen är upptagen med hoppknappen valdes den här tilldelningen.



Figur 5: En Xbox-kontroll där spelfunktionerna är tilldelade olika knappar.

2.6 Kamera

Kameran bestämmer vilka delar av banan spelaren kan se vid varje given tidpunkt. Kameran bör begränsas till de områden och riktningar som har ett värde för spelaren. I annat fall riskeras att spelaren vilseleds och på så sätt förstörs spelets flöde [22, s.132]. I spelet hanteras kameran på två olika sätt beroende på hur banan är uppbyggd. Det mest grundläggande fallet är en statisk kamera som används när banan är av tillräckligt små dimensioner för att få plats på en datorskärm. I annat fall används en skrollande kamera med centrum på spelaren, såvida det inte hindras av banans kanter. Den skrollande kameran kan låsas i x- eller y-led för att undvika ovannnämnda vilseledning av spelaren.

I subvärlden Downtown används en speciell metod som på engelska kallas "parallax scrolling", parallaxerande skrollning. Metoden utnyttjar det som kallas rörelseparallax, uppfattningen att objekt som befinner sig på längre avstånd från åskådaren rör sig långsammare än de som befinner sig nära. Genom att förflytta objekt med olika hastighet i förhållande till kameran ges illusionen av att somliga objekt existerar i banans bakgrund. Ju längre bort objekt skall framstå, desto lägre hastighet skall de ha [28]. Genom att kombinera detta med en grafisk profil där objekt är mindre och blekare desto längre bort från spelkaraktären de är kan man skapa ett artificiellt djup i banorna vilket gör dem mer levande. (På grund av fukt och partiklar i luften upplevs saker blekare ju längre bort de finns [29]).

2.7 Spelmekanik

För att uppfylla visionen om ett spel med låg komplexitet har karaktären ett begränsat antal förmågor som inte förändras under spelets gång. Detta försäkrar att spelaren kan bemästra kontrollen av karaktären på ett djupare plan jämfört med om denne måste lära om hur karaktären kan förflyttas under spelets fortlöpande. Istället levereras spelets diversitet genom olikt designade banor med skilda spelmekaniker som kombineras på utvalda sätt för att skapa miljöer och situationer som tvingar spelaren att reagera och anpassa sin spelstil.

2.7.1 Spelkaraktärens rörelse och förmågor

Designen av spelarförmågor för genren 2D-actionplattformsspel har vid en snabb blick på olika titlar många gemensamt återkommande idéer. De flesta idéer kretsar kring rörelse eller att skjuta projektiler, där det sistnämnda inte är aktuellt för projektet. Hur spelkaraktärens förmågor bör fungera måste vägas mot hur resten av spelet designats. All spelmekanik utgår från spelkaraktärens förmåga att navigera i spelet. Därmed påverkar designvalen av förmågor hur spelvärlden samt eventuella hinder bör se ut [22, s.92].

Spelkaraktärens förmågor inspirerades till stor del av de som återfinns i Super Meat Boy, med tilläggen dubbelhopp och snabbdykning. Motivet bakom spelkaraktärens förmågor är att öka spelarens möjlighet till skicklig och snabb navigering bland hinder, för att stärka den actionorienterade känslan. För att ge spelaren auditiv återkoppling har hoppförmåga försetts med en ljudeffekt som spelas upp när den utförs.

På marken har spelkaraktären en bromssträcka som är olika lång beroende på hastighet. Projektgruppen anser att det ger spelkaraktären en mer naturlig känsla eftersom det hindrar att spelkaraktären tvärt kan stanna eller byta håll. Ett naturligt rörelsemönster är dock inte

alltid det som eftersträvas i ett spel. För att tillåta mer precisionsbaserade utmaningar kan spelarkaraktären i luften bromsa sin hastighet eller helt byta riktning. Spelaren kan också välja hur högt spelkaraktären skall hoppa genom att släppa hoppknappen när den önskade höjden är nådd [22, s.93-103]. Dock finns dock en maxhöjd som ett hopp kan nå.

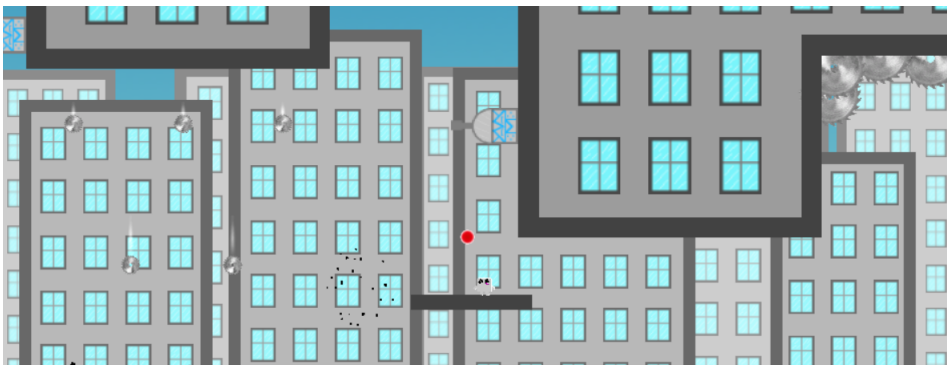
Dubbelhopp ger spelaren mer kontroll i luften. Det kan användas för att hoppa högre, längre eller för att sänka hastigheten i y-led vid fall. Snabbdykning är ännu ett sätt för spelaren att påverka sin position i luften och kan utföras medan spelkaraktären inte rör vid marken. Vid snabbdykning låses karaktärens position i x-led och den får en snabb hastighet nedåt. För att förstärka känslan av en snabb rörelse ackompanjeras detta med en visuell återkoppling i form av en partikeleffekt samt att kameran skakar. Slutligen är förmågan att hoppa från väggar en nödvändighet för att låta spelaren hastigt förflytta sig vertikalt utan att förlora tempot i spelet. Förmågan tillåter byggandet av vertikala banor och utmaningar vilket gör designen mer diversiv.

Avslutningsvis har spelkaraktären förmågan att simma i vatten-subvärlden vilket realiseras genom att tillåta spelkaraktären att hoppa ett oändligt antal gånger med en lägre gravitation.

2.7.2 Spelvärlden

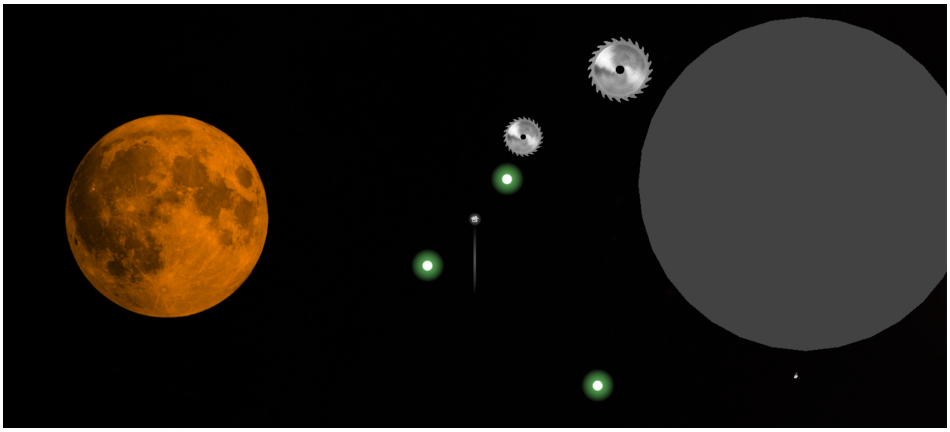
De mest grundläggande objekten som utgör den interaktiva basen i en bana är mark att gå på, väggar som spelkaraktären kan hoppa från samt tak. Banornas utsträckning är begränsad och rör spelkaraktären sig utanför banan, exempelvis genom att ramla ner för ett hål, dör den och banan laddas om. En marginal för hur långt utanför karaktären måste vara innan det sker har skapats för att ge spelkaraktären möjlighet att ta sig tillbaka in på banan.

Utöver de grundläggande byggblocken finns plattformar i rörelse samt mark och väggar som går sönder en kort stund efter spelaren vidrört dem. Plattformar i rörelse öppnar upp en mängd intressanta designmöjligheter för utmaningar på banor. Dels ger de upphov till mer precisionskrävande hopp eftersom spelaren måste ta både spelkaraktärens och plattformens hastighet i beaktning för att effektivt kunna landa på den. En bana kan även designas så att plattformen krävs för att ta sig i mål, se figur 6. Mark och väggar som går sönder är ett verktyg som används dels för att tvinga spelaren att agera kvickt och dels sätta upp situationer där spelaren bara har en möjlighet att klara en utmaning.



Figur 6: Plattformen är betydande för att klara banan

Subvärlden Space, se figur 7, är centrerad kring konceptet att byta riktning på spelkaraktärens gravitation. Via strategiskt utplacerade objekt tvingas spelaren ändra gravitationen för att manövrera runt olika hinder i banan. På dessa banor har målet en egen gravitation för att underlätta målgången och därmed undvika det frustrerande moment där spelaren tar sig förbi alla hinder men missar målet med ett kort avstånd.



Figur 7: I subvärden Space används de gröna cirklarna för att byta riktning på spelkaraktärens gravitation.

Spelvärlden i det producerade spelet innehåller en mängd objekt som kan döda spelkaraktären. Stationära sågblad är ett exempel som återfinns på majoriteten av alla banor på grund av dess breda användningsområden. Utöver att användas som ett strikt hinder placeras sågblad ut för att intuitivt leda spelaren genom banan genom att strategiskt placera ut sågblad för att visa spelaren hur man lättast tar sig igenom en sektion. Det förekommer även sågblad som förflyttar sig mellan två punkter och är mer intressanta gällande produktionen av synkroniserade utmaningar. Genom noggrann utplacering och förståelse för spelkaraktärens rörelse skapas engagerande utmaningar där tempot i spelet bibehålls samtidigt som det kräver precisionsstyrning från spelaren. Detsamma gäller för de sågbladsgenererare som skjuter sågblad.

I spelet finns, förutom sågbladsgenererare, andra fiendetyper som skjuter objekt. Spelaren möter stationära fiender som siktar och skjuter en projektil mot spelkaraktärens position om denne är inom fiendens skottradie. Detta ger möjlighet att skapa banor och situationer som tvingar spelkaraktären att ständigt förflytta sig. Det förekommer även en fiendetyp som rör sig i konstant hastighet genom banan och skjuter projektiler åt ett, eller flera, förutbestämda håll. Det gör att spelaren måste vara uppmärksam, inte bara på det framför spelkaraktären, utan även det ovanför. Slutligen finns sågblad i subvärlden space som skjuts mot specifika planeter i omloppsbana vilket ger en större känsla av att vara i rymden. Spelaren tvingas därmed att inte vara stationär en längre tid.

Slutligen finns flertalet fiendetyper som rör sig antingen efter ett förbestämt mönster med olika utseende eller jagar spelaren. En av de jagande fienderna ökar i storlek och hastighet under tiden den jagar spelaren.En fiende som rör sig längs marken tills den stöter på en vägg i vilket fall den vänder och går åt motsatt håll förekommer också. Alla dessa fiendetyper skapar situationer där spelaren måste ifrågasätta om området spelkaraktären står på faktiskt är säkert och att göra detta på olika sätt för att skapa en varierande upplevelse.

2.7.3 Meny och Pausfunktion

Huvudmenyn är det första spelaren exponeras för då spelet startas. Det är därför viktigt att huvudmenyn motsvarar de förväntningar spelaren kan tänkas ha [22, s.187]. Dess funktion är att leda spelaren till bakomliggande funktionaliteten i spelet, såsom att ladda en sparfil eller ändra en inställning samtidigt som den smälter in i spelets övriga estetik. En visuellt tilltalande och tydlig huvudmeny, se figur 8, med bra bakgrundsmusik är därmed en viktig del av ett spel.

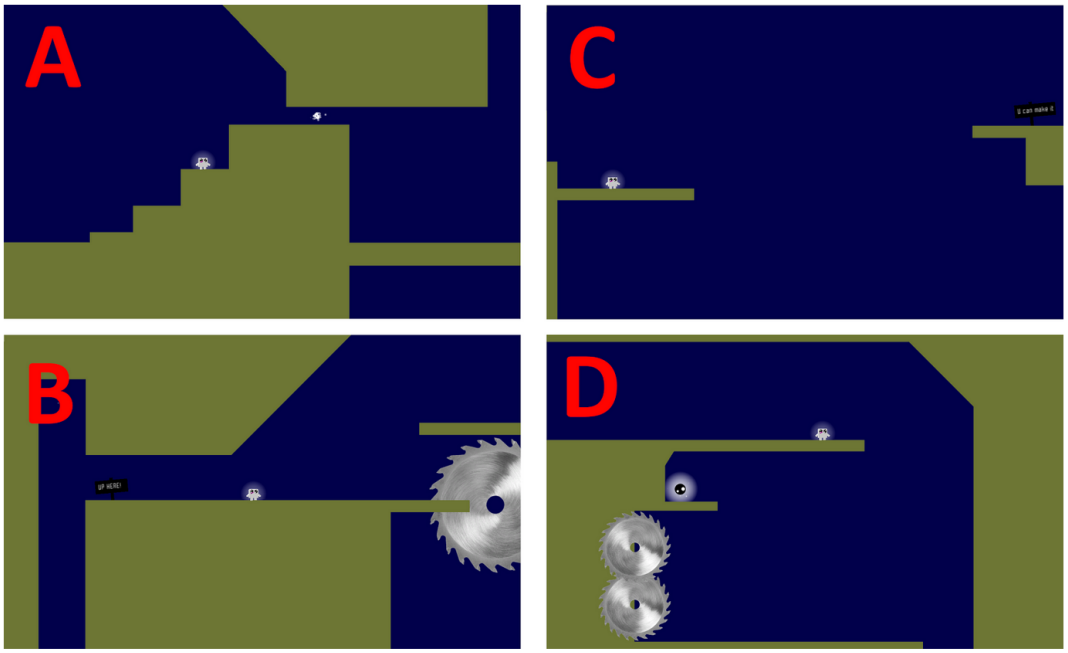
Figur 8: En visuellt tilltalande och tydlig huvudmeny

Utöver huvudmeny är en vital del av ett spel en paus-funktion, för att kunna pausa spelet. Det är viktigt att spelaren inte känner att den missar något då spelet är pausat [22, s.190]. Det skall heller inte ge någon spelfördel att pausa, som att exempelvis i lugn och ro kunna iaktta banan. Pausmenyn har flera gemensamma funktioner med huvudmenyn. Undantaget är möjlighet att förflytta spelkaraktären till hub-banan, ladda en avklarad bana eller öppna huvudmenyn. Pausmenyn, likt huvudmenyn, bör inte avvika från spelets estetik.

2.8 Inlärningsbanor

Syftet med en inledande, pedagogisk sektion är att lära spelaren den grundläggande spelmekaniken. En bra sådan skall kunna introducera ett avancerat och omfattande spel utan att tappa spelarens intresse. Detta projekt skall emellertid varken leverera ett särskilt avancerat eller omfattande spel men det är fortfarande viktigt att ha partier i spelet som introducerar spelkaraktärens rörelseförmågor.

För att lära spelaren hur spelet fungerar kan textrutor användas, men de drar ner speltempot och kan försämra spelarens fokus. Det finns dessutom en risk att spelaren inte läser dem. Därför har en interaktiv sektion skapats som låter spelaren utföra olika pedagogiska inlärningsmoment. Detta garanterar att spelaren inte kan fortsätta utan att ha förstått alla moment [26, s.146]. Dessutom har lärandet delats in i mindre, mer överkomliga utmaningar för att förhindrar att spelaren utsätts för allt för många intryck på en gång.



Figur 9: Interaktiva moment som lär spelaren olika rörelseförmågor

I figur 9 visas några exempel på inlärningsmoment från spelet. Trappan i bild A ämnar lära spelaren hoppa olika högt, där det sista trappsteget kräver ett högre hopp än de tidigare. Det följande samlingbara objektet tvingas genom bandesignen plockas upp av spelaren vilket påvisar att det både är ofarligt och är något spelaren bör göra. I bild B finns två inlärningsmoment. Spelkaraktären börjar banan i hålet, vilket kräver att spelaren lär sig hoppa från väggar. Sågbladet på samma bild tvingar spelaren att utföra dubbelhopp för att fortsätta. För att klara av momentet i bild C måste spelkaraktären både springa och dubbelhoppa för att nå andra sidan. I den sista bilden, bild D, måste spelaren lista ut ett sätt för att nå målet som befinner sig i rummet ovanför sågbladen. Enda sättet att nå den platsen är att springa och hoppa från väggen på motsatt sida.

3 Utveckling i Unity för 2D-spel

Unity är det utvecklingsverktyg som användes för att implementera speldesignen. Detta avsnitt kommer introducera några av Unitys mest grundläggande implementationsmedel för utveckling av 2D-spel.

3.1 Spelobjekt och Komponenter

En fundamentalt del inom Unity är spelobjekt. Allt i spelet är i grunden spelobjekt. Spelobjekt är en behållare av specifika egenskaper som definierar dess beteende och funktionalitet. Egenskaperna får spelobjekt genom komponenter. På så sätt utgör spelobjekt och komponent de grundläggande byggstenarna för all utveckling i Unity.

Egenskaper som finns hos alla spelobjekt är att de har en position, rotation och skala. De här egenskaperna får spelobjektet genom komponenten Transform som spelobjektet tilldelas när det skapas. Transform är på så sätt speciell eftersom det är den enda komponenten som måste finnas hos alla spelobjekt. I tabell 1 presenteras några vanliga komponenter som används för utveckling av 2D-spel.

Det finns några restriktioner gällande olika komponenter hos ett spelobjekt. Vissa komponenter får det endast existera en instans av. Det får exempelvis inte finnas två Transformkomponenter på samma spelobjekt. Utöver det finns komponenter som inte är kompatibla med varandra. Exempelvis får inte en Sprite Renderer och Mesh Renderer användas tillsammans.

Komponent	Förklaring
Transform	Används för att förvara och modifiera spelobjektets position, rotation och skala.
Collider2D	Används för att definiera ett spelobjekts kollisionsområde i två dimensioner. Finns i olika typer, bl.a. rektangel, cirkel eller rak linje. Komponenten spelar en viktigt roll i hantering av fysik och händelser.
Rigidbody2D	Används för att spelobjektet skall kunna bli påverkad av fysikmotorn och tillför storheter såsom massa och hastighet.
Behaviour Component (Skript)	Skript skrivna i C# eller Javascript och används för att ge spelobjektet unika beteenden. Det finns inga inbyggda skript-komponenter i Unity utan måste skrivas av utvecklaren.
Sprite Renderer	Används för att rendera en bild och skapa en visuell representation av spelobjektet.
Canvas	Används i samband med grafiska användargränssnitt och motsvarar det området där gränssnittets komponenter får förekomma.
Mesh Filter	Används för att definiera utseendet för en geometrisk figur utifrån, en så kallad mesh.
Mesh Renderer	Används för att rendera den geometriska figuren som definieras av Mesh Filter-komponenten.
AudioSource	Används för att spela upp angivna ljudfiler. Den har bl.a. funktionalitet för att spela, pausa samt göra inställningar för ljudvolym.
Particle System	Används för att rendera visuella effekter.
Button	Används för skapa en knapp som vid tryck aktiverar en given händelse.

Tabell 1: Olika komponenter och dess användningsområden för utveckling av 2D-spel i Unity.

3.2 Skript

Skript är en viktig del inom all typ av spelutveckling i Unity och kan användas för att skapa allt från visuella effekter till fysikaliska krafter vid fysiksimulering. Unity hanterar skript som vanliga komponenter, se tabell 1. I regel används skript när de inbyggda komponenterna inte räcker till för att åstadkomma det beteende som eftersträvas. Skript är antingen skrivna som en C#-klass eller Javascipt-fil, vilket innebär att utvecklaren får tillgång till allt som vanligtvis finns tillgängligt för språken.

Alla skript använder sig av MonoBehaviour som är en klass från biblioteket UnityEngine. Genom MonoBehaviour ärver skriptet en mängd användbara metoder som styrs av spelmotorn, som exempelvis för att ladda in scener eller för att uppdatera tillstånd hos spelobjekt. UnityEngine innehåller ett flertal användbara klasser utöver MonoBehaviour. Bland dessa finns

Vector2 som representerar tvådimensionella vektorer. Genom MonoBehavior får också skriptkomponenten tillgång till en stor mängd händelserelaterade metoder som är användbara vid exempelvis kollisioner mellan spelobjekt. För att dessa metoder skall fungera måste det finnas en Collider-komponent hos samtliga av de inblandade spelobjekten.

Skript-komponenter har en möjlighet att bearbeta och manipulera andra komponenter. Detta görs via fördefinierade metoder och variabler hos varje komponenttyp precis som för en C#-klass. För att hämta enskilda komponenter hos spelobjektet används GetComponent från MonoBehaviour-klassen, ett exempel visas i figur 10.

```
1  using UnityEngine;
2  using System.Collections;
3
4  public class ScriptName : MonoBehaviour {
5
6      // Use this for initialization
7      void Start () {
8
9          // Set the position of the game object to (0 , 0, 0)
10         GetComponent<Transform>().position = Vector3.zero;
11
12         // Store the component locally
13         Transform transform = GetComponent<Transform>();
14
15         // Define movement
16         Vector3 movement = new Vector3(1, 1, 0);
17
18         // Move the game object
19         transform.Translate(movement);
20
21     }
22
23 }
```

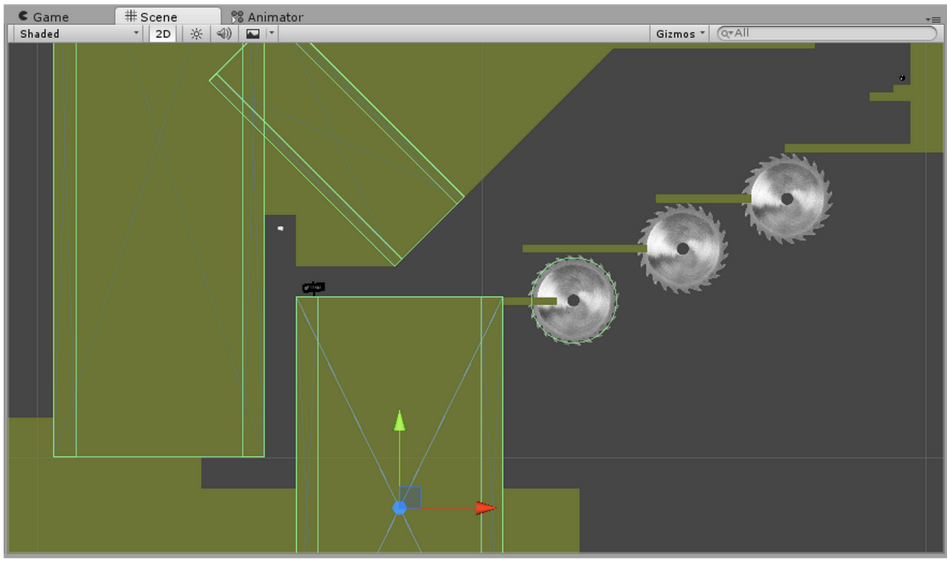
Figur 10: När spelobjektet börjar användas nollställs dess position för att sedan flyttas fram ett steg i x- och y-led. Start är en metod från MonoBehaviour-klassen.

En Transform-komponent erbjuder en mängd metoder för att modifiera spelobjektet med hjälp av dess metoder. Förutom publika metoder är det också möjligt att modifiera komponenten genom dess publika variabler. Den publika variabeln Transform.position refererar till positionen hos Transform-komponenten och styr på så sätt spelobjektets position. På samma sätt erbjuder också Rigidbody2D-komponenten användbara metoder och variabler. Exempelvis finns en metod för att addera krafter till spelobjektet och en variabel för att hålla dess hastighet.

3.3 Scener

Ett grundläggande koncept i Unity är scener och scenfönstret. En scen är en samling spelobjekt som motsvarar en sektion av spelet och identifieras med hjälp av en textsträng. Möjligheten finns att placera spelobjekt hierarkiskt i en scen, där underobjekt kallas barn, och överobjekt kallas förälder. Endast en scen i taget är aktiv och när en ny läses in förstörs alla spelobjekt i den föregående, förutsatt att inga ytterligare inställningar förekommer på spelobjekten. I träningsmaterial från Unitys officiella hemsida översätts ibland scen förenklat till bana.

Scener har dock ett bredare användningsområde än så, exempelvis kan de användas för att skapa menyer. I scenfönstret placeras alla spelobjekt som skall finnas med i scenen och får automatiskt koordindater som utgör dess position. På så sätt sker allt i spelet inom ett koordinatsystem. Figur 11 illustrerar en bana som konstrueras i scenfönstret.



Figur 11: Scenfönstret i editorn används för att placera ut saker i spelet

3.4 Kompletta spelobjekt

Då ett spelobjekt skapats i scenfönstret kan det sparas och kallas då för en Prebab. De kompletta spelobjekten kan sedan användas av utvecklaren som en mall för att skapa nya spelobjekt med liknande egenskaper. Fördelen är att ändringar kan göras direkt i den sparade mallen istället för att ändra varje enskild instans av spelobjektet. Varje enskilt spelobjekt har dessutom möjligheten att överskugga egenskaperna hos den sparade Prefaben, vilket medför ett smidigt och enkelt sätt att bygga en omfattande spelvärld med ett stort antal olika spelobjekt.

MainCamera är ett komplett spelobjekt som automatiskt finns tillgängligt i Unitys editor. Som namnet antyder är MainCamera ett spelobjekt som utgör spelets kamera. MainCamera innehar bland annat komponenterna Camera och Transform. Camera-komponentens publika Projection-variabel kan ställas in beroende på om spelvärlden skall visas i två eller tre dimensioner. För 2D-spel sätts Projection-variabeln till Orthographic, vilket innebär att kameran bortser från z-axeln.

3.5 Taggar och lager

För att ordna spelobjekten erbjuder Unity olika sätt att kategorisera och sortera dem. Detta kan göras med hjälp av taggar och lager, i form av strängar, som kan ställas in för varje enskilt spelobjekt. Taggar är markörer som gör det enkelt att identifiera spelobjekt i skript. Ett exempel är den statiska metoden FindWithTag, från biblioteket UnityEngine, som givet en tagg returnerar ett aktivt spelobjekt med samma tagg. Ett annat exempel är den metoden ComprareTag som jämför spelobjektets tagg med en given sträng. Lager används för att

gruppera spelobjekt som delar funktionalitet. Unity använder en så kallad lagerbaserad kollisionshantering, vilket innebär att det är möjligt att markera vilka lager som fysikalisk skall kunna interagera med varandra. Utöver det kan lager användas för att bestämma i vilken ordning spelobjekt ska renderas.

3.6 Fysik i två dimensioner

Genom fysikmotorn kan spelobjekt kollidera, glida nerför backar och falla fritt, utan att ytterligare funktionalitet måste implementeras. Det finns två typer av komponenter som är viktiga för tvådimensionell fysik; dessa är Rigidbody2D och Collider2D. Rigidbody2D-komponenten ger spelobjekt viktiga fysikaliska egenskaper såsom massa, hastighet samt påverkan av gravitation och andra krafter. Collider2D-komponenten tillför form och storlek. Tillsammans utgör de allt ett spelobjekt behöver för att både röra sig och delta i en kollision via fysikmotorn.

3.7 Trigger- och kollisionsevent

Unity erbjuder ett effektiva sätt att utlösa händelser via fysikmotorn. Det kan antingen ske genom en vanlig kollision mellan två spelobjekt eller med hjälp av en så kallad trigger. En trigger är en Collider som är inställd att agera trigger, närmare bestämt den booleska IsTrigger-variabeln hos komponenten är satt till true, vilket kan ordnas direkt i Unitys editor. När en Collider sätts till trigger tappar komponenten sin kapacitet att ingå i vanlig kollision. Den kan istället anropa specifika metoder när en annan Collider-komponent korsar, lämnar triggern eller befinner sig inom dess ramar. Metoderna återfinns i MonoBehaviour-klassen och kan därmed nås via skript.

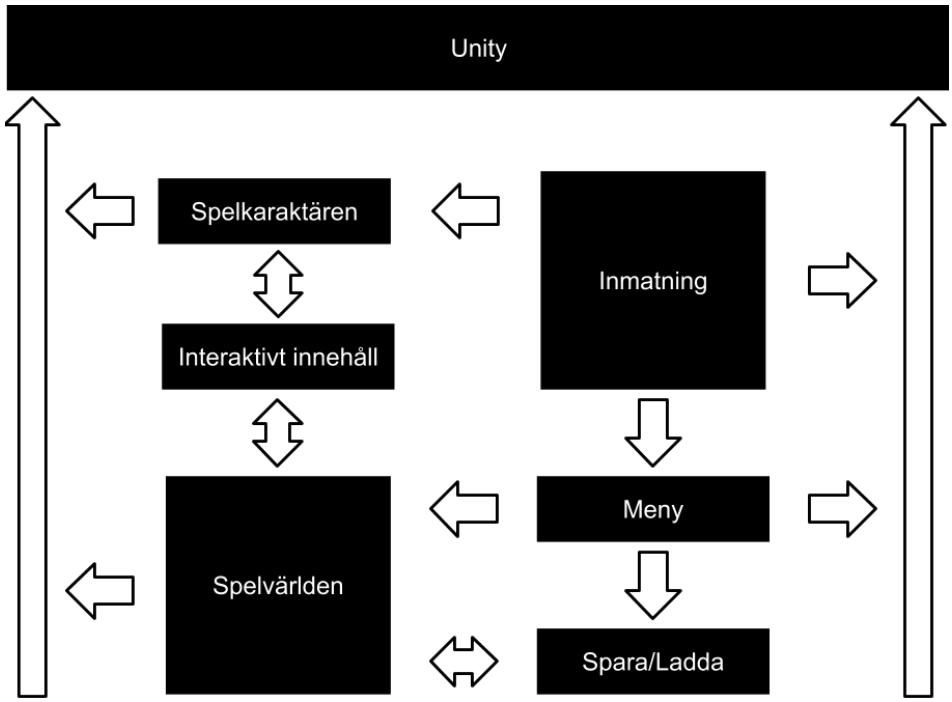
4 Spelet och dess implementation

Slutprodukten uppfyller samtliga uppsatta mål beskrivna i avsnitt 1.2, med undantag för utmärkelselistan som tyvärr fick uteslutas på grund av tidsbrist. För att spelet skall fånga spelaren och locka spelaren till att fortsätta spela har det designats utefter konceptet för intressekurvan, vilket beskrivs i avsnitt 2.1. Till vilken grad detta lyckats är svårt att mäta eftersom intressenivån en spelare har för ett spel är svårt att kvantifiera och högst subjektiv. I avsnitt 2 redogörs dock de metoder som anammades för att i största mån forma spelet efter kurvan.

Spelet är ett 2D-actionplatformsspel med fokus på en snabb och reaktiv spelstil. Det är designat för att vara intuitivt och utmanande utan att vara särskilt komplicerat. I spelvärlden finns ett antal subvärldar som innhåller ett antal banor. Spelarens mål är att med hjälp av spelkaraktärens förmågor nå slutet av varje bana i en subvärld, på så sätt låsa upp nästa subvärld och upprepa detta tills sista subvärlden avklarats. För att hindra spelarens framgångar finns hinder och utmaningar utplacerade på varje bana. Spelaren kommer på sin resa genom spelet behöva övervinna stationära hinder såsom sågblad och fallgropar; hinder i rörelse såsom flygande fiender, fiender som går på marken och sågblad som rör sig; samt fiender som skjuter projektiler som antingen siktar på spelaren eller skjuts i någon bestämd riktning. Utöver detta måste spelaren se upp för mark och väggar som går sönder en kort stund efter kontakt med spelkaraktären samt vara uppmärksam på de olika plattformar som rör sig. I subvärldarna Waterworld och Space presenteras spelaren dessutom med, för spelet, unika utmaningar. Waterworld utspelas under vattnet och ger spelkaraktären förmågan att simma och i Space kan gravitationen som påverkar karaktären vändas upp och ner när spelaren rör vid vissa objekt. För att navigera omkring alla dessa hinder och utmaningar kan spelkaraktären gå, springa, hoppa, dubbelhopa, hoppa från väggar och snabbdyka. För den riktigt skickliga spelaren finns även ett samlingsbart objekt att hitta på varje bana, antingen gömt eller placerat på utmanande ställen. Fortsättningsvis i detta avsnitt presenteras hur spelet och dess olika delar har implementerats med Unity.

4.1 Implementation

Översiktligt kan spelobjekt och komponenter som används i spelet delas in i moduler. I figur 12 återfinns en grafisk representation av systemet, där pilarna indikerar hur modulerna kan påverka varandra. Användarens kommandon matas in via inmatningsmodulen, som i sin tur använder Unitys inmatningsmetoder. Detta vidarebefordras via spelkaraktärsmodulen eller menymodulen som utför tillhörande instruktioner. Menymodulen kan i sin tur påverka spelvärlden genom att exempelvis pausa eller ändra ljudinställningar. Samma modul kan också få spara/ladda-modulen att spara eller ladda de tillståndvariabler som behövs för en spelsession. Spelkaraktärsmodulen använder sig av Unitys metoder för att påverka spelkaraktären. Spelkaraktärsmodulen kan även påverka eller påverkas av spelvärlden genom vad som kan sammanfattas som interaktivt innehåll, t.ex. när spelkaraktären interagerar med spelvärlden genom att exempelvis plocka upp ett samlingsbart objekt. I denna implementationsdel presenteras några exempel på skript av särskild betydelse eller som på ett eller på något sätt utmärker sig.



Figur 12: Koppling mellan olika moduler

4.1.1 Scenhantering

Spelet är uppbyggt av ett antal scener: en startscen, en scen för varje bana, en scen för huvudmenyn och en scen för hub-banan. När spelet startas läses startscenen in, vilket endast sker en gång per spelsession. Startscenen initierar spelobjektet Managers vars syfte är bevara centrala skript. Det är därför inställt att inte förstöras i samband med att nya scener läses in. Direkt efter att spelobjekten har initialiserats läses scenen för huvudmenyn in för att sedan låta spelarens inmatningar påverka spelsessionen.

GameController är det skript på Managers med flest kopplingar till andra skript. Det både innehåller och tilldelar de variabler som måste bevaras mellan scener. Skriptet har också starka kopplingar till ljudhanteringen, där olika spelobjekt kan avge ljud, men också till scenhanteringen för laddning av banor samt visst statiskt innehåll som exempelvis samlingsbara objekt.

Spelobjektet som utgör spelkaraktären skapas när en scen för en bana laddas in. Därefter söker GameController upp det spelobjekt som motsvarar spelkaraktärens startposition och placerar spelkaraktären där. Målgången för varje bana är representerad av ett spelobjekt i form av en portal, som dels hanterar övergången till nästkommande scen, dels sparar data om den avklarade banan. Portalen renderas med hjälp av en Sprite Renderer-komponent. I den fristående C#-klassen Records, vars enda syfte är att lagra information, finns två listor som ser till att upplåsningen av banor sker korrekt. Den ena listan innehåller namnen på alla subvärldar, sorterade efter vilken ordning dessa ska låsas upp och den andra innehåller namnet på de banor som avklarats. Då spelet startar är endast subvärlden Tutorial tillgänglig. När den är avklarad hämtas namnet på nästa subvärld från listan och låses upp. På det sättet förhindras spelkaraktären från att beträda låsta subvärldar via hub-banan innan de låsts upp.

4.1.2 Kamera och parallaxerande skrolling

Spelets kamera utgörs av spelobjektet MainCamera med Projection-variabeln satt till Orthographic. På samma sätt som Managers förstörs inte kameran i samband med att nya scener läses in. Kamerans beteende styrs av CameraController som tillsammans med Unitys inbyggda funktionalitet ser till att den ständigt centreras på spelkaraktären. Banans gränser utgörs av en rektangulär Collider2D-komponent vilken kameran ej kan röra sig utanför. CameraController har dessutom möjligheten att låsa kamerans rörelse i både x- och y-led. Om så är fallet hoppar CameraController över uträkningar i det led den är satt att vara låst. Statisk kamera åstadkoms genom att låsa rörelser i både x- och y-led.

Parallaxerande skrollning skapas med hjälp av ett skript som återfinns på spelobjektet Managers. Inmatningen till skriptet är en lista med spelobjekt som utgör den grafik som effekten skall påverka samt en skalfaktor för hur stark effekten skall vara. Spelobjekten placeras på olika avstånd i z-led från det interaktiva lagret beroende på hur långt bort de skall upplevas vara. Skriptet använder sedan spelobjektens Transform-komponent och den givna skalfaktorn för att bestämma hur snabbt de skall röra sig i förhållande till kameran. Varje gång kamerans Transform-komponent förändras beräknar skriptet hur spelobjekten i listan skall förändras.

4.1.3 Ljud och musik

Musik hanteras med skriptet MusicManager och en AudioSource-komponent som båda återfinns på spelobjektet Managers. Från Unitys editor matas varje scennamn med sin respektive ljudfil in i MusicManager för att mappas i en hashtabell. När en ny scen läses in används dess namn för att hitta ljudfilen som är kopplad till scenen och använder sedan AudioSource-komponentens inbyggda metoder för att spela upp den. Ljudeffekter laddas också in via Unitys editor, men placeras på de spelobjekt som skall associeras med dem. Ljudfilerna spelas upp via egna Audiosource-komponenter.

4.1.4 Rörliga objekt

I spelet förekommer flera spelobjekt som rör sig. De spelobjekt som följer ett bestämt mönster använder skripten PathDefinition och FollowPath. PathDefinition definierar rörelsemönstret och FollowPath hanterar själva förflyttningen. Inmatning till PathDefinition utgörs av en lista med Transform-komponenter som representerar rörelsemönstrets olika koordindater medan FollowPath tar en parameter som bestämmer plattformens förflyttningshastighet. FollowPath avgör dessutom om plattformen ska stanna vid sista koordinatern eller periodiskt röra sig fram och tillbaka. PathDefinition och FollowPath utgör tillsammans en generell funktionalitet som kan appliceras på vilket spelobjekt som helst för att få det att följa varierande rörelsemönster. De olika fienderna som rör sig i förutbestämda mönster använder även de ovan nämnda skripten.

För att hindra spelkaraktären från att glida av plattformar som rör sig utnyttjas spelobjektens hierarkiska egenskaper. När spelkaraktären beträder plattformen sätts dess spelobjekt som barn till plattformens spelobjekt. Spelkaraktären ärver på så sätt förälderns Transform-komponent, och därmed position, under hela kontakten.

En annan typ av rörliga spelobjekt är de himlakroppar som befinner sig i omloppsbana kring

andra spelobjekt. De är implementerade med hjälp av ett skript, som applicerat på himlakroppen, påverkar dess Transform-komponent ett gradantal per tidsenhet runt ett centrerat spelobjekt. Radien hittas genom att jämföra spelobjektens Transform-komponenter. Skriptet erbjuder också möjligheten att successivt minska radien. Då ett givet avstånd nås förstörs spelobjektet eller återgår till sin ursprungliga position.

4.1.5 Spelkaraktären och dess förmågor

Spelkaraktären är uppbyggd av en Rigidbody2D-komponent, skriptet Player, en Collider2D-komponent för samt två övriga Collider-komponenter som kontrollerar interaktioner med mark och väggar. Spelobjektet renderas med en SpriteRender-komponent. Vissa av spelkaraktärens förmågor har visuella och auditiva effekter. De är inlagda via Unitys editor för att vidare läsas in i Player som initialiserar dem i respektive metodanrop. All inmatning från spelaren läses in genom skriptet PlayerController, som återfinns som komponent på spelobjektet Managers. Det aktiverar i sin tur respektive metoder i Player.

I Player finns en metod vid namn Movement som agerar funktionspekare. I utgångsläget pekar Movement på den metod som får spelkaraktären att gå. Då knappen för att springa är intryckt ändras Movement till att istället peka på den metod som får spelkaraktären att springa. Spelkaraktärens Rigidbody2D-komponent erhåller en fysikalisk kraft i x-led, vars magnitud varierar beroende på om spelkaraktären springer eller går. Kraften multipliceras med ett flyttal som PlayerController genererar beroende på om vänster- eller höger- knapp är intryckt, där -1 är fullt utslag åt vänster, 1 åt höger och 0 är stillastående, vilket ger kraften rätt riktning och magnitud.

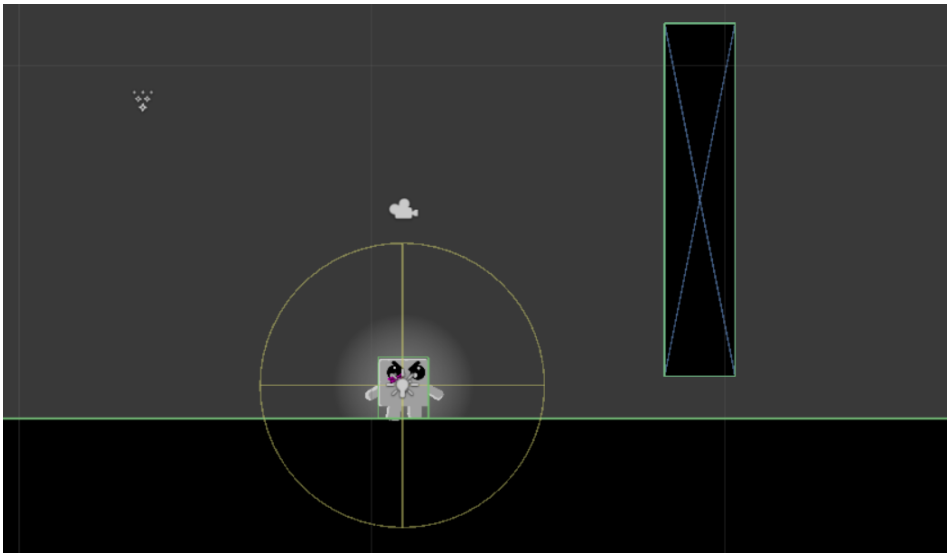
All funktionalitet för att utföra ett standardhopp finns i den specifika metoden Jump, vilken applicerar en fysikalisk kraft i y-led på spelkaraktärens Rigidbody2D-komponent. Jump aktiveras kontinuerligt så länge hoppknappen är intryckt. Släpps knappen innan spelkaraktären nått hoppets maxhöjd ändras Rigidbody2D-komponentens hastighet i y-led till noll. Hoppet blir då kortare och höjden kan således kontrolleras av spelaren. Eftersom spelkaraktären kan utföra ett extra hopp i luften används en räknare som begränsar antalet hopp spelaren kan utföra innan spelkaraktären återigen behöver komma i kontakt med marken. I Waterworld har spelaren emellertid obegränsat med hopp. Då spelkaraktären kolliderar med en vägg och spelaren samtidigt trycker ned hoppknappen utför spelkaraktären ett vägghopp. Spelkaraktärens Rigidbody2D-komponent erhåller då en fysikalisk kraft riktad snett uppåt bort från väggen. Flera vägghopp kan utföras efter varandra, men efter ett vägghopp kan inte spelkaraktären utföra ett dubbelhopp. Detta sköts med hjälp av räknaren. I fallet då spelkaraktären springer så är hoppkraften i x-led följaktligen större.

Spelkaraktären kan även i vissa situationer behöva röra sig snabbt neråt, det vill säga göra en snabbdykning. En kraft appliceras då på spelkaraktärens Rigidbody2D-komponent i negativt y-led och all rörelse i x-led sätts till noll, förutsatt att inte spelarobjektet är i kontakt med marken.

4.1.6 Spelvärlden

Allt som finns i spelvärlden interagerar med spelkaraktären, med undantag för bakgrunden, förgrunden och skyltarna. Bakgrunden och förgrunden renderas antingen med hjälp av en SpriteRenderer-komponent eller en MeshRenderer-komponent. Till vissa bakgrunder används även Unitys inbyggda partikelsystem, ParticleSystem-komponenten, för att emittera exempelvis bubblor eller ljus.

Andra interagerbara spelobjekt är implementerade med både Collider-komponenter som hanterar händelser via skript och Rigidbody2D-komponenter med fysikaliska krafter. Detta gäller bl.a. för all mark och väggar för att spelkaraktären inte skall kunna röra sig igenom dem. För att skilja mellan vägg och mark har spelkaraktären två dedikerade Collider-komponenter vars uppgift är att hålla uppsikt över vad som är väggar och vad som är mark. Mark och väggars utseende är implementerade som antingen SpriteRenderer- eller MeshRenderer-komponenter. De ligger sorterade i varsitt lager för att kollisionshantering skall ske korrekt. För att kunna skilja dem åt vid kollisioner är de märkta med taggar. Skriptet Player observerar kontinuerligt om spelkaraktären är i kontakt med väggar eller mark. För att veta om spelkaraktären befinner sig på marken undersöks ifall dess Collider-komponenten vid fötterna är i kontakt markens Collider-komponent. Samma sak gäller för väggarna, med skillnaden att man istället observerar Collider-komponenten vid kanterna av spelkaraktären. Vissa spelobjekt, som representerar mark och väggar, har ytterligare ett skript som vid interaktion med spelarkaraktären, se figur 13, utlöses en nedräkning. Spelobjektet förstörs då efter en angiven tid och spelar upp en visuell effekt för att visa detta.



Figur 13: Här är mark, väggar och spelarens kollisionskomponenter markerade

Spelkaraktären interagerar med en mängd fiender och projektiler och i spelet förekommer bland annat källor som med en bestämd periodicitet avfyrar målsökande projektiler. Dessa källor aktiveras då spelkaraktären inträder i en trigger för att sedan avbrytas vid utträde. De innehåller ett skript som tar det spelobjekt som skall avfyras som parameter, samt parametrar för att bestämma intervall, hastighet och startfördröjning. Avfyrningen sker genom att en normaliserad vektor, av typen Vector2, skapas med hjälp av Transform-komponenterna hos

källan och spelkaraktären. Vektorn multipliceras med en konstant för att ge önskad hastighet som sedan appliceras på projektilen. Andra målsökande spelobjekt är fiender som förföljer spelarkaraktären. De har ett skript som påverkar deras Transform-komponenten utefter spelkaraktärens Transform-komponent. Vissa fiender har skript som med ett givet intervall påverkar deras Transform-komponent genom att förändra skalan. På så sätt kan de öka och minska i storlek. Det förekommer även en fiende som vandrar längs marken och vänder när den träffar ett annat spelobjekt. För att fienden ska vända används ytterligare Collider-komponenter som kontinuerligt kontrollerar om en kollision uppstår. Om en annan Collider-komponent korsar dess ramar så vänder fiender. För att fiender och projektiler skall kunna förstöras används skriptet ReloadLevelOnCollision, vilket får scenen att läsas in på nytt om då de kolliderar med spelaren.

De samlingsbara objekt som förekommer på de olika banorna är implementerade med en Collider-komponent och ett skript som roterar spelobjektet genom Transform-komponenten. Då kollision sker mellan spelare och ett samlingsbart objekt körs ytterligare ett skript. Skriptet spelar upp en ljudeffekt genom ett AudioSource-komponent och indikerar till GameController att det samlingsbara objektet har plockats upp. Sedan förstörs objektet och information om hurvida spelaren tog det samlingsbara objektet innan målgång eller inte sparas av GameController.

Space subvärlden innehåller två unika förändringar på spelkaraktärens rörelsemönster som finns beskrivna i avsnitt 2.7.1. För att få spelkaraktären att gravitera mot målet i dessa banor skapas kontinuerligt en normaliserad riktningsvektor mellan målet och spelkaraktären. Den multipliceras med en framtagen konstant för att sedan adderas till spelkaraktärens Rigidbody2D-komponent. För att vända gravitationen byts tecknet på spelkaraktärens gravitations-, kraft- och hastighetskonstanter (för hopp och liknande). Utöver detta roteras även själva spelkaraktären 180 grader genom dess Transform-komponent.

På några av vattenbanorna förekommer målsökande maneter. De står stilla tills spelkaraktären befinner inom ett bestämt avstånd från maneten. När spelaren kommer för nära börjar maneten jaga spelkaraktären. För att ta reda på det momentana avståndet används metoden Distance från Vector3.

4.1.7 Meny

Menyn är implementerad via Unitys egna metoder. Huvudmenyn är en egen scen som är placerad utanför sparfunktionerna. Det innebär att det alltid är samma huvudmeny oberoende av vilken sparfil det är som körs. Huvudmenyn har en Canvas-komponent, vars skalbarhet styrs av kamerans inställningar. Canvas-komponenten innehåller paneler som var och en kan skalas enligt kameran eller ha en fixerad storlek. De har som främsta syfte att hålla underliggande komponenter som exempelvis knappar. Menyerna innehåller en rubrik och knappar för att navigera sig till olika delar av menyn eller utföra en viss operation. Genom Button-komponenten visas text på knappen samt ett skript som exekverar en lista av funktioner då knappen trycks ned. För veta om knappen blir nedtryckt finns ett separat eventsystem som redan vid inladdning av menyn letar upp tryckbara knappar och hur de är kopplade.

Då spelet pausas kommer det upp en pausemeny som hänvisar användaren vidare till me-

toder för att navigera sig till de olika banorna, ändra inställningar eller att ta sig tillbaka till huvudmenyn. Pausmenyn är implementerad på samma sätt som huvudmenyn.

4.1.8 Sparfunktion

Skriptet Saveload sparar och laddar den data som skall behållas mellan olika körningar av spelet. För ändamålet används BinaryFormatter från biblioteket UnityEngine vars metod Serialize omvandlar givna objekt till binärt format och sparar det i filer som vid senare tillfälle kan läsas in, tolkas och återskapas. Varje sparfil är representerad av en C#-klass bestående av en textsträng och en hashmap. Textsträngen innehåller namnet på den bana som senast spelats och hashmapen ett objekt för varje bana som spelaren låst upp. Vart och ett av dessa instanser innehåller i sin tur information kring huruvida respektive bana avklarats, om dess samlarobjekt tagits samt den kortaste tid på vilken banan fullföljts. Då en bana laddas kontrollerar sparfunktionen i tidigare nämnda hashmap om banan redan existerar, annars läggs den till. Vid slutet av en bana sparas informationen att den avklarats samt vilka prestationer som utförts. Samtliga sparfiler finns lagrade i en hashmap där de kan hittas med hjälp av sina respektive filnamn och kan laddas in via huvudmenyn.

5 Diskussion

Detta avsnitt diskuterar projektets arbetsprocess och produkt samt granskar de val som gjorts. Det kommer också presentera olika möjligheter som gavs under projektets gång som exempelvis hantering av tvådimensionell fysik.

5.1 Arbetsprocessen

En faktor som påverkade valet av arbetsprocess avsevärt var att projektgruppen saknade förkunskaper inom Unity. Det enda som stod klart om Unity i början av projektet var att det var en populär utvecklingsmiljö för olika typer av dator- och tv-spel, inklusive plattformsspel i 2D. Ovisshet skapade oklarheter på många plan. Hur fungerar Unity? Hur ser arbetsflödet ut? Vad finns tillgängligt från början och vad behöver utvecklaren skapa själv? Att svara på dessa frågor är ingen trivial uppgift. Det kräver att man väl studerar dessa områden och ålägger många arbetstimmar för att bekanta sig med programmet. Vi planerade därför in två veckors introduktion till Unity, som skedde parallellt med arbetet, för att lära oss verktyget och försöka svara på dessa frågor. I efterhand är det tydligt att tidsdisponeringen i det inledande arbetet varit bristfällig. Eftersom Unity tillhandahåller en så pass fulländad spelutvecklingsmiljö, med så många möjliga tillvägagångsätt, är det en stor utmaning att hitta ett sätt som passar bra för det spel som ska utvecklas. Att på två veckor skapa sig en tillräckligt god kunskapsbas för att bygga en effektiv grund för spelet och forma ett arbetsflöde som gör det lätt att fylla spelet med innehåll visade sig för optimistiskt. Att kontinuerligt applicera nyvunnen kunskap på produkten är dessutom ingen effektiv metod för att kompensera för detta. Ett spel, likt annan mjukvara, har ofta en komplex underliggande arkitektur vars olika delar är beroende av varandra. Det är därför ingen trivial uppgift att strukturera den effektivt. Ett exempel på en inlärningsmetod som förmodligen passat bättre för en grupp nybörjare i Unity innan arbetet med slutprodukten börjar är att utveckla några mindre spel enbart i inlärningssyfte. De första spelen kan då hjälpa projektgruppen att fördjupa sig i både utvecklingsverktyget och arbetsprocessen. Saker kan då experimentellt implementeras utan en djupare eftertanke på om det skall finnas kvar i den slutgiltiga produkten. Detta behöver inte ske i grupp utan medlemmar kan på egen hand implementera idéer som skall testas.

Vi stod efter inlärningsfasen inför valet av hur arbetet skulle delas upp. Metoden som valdes var att tillsammans skapa en programgrund för spelet. Grunden innefattade samtliga av spelakaraktärens förmågor samt den grundläggande funktionaliteten hos spelvärlden. Detta gav oss möjlighet att tillsammans kalibrera spelkänslan eftersom den i huvudsak sitter i spelkaraktärens kontroller. En stor fördel var att alla kände sig delaktiga i utvecklingen vilket skapade större engagemang och förståelse för hur implementationerna fungerade. Ytterligare en konsekvens av det gemensamma arbetet var att ingen halkade efter i inlärningen av utvecklingsmiljön. Det möjliggjorde senare individuella implementationer av spelmekaniska inslag i de olika subvärldarna som därmed fick en större variation i både utseende och innehåll. Diversifieringen av problemlösningar gav upphov till en bredare analys av olika implementationer som sedan kunde vägas mot varandra. Mest nämnvärt är hanteringen av fysik och hur spelkontrollen implementerats.

Projektets syfte, att undersöka hur spelmekanik kan implementeras, stod i opposition till en mer tidseffektiv slutproduktsutveckling. Eftersom mycket tid spenderades på att imple-

mentera samma spelmekanik på olika sätt övervägdes initialt en alternativ arbetsprocess. Alternativet var att komma överens om de olika spelmekaniska inslag som skulle finnas i spelet och sedan fördela implementationsarbetet mellan gruppmedlemmarna. I slutprodukten existerar det en del skript som med det alternativa arbetssättet kunnat generaliseras och på så vis skapat en mer logisk hierarki. Spelet hade då lättare kunnat återanvända egenskaper hos vissa spelobjekt med likanande egenskaper. Exempelvis hade spelaren och fiender kunnat dela samma skript för rörelse. Den alternativa arbetsmetoden kräver dock ett högre individuellt ansvar, och värdefull kollegial granskning som parprogrammering medför går förlorad om arbetet sker individuellt. Vi antog därför att den här arbetsprocessen lämpar sig bättre för en erfaren utvecklingsgrupp. Arbetsprocessen som användes har dock inslag av den alternativa. I senare delar av implementationen hade vi erhållit en större förståelse för utvecklingsmiljön som möjliggjorde en uppdelning av arbetsuppgifter.

En övergripande iakttagelse är alltså att projektet har både syftet att undersöka hur något kan implementeras, samt att skapa en slutprodukt. Detta är två skilda saker som kräver olika arbetsmetodik. För att testa hur man kan göra något, är det fördelaktigt att experimentera med många olika sätt att göra samma sak. För att tidseffektivt skapa en slutprodukt vill man dela upp arbetet och fokusera på olika saker.

5.2 Speldesign och Intressekurvan

Den undersökning vi gjort i speldesign har endast skrapat på ytan av den information som finns att tillgå inom området. Det finns ett flertal faktorer som gör ämnet särskilt komplicerat att studera. Det är för det första väldigt svårt att bedöma huruvida man har en bra speldesign eller inte. Det finns inte några enkelt kvantifierbara storheter och olika människors upplevelse av vad som är bra speldesign skiljer sig utefter personliga preferenser. Många av de texter och böcker vi studerat har också talat om spel i allmänhet där fokuset, i vår uppfattning, oftast placeras på 3D-spel. Slutligen finns också en avsaknad av vetenskaplig forskning inom området att tillgå, i alla fall genom de medel vi hade att tillgå. Något som upptäcktes och återfanns i flera böcker och diskussioner kring speldesign var konceptet med intressekurvan, som beskrivs i avsnitt 2.1. Faktum är att den inte bara dök upp samband med spel utan verkar vara ett viktigt verktyg inom all form av underhållning eller interaktion med publik. Jeff Schell beskriver till exempel dess relevans vid ett cirkusframträdande och i en artikel av GamaSutra används den för att beskriva fortlöpandet av både en film och ett TV-avsnitt [30] [26, s. 250]. Den tycks på många håll framställas som ett universalt medel till framgång varpå vi valde att förlita oss på och centrera designen av spelet kring den.

I sökandet efter kritik mot intressekurvan hittades ingen. Även om vi tror att det är ett bra verktyg för den typen av spel vi producerat så ifrågasätter vi dock att den alltid är användbar. För att bygga ett spel efter intressekurvan krävs att man kan kontrollera spelets flöde och händelser på något sätt. Över de senaste åren har spelscenen exploderat med Survival-spel, exempelvis Rust, MineCraft(Mojang), DayZ(Bohemia Interactive), The Forest(Endnight Games) och Stranded Deep(Beam Team Games). Dessa spel karaktäriseras av att ha öppna världar som spelaren fritt får utforska och skall överleva i, ibland i flerspelarkonfiguration. Detta gör det svårt att försöka kontrollera spelarens upplevelse för att forma den efter kur-

van. Särskilt i spel som Rust och DayZ som är helt beroende av interaktionen mellan olika spelare för att skapa intressanta situationer. Trots detta får dessa spel ofta väldigt högt betyg i användarrecensioner och säljer väldigt bra, många gånger redan i alpha-stadiet genom försäljningsplattformen Steams Early-Access program [31]. Således tycks inte alla spel behöva följa kurvan även om det generellt är ett bra verktyg att använda sig av, i de situationer det är möjligt.

Att se hur andra framgångsrika spel implementerat sin speldesign är inte något man skall underskatta vid utveckling. Det finns mycket att lära genom immitation, och vidareutveckling av tidigare idéer. Super Meat Boy är ett simpelt spel med en mycket robust implementation, och en värdig förebild då man skapar ett actionorienterat 2D-plattformsspel.

5.3 Validitet hos spelkänslan

En aspekt av spelet som har fått mycket uppmärksamhet under arbetet, utan att riktigt representeras i rapporten, är spelkänslan. Det vill säga hur det känns att kontrollera spelkaraktären. Spelkänslan beror på hur spelet är designat och implementerat men är samtidigt subjektiv och är beroende av spelarens egna preferenser. En väsentlig andel arbetstimmar har lagts på att testköra och justera detaljer i implementationen för att skapa rätt känsla hos spelkaraktärens rörelseförmågor. Frågan ifall spelet har en bra spelkänsla eller inte beror därför mycket på vår egen uppfattning. Endast ett fåtal personer utanför projektgruppen har testat och gett återkoppling på spelet. Flera sådana inslag i projektet hade gett arbetet kring spelkänslan mer tyngd. Orsaken till att detta inte gjordes är att det inte fastställdes några mål som specificerar att spelet "ska kännas bra" eftersom spelkänsla är subjektivt och det är svårt att veta om ett sådant mål är uppfyllt. Det kan dessutom anses uppenbart att alltid sträva efter att forma ett spel som känns bra att spela.

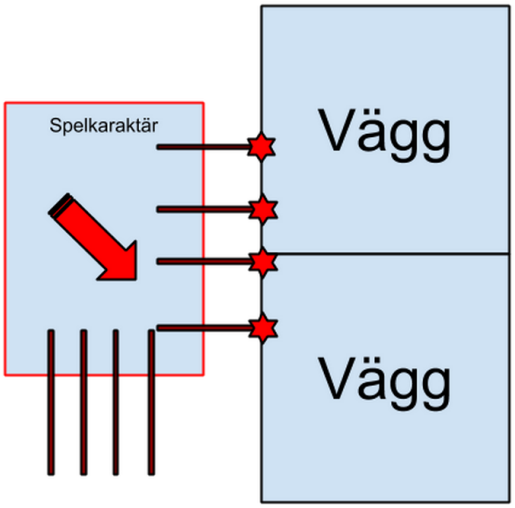
5.4 Hantering av fysik

Fysik i olika spel kan fylla olika roller och variera kraftigt i komplexitet. I vissa spel behövs ingen fysiksimulering överhuvudtaget medan det i andra är väldigt viktigt med realistisk fysik. Unitys fysikmotor kan simulera extremt komplexa fysikaliska situationer, där en situation kan bestå av hundratals objekt av diverse material som rör sig och interagerar med varandra. Slutprodukten utnyttjar fysikmotorn i samtliga av spelkaraktärens rörelseförmågor, genom att applicera fysikaliska krafter på spelobjektet via en Rigidbody2D-komponent. I dokumentationen för RidgidBody2D antyds det däremot att krafter inte är det bästa sättet att förflytta spelkaraktären på. Det är också en åsikt som delas av andra utvecklare på bloggar och forum [32]–[37]. I slutprodukten upptäcktes att det existerade oväntade residualkrafter från vissa typer av rörelser. Ett exempel på detta är när spelfiguren rör sig upp för en lutande backe och stöter på en vägg. Då förväntar man sig att den skall stanna. Istället så får spelfiguren en oavsiktlig residualkraft som gör att figuren far rakt upp längs väggen, utan att hoppa. Detta är oacceptabelt i ett spel som kräver hög och omedelbar precision i sin spelkontroll och fysikhantering. När fenomenet med residualkrafter skulle avlusas fann vi ett alternativt sätt att angripa problemet. Unity har en egen komponent, CharacterController, som fyller syftet att styra en spelkaraktär utan krafter. Även i dokumentation för den komponenten finns en antydan om att det inte alltid är optimalt att styra spelkaraktären med krafter [38]. I CharacterController finns metoden Move som ger ett effektivt sätt att förflytta spelkaraktär och ta

hänsyn till kollision, utan att använda sig av krafter. Tyvärr finns det ett par nackdelar med att använda en CharacterController i ett 2D-plattformsspel. En stor nackdel är att den inte kan användas med Unitys fysikmotor för 2D-fysik. Med andra ord är den inte kompatibel med vissa 2D-komponter. Om vi hade velat använda den här komponenten för att implementera spelkaraktärens rörelseförmågor hade spelet alltså behövts utvecklas precis som ett 3D-spel, men med all fysik låst till två dimensioner. En annan nackdel hos CharacterController är att den är begränsad till att hantera kollision baserat på en kapselformad Collider-komponent. Andra utvecklare, på tidigare nämnda bloggar och forum, som använt sig av CharacterControllern upplever ofta problem med den kapselformade komponenten. Detta för att precisionen i spelkaraktärens rörelserna inte blir tillräckligt bra.

Ett alternativt sätt som löser samtliga problemen med CharacterController är att helt enkelt skapa en egen implementation av komponenten i form av ett skript som härmar dess funktionalitet. Detta innebär att skriptet bör innehålla en metod, Move, som kan förflytta spelkaraktären och samtidigt ta hänsyn till kollisioner. Ett spelobjekt med en CharacterController bör dessutom inte kunna bli påverkat av fysikmotorn genom krafter, vilket är en inställning som enkelt kan göras hos spelkaraktärens Rigidbody2D-komponent. Vanligt bland klassiska plattformsspel är också att spelkaraktären kollisionsyta är formad som en rektangel istället för en kapsel, därför bör Collider-komponenten vara av typen BoxCollider2D.

Den egenutvecklade CharacterController för 2D måste alltså både hantera förfyttning och kollision. Man skulle då kunna utnyttja funktionaliteten bakom Physics2D.Raycast. I implementationen för CharacterController för 2D-spel skickas strålarna från spelkaraktären insida ut mot spelvärlden. Med hjälp av dessa strålar kan man få ut viktig information, framförallt avgöra ifall förflyttningen som spelkaraktären skall göra är möjlig eller behöver förminskas. Längden på strålarna behöver alltså motsvara förflyttningens storlek, på så sätt uppstår det en träff om en Collider-komponent befinner sig i vägen för förflyttningen. Sedan mäts sträckan från utgångspunkten till kollisionen, om någon uppstod, och begränsar förflyttningen efter den. I figur 14 visas ett exempel hur metoden används.



Figur 14: Exempel på hur förflyttning och kollision kan hanteras i en egenutvecklad implementation för CharacterController i 2D-spel.

Strålarna som skickas ut från spelkaraktären kan användas i en mängd sammanhang. De kan bland annat utnyttjas för att implementera ett korrekt rörelsebeteende i uppför- och nerförbackar. I sådana fall undersöks vinkeln som uppstår mellan strålarna och omgivningen (vilket på figur 14 motsvarar 90 grader, eftersom det är en rak vägg). Ett annat användningsområde för strålarna är då spelkaraktären står på en plattform som rör på sig. Hastigheten hos plattformen kan hämtas via de vertikala strålarna som sedan kan används för påverka spelkaraktären. Observera att strålarna utgångspunkt är på insidan av spelkaraktären istället för vid kanterna. Detta är för att ge strålarna en konstant längd så att kollision alltid kan upptäckas även om spelkaraktären står stilla.

En egen version av CharacterController för 2D-spel implementerades under projektet, tillsammans med alla rörelseförmågor hos spelkaraktären. Detta gjordes efter det att implementation för spelkaraktärens förmågor var implementerade i slutprodukten. Det utvecklades alltså totalt två implementationer av spelkaraktärens rörelseförmågor. Anledningen till att två implementationer gjordes var för att kunna jämföra dem och producera en intressant diskussion och eventuellt använda det som gav bäst resultat. Tyvärr, på grund av tidsbrist, kunde inte en överföring av den egenimplementerade CharacterControllern genomföras till slutprodukten.

Att det överhuvudtaget existerar en CharacterController upptäcktes långt efter arbetet med slutprodukten hade påbörjats. Eftersom komponenten inte stödjer 2D-fysik finns den inte i träningsmaterial från Unitys hemsida om utveckling i 2D. Det var inte förrän mer djupgående koncept inom Unity började utforskas som komponenten upptäcktes. Det pekar på att arbetsprocessen som beskrivs i avsnitt 5.1 kanske hade gjort att projektgruppen upptäckt möjligheten tidigare eftersom mer tid då att hade lagts på att utforska Unitys utbud av funktionalitet.

5.5 Oväntade implementationsproblem

I slutprodukten så hanteras spelets banor genom separata scener. Detta var tillsynes en praktisk lösning i början av projektet som isolerade arbetet på varje bana. De versionshanteringskonflikter som uppstod när två eller fler gruppmedlemmar arbetade på samma scen var tidskrävande att lösa, eftersom en scen består av tiotusentals rader automatiskt genererad kod. Beslutet togs därför att separera alla banor till olika scener, vilket gjorde versionshanteringen enkel. Senare upptäcktes dock en oväntad begränsning som medföljde detta val. Varje gång en scen laddas, exempelvis vid förlust eller byte av bana så nollställs inmatningen av kontrollens rörelse. Detta innebar att om till exempel springknappen hålls ner när en scen laddas måste spelaren släppa knappen och trycka ner den igen för att det skall registreras. En sökning på Unitys forum avslöjade varför detta sker. När en scen laddas körs ovillkorligt en metod som nollställer all inmatning[39]. I ett långsammare spel hade detta inte varit något problem, men vårt kräver en snabbare responsivitet. När spelaren dör på en bana, vilket kan hända ganska ofta, var det tänkte att spelaren skulle kunna hålla inne rörelseknappen och börja springa så fort banan laddats om.

Ett avancerat sätt att angripa problemet på hade varit att låta varje värld vara en gigantisk scen. Vi hade kunnat strukturera varje bana som ett separat deaktiverat spelobjekt i scenen, konstruerade så att endast banan spelaren befinner sig på är aktiv. När målet på en bana vidrörs så förflyttas spelarens transform till nästa bana, aktiverar den samtidigt som föregående deaktiveras. Vi hade då också behövt implementera en omfattande nollställare av alla värden på banan som förändrats, exempelvis fienders positioner. Då det bedömdes att denna metod skulle kräva en betydande mängd tid att implementera för att lösa ett tämligen litet problem valdes det bort.

Ett betydligt lättare sätt att lösa problemet är att använda en metod kallad LoadLevelAdditiveAsync[40]. Vad den gör tillskillnad från LoadLevel, som slutprodukten använder, är att den laddar in och lägger till spelobjekten från en annan scen in i den nuvarande scenen. I praktiken betyder detta att nästa scens objekt läggs rakt ovanpå den nuvarande scenen. Genom att implementera ett sätt att ta bort de gamla spelobjekten på scenen hade detta varit ett effektivt sätt att komma runt nollställningen. Den här lösningen upptäcktes väldigt sent och kunde därför inte implementras men bedöms avsevärt mer genomförbar än metoden ovan.

Ett till oväntat problem var hanteringen av spelkaraktärens förflyttning på rörliga plattformar. Det förväntade beteendet när spelkaraktären står på en plattform är att den förflyttas tillsammans med plattformen. Istället åker plattformen iväg och spelfiguren glider av. Den första lösningen som utvecklades för att komma runt detta problemet innebar att man beräknade plattformens momentana hastighet i x- och y-led och adderade den till spelfigurens hastighet. Detta gav ett funktionellt resultat men visade sig vara en onödigt komplicerad lösning. I Unity ärver spelobjekt som ligger placerade i hierarkin under ett förälderobjekt dess transforms position. Därför sätts i vår lösning spelfigurens objekt som ett barnobjekt till plattformen då spelfiguren kolliderar med plattformen och följer därför med dess förflyttning.

6 Slutsats

De mål som sattes upp för projektet har uppfyllts med undantag för en utmärkelselista. Trots detta kan en del problem identifieras i slutproduktens implementation. För att hantera rörelsen på karaktären användes Unitys fysikmotor vilket senare visade sig medföra oväntade fysikaliska beteenden då spelets komplexitet ökade. När sedan möjligheten att implementera spelkaraktärens rörelser utan fysikmotorn upptäcktes hade processen emellertid fortlöpt till en sådan grad att förändring skulle leda till ett omarbete som ej bedömdes genomförbart under den projekttid som fanns kvar. De tester som gjordes indikerar att den alternativa hanteringen av fysik skapar en högre precision i ett spel med så höga krav på en responsiv kontroll som projektets, och kan därför ses som mer lämplig.

I projektet studerades speldesign. Det är ett svårt ämne att studera eftersom det är beroende av personliga preferenser. Det finns också en avsaknad av tidigare forskning inom området. Ett verktyg som varit användbart vid designen av andra spel av samma karaktär är den så kallade intressekurvan. Intressekurvan behöver dock inte vara en optimal modell att utgå ifrån i all spelutveckling men har visat sig passa projektet bra. Konceptet har därför genomgående utnyttjats i spelutvecklingen. Att ha Super Meat Boy som inspirationskälla för vårt projekt gav även en bra vägledning för hur olika designproblem kunde lösas.

Arbetet har gett en djupare förståelse för spelmotorn Unity som ett kraftfullt och kostnadsfritt verktyg för aspirerande spelutvecklare. Den arbetsplanering som användes visade sig vara för tidsoptimistisk, vilket lyser igenom i delar av spelets kvalitet. Många problem som uppstått under utvecklingen har givit en djupare förståelse för den komplexitet som återfinns i utvecklingsprocessen av ett mer omfattande spelprojekt. Om liknande projekt skall göras i framtiden rekommenderas att disponera mer tid till inlärning och praktisk övning med utvecklingsverktyget, att tidigt fastställa spelets spelmekaniska inslag samt mer effektivt abstrahera funktionalitet.

Referenser

[1] Newzoo. (23 juni 2014). Top 100 countries represent 99.8% of $81.5bn global games market. http://www.newzoo.com, URL: http://www.newzoo.com/insights/top-100-countries-represent-99-6-81-5bn-global-games-market/ (hämtad 2015-02-08).

[2] Gartner. (29 okt. 2013). Gartner says worldwide video game market will reach $93 billion in 2013. www.gartner.com, URL: http://www.gartner.com/newsroom/id/2614915 (hämtad 2015-02-13).

[3] M. Nayak. (13 juni 2013). Factbox - a look at the $66 billion video-games industry. http://in.reuters.com, URL: http://in.reuters.com/article/2013/06/10/gameshow-e-idINDEE9590DW20130610 (hämtad 2015-02-13).

[4] S. E. Siwek. (12 nov. 2014). Video games in 21st century, the 2014 report. ESA, URL: http://www.theesa.com/wp-content/uploads/2014/11/VideoGames21stCentury_2014.pdf (hämtad 2015-01-08).

[5] Newzoo. (15 maj 2014). Global game markets will reach $102.9 billion in 2017. www.newzoo.com, URL: http://www.newzoo.com/insights/global-games-market-will-reach-102-9-billion-2017-2/ (hämtad 2015-02-13).

[6] Statista. (). Compound annual growth rate of global video game industry revenue between 2013 and 2018. www.statista.com, URL: http://www.statista.com/statistics/307326/growth-of-global-video-game-industry-revenue-platform/ (hämtad 2015-02-13).

[7] Nielsen. (). Multi-platform gaming: For the win! Nielsen, URL: http://www.nielsen.com/us/en/insights/news/2014/multi-platform-gaming-for-the-win.html (hämtad 2015-06-01).

[8] ESA. (15 okt. 2014). The 2014 essential facts about the computer and video game industry. http://www.theesa.com, URL: http://www.theesa.com/wp-content/uploads/2014/10/ESA_EF_2014.pdf (hämtad 2015-02-08).

[9] M. Overmars. (30 jan. 2012). A brief history of video games. http://www.cs.uu.nl, URL: http://www.cs.uu.nl/docs/vakken/b2go/literature/history_of_games.pdf (hämtad 2015-02-13).

[10] D. Giltinan. (14 juli 2014). The rise and fall (and rise again!) of 2d-platformers. http://readretro.com/, URL: http://readretro.com/features/rise-fall-rise-2d-platformers/ (hämtad 2015-02-13).

[11] Metacritic. (2015). Braid: Xbox 360 (2008). www.metacritic.com, URL: http://www.metacritic.com/game/xbox-360/braid (hämtad 2015-02-13).

[12] ——, (2015). Super meat boy xbox 360(2010). http:// www.metactitic.com, URL: http://www.metacritic.com/game/xbox-360/super-meat-boy (hämtad 2015-02-09).

[13] E. Caoili. (3 jan. 2012). Super meat boy sells 1 million copies. Gamasutra, URL: http://www.gamasutra.com/view/news/128937/Super_Meat_Boy_sells_1_million_copies.php (hämtad 2015-02-09).

[14] W. Harrel. (9 aug. 2013). The indie revolution: How little games are making big money. http://www.gameacademy.com, URL: https://www.gameacademy.com/the-indie-revolution/ (hämtad 2015-02-09).

[15] Unity. (9 juli 2014). Unity technologies doubles community to two million developers. http://unity3d.com, URL: http://unity3d.com/company/public-relations/news/unity-technologies-doubles-community-two-million-developers (hämtad 2015-02-08).

[16] U. Engine. (2015), URL: https://www.unrealengine.com/ (hämtad 2015-02-09).

[17] Unity. (2015), URL: http://unity3d.com/showcase/gallery/games (hämtad 2015-02-13).

[18] ——, (19 juni 2014). Card life: Hearthstone by blizzard entertainment. www.unity3d.com, URL: http://unity3d.com/showcase/case-stories/hearthstone (hämtad 2015-02-13).

[19] T. M. Blog. (), URL: http://supermeatboy.com/ (hämtad 2015-02-13).

[20] J. Björk, J. Ljungdal och J. Bosch, "Lean product development in early stage startups.", *PCEUR Worshop Proceedings: From Start-ups to SaaS Conglomerate - Life Cycles of Software Products Workshop*, vol. 1095, s. 19–32, juni 2013.

[21] A. Järvi, T. Mäkilä och S. Hyrynsalmi, "Game development accelerator — initial design and research approach", *PCEUR Worshop Proceedings: From Start-ups to SaaS Conglomerate - Life Cycles of Software Products Workshop*, vol. 1095, s. 47–58, juni 2013.

[22] S. Rogers, *Level Up! The Guide to Great Video Game Design 2nd Edition*. Chichester John Wiley och Sons Ltd, 2015.

[23] F. P. M̃. M. Müller, "Analyzing the cost and benefit of pair programming", s. 166–177, sept. 2003.

[24] S. Wray, "How pair programming really works", *Ieee Software*, vol. 27, nr 1, s. 50–55, jan. 2010.

[25] K. M. Kapp, "Games, gamification, and the quest for learner engagement", *T+D*, vol. 66, s. 64–68, juni 2012.

[26] J. Schell, *The Art of Game Design A Book of Lenses*. 30 Corporate Drive, Suite 400, Burlington, MA 01803, USA: Morgan Kaufmann, 2008.

[27] E. Credits. (). Making your first game: Minimum viable product - how to scope small and start right. Youtube.com, URL: https://www.youtube.com/watch?v=UvCri1tqIxQ (hämtad 2015-05-12).

[28] D. Hii, "“zlayer: Simulating depth with extended parallax scrolling", *VRST '97 Proceedings of the ACM symposium on Virtual reality software and technology*, vol. 1, s. 65–69, sept. 1997.

[29] A. Perspective. (). Encyclopaedia britannica online academic edition. Encyclopaedia Britannica Inc, URL: http://academic.eb.com.proxy.lib.chalmers.se/EBchecked/topic/7229/aerial-perspective (hämtad 2015-05-13).

[30] M. Lopez. (). Gameplay fundamentals revisited: Harnessed pacing & intensity. gamasutra.com, URL: http://www.gamasutra.com/view/feature/3848/gameplay_fundamentals_revisited_.php?print=1 (hämtad 2015-06-02).

[31] B. Parfitt. (). Rust overtakes dayz to become best-sellingsteam title, makes $8.8m. gamerevolution.com, URL: http://www.mcvuk.com/news/read/rust-overtakes-dayz-to-become-best-selling-steam-title-makes-8-8m/0126942 (hämtad 2015-06-02).

[32] (). Game development stack exchange. Encyclopaedia Britannica Inc, URL: http://gamedev.stackexchange.com/questions/22348/2d-character-controller-in-unity-trying-to-get-old-school-platformers-back (hämtad 2015-03-10).

[33] (18 maj 2014). 2d platformer: Rigidbody ord ray casts? Reddit.com, URL: http://www.reddit.com/r/Unity2D/comments/1wbubt/2d_platformer_rigidbody_or_ray_casts/ (hämtad 2015-03-10).

[34] N. Dimucci. (17 dec. 2013). 2d platformer collision detection in unity. Overdevelop, URL: http://overdevelop.blogspot.se/2013/12/2d-platformer-collision-detection-in.html (hämtad 2015-03-10).

[35] Y. Pignole. (10 aug. 2013). The hobbyist coder #1: 2d platformer controller. Gamasutra.com, URL: http://www.gamasutra.com/blogs/YoannPignole/20131010/202080/The_hobbyist_coder_1_2D_platformer_controller.php (hämtad 2015-03-10).

[36] T. Martin. (4 jan. 2014). 2d platformer collision detection with raycast, part 1: Gravity. The Deranged Hermit's Devlog, URL: http://deranged-hermit.blogspot.se/2014/01/2d-platformer-collision-detection-with.html (hämtad 2015-03-10).

[37] Unity. (). Rigidbody2d. Unity3D.com, URL: http://docs.unity3d.com/Manual/class-CharacterController.html (hämtad 2015-03-10).

[38] ——, (). Charactercontroller. Unity3D.com, URL: http://docs.unity3d.com/Manual/class-CharacterController.html (hämtad 2015-03-10).

[39] ——, (). Input.resetinputaxes. Unity3D.com, URL: http://docs.unity3d.com/ScriptReference/Input.ResetInputAxes.html (hämtad 2015-06-02).

[40] ——, (). Application.loadleveladditiveasync. Unity3D.com, URL: http://docs.unity3d.com/ScriptReference/Application.LoadLevelAdditiveAsync.html (hämtad 2015-06-02).