

**Prosjektrapport**

3D Spillutvikling med Unity

Utviklings-rapport for spillprosjektet «Pepigo»



Utført av Jan Andreas Sletta

Innhold

[Introduksjon 2](#_Toc41042713)

[Sammendrag 2](#_Toc41042714)

[Visjon 2](#_Toc41042715)

[Oppsett 2](#_Toc41042716)

[Administrasjon 3](#_Toc41042717)

[Solo prosjekt 3](#_Toc41042718)

[Arbeidsrutiner og tidsbruk 3](#_Toc41042719)

[Framdriftsplan 4](#_Toc41042720)

[Risiko 6](#_Toc41042721)

[Versjons-kontroll 6](#_Toc41042722)

[Feilsøking og testing 7](#_Toc41042723)

[Utviklingsmetode 7](#_Toc41042724)

[Læringsmetoder 7](#_Toc41042725)

[Programvare for spillutvikling 8](#_Toc41042726)

[Implementeringsmetode 8](#_Toc41042727)

[Planen var waterfall 9](#_Toc41042728)

[Utviklingen 9](#_Toc41042729)

[Teknisk 9](#_Toc41042730)

[Sammenkobling mellom Unity og C# 10](#_Toc41042731)

[Teknisk inkompatibilitet 11](#_Toc41042732)

[Unity hierarkiets ferdigheter 12](#_Toc41042733)

[3D Animasjon 13](#_Toc41042734)

[Kode-prinsipp under Monobehavior 14](#_Toc41042735)

[Prosessen 15](#_Toc41042736)

[Hvordan ideen og realiteten kolliderte 15](#_Toc41042737)

[Prosjekt-omstart 16](#_Toc41042738)

[Fristelsen av å bruke gratis ferdiglagde modeller 17](#_Toc41042739)

[Begrensninger og tidspress 17](#_Toc41042740)

[Systemdokumentasjon 18](#_Toc41042741)

[Spillerkontroll 18](#_Toc41042742)

[Fiende AI 19](#_Toc41042743)

[GUI, resolusjon og grafikk 20](#_Toc41042744)

[Bilde-effekt 21](#_Toc41042745)

[Modellering og animasjon 21](#_Toc41042746)

[Komponentbasert oppsett 23](#_Toc41042747)

[Konklusjon 24](#_Toc41042748)

[Et ambisiøst prosjekt 24](#_Toc41042749)

[Måloppnåelsen 24](#_Toc41042750)

[Erfaringer og utbytte 25](#_Toc41042751)

[Format 26](#_Toc41042752)

[Utgivelse og publisitet 26](#_Toc41042753)

[Referanser 27](#_Toc41042754)

[Kilder 27](#_Toc41042755)

# Introduksjon

## Sammendrag

Oppgaven jeg la ut for meg selv var å lage et 3D-spill med Unity og publisere det til en spill-plattform. Spillet skulle basere seg på en spiller-kontroll fra tredje-perspektiv som hadde ulike egenskaper, der man kamper mot en eller flere fiender i en håndlagd verden. Prosjektet ville kreve ulike tekniske midler som programmet Unity som spillmotor, Blender for modellering og Visual studio som kode editor, samt lære de forskjellige systemene og dokumentere de.

## Visjon

I starten av prosjektet hadde jeg en veldig svak visjon på hva jeg faktisk ville lage. Jeg visste jeg ville lage et spill som var 3D, hadde spiller-kontroll med visse ferdigheter, håndlagd verden og modeller og et kampsystem, men det var alt. Jeg hadde ikke godt bilde av hvordan det ferdige spillet ville se ut, bare hva jeg ville inkludere. Ideen hang løst, men jeg tror det var en god ting og hjalp meg med å utvikle og gjøre ferdig spillet.

Jeg tror det blir litt som å skrive en bok. Du vet hvordan du vil starte den, hva den skal handle om og kanskje til og med hvordan den skal avsluttes, men du vet ikke nødvendigvis hva hver side skal inneholde. Jeg visste aldri hvordan ting skulle se ut før jeg starta å lage de, hvilke script jeg kom til å kode eller hvordan kampsystemet ville se og føles ut, men jeg tror det kanskje var den største motivasjonsfaktoren og holdt prosjektet interessant for meg selv. På mange måter vil jeg kalle dette prosjektet en lang improvisasjon.

## Oppsett

Når det kommer til rapport og dokumentasjon vil jeg legge mer vekt på den faglige beskrivelsen som dokumentasjon av prosessen, design og forklaringer av ulike systemer ved at dette er et ganske ambisiøst solo-prosjekt. Jeg kommer til å legge vekt på utviklingen og de hindrene jeg traff, og samtidig prøve å forklare de systemene jeg har vært innom på en god og oversiktlig måte.

Koding i Unity er også script-basert, som betyr at hvert individuelle objekt har egne script som ofte består av en uavhengig klasse, samtidig som visjonen av spillet er basert relativt mye på modellering, design og sammensetning av forskjellige komponenter. Unity er komponent-basert, men kan også framstå som å være bygget på MVC-prinsippet fra et helhetlig program-perspektiv. Derfor vil jeg ikke bryte den tradisjonelle Unity-strukturen ved å legge på flere lag med strukturerte klasse-oppsett, kode-prinsipp eller originale komponenter(objekter) der jeg ikke behøver. Jeg vil bruke sjansen til å gå grundig inn i de ulike systemene og forklarer hvordan ting funker, hvor fallgruvene ligger og hvordan prosessen var.

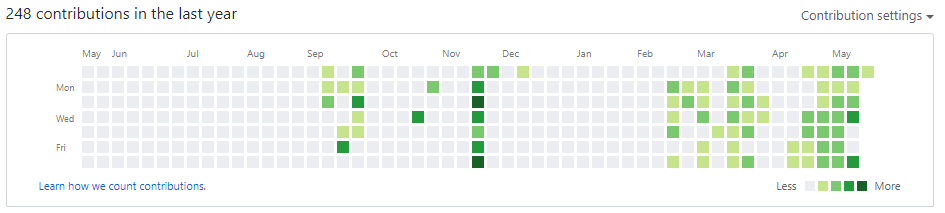
# Administrasjon

## Solo prosjekt

Siden det er et solo prosjekt, var det opp til meg selv å organisere meg selv med rutiner og metoder for å fullføre prosjektet. Dette ligger det naturligvis svakheter i som å ikke ha en gruppe til å motivere hverandre og ikke ha faste fysiske oppmøter å holde seg til. Jeg fikk heller ikke muligheten til å prøve meg på utviklings-strategier som sprinter og par-programmering. Men dette var et prosjekt jeg var forberedt på å fullføre alene uavhengig av hvilke fordeler både praktisk og teoretisk et gruppeprosjekt ville gi meg. Jeg hadde sterk motivasjon og et godt grunnlag i programmering til å komme meg gjennom det.

## Arbeidsrutiner og tidsbruk

I prosjektbeskrivelsen sa jeg at jeg ville bruke 3 timer hver dag alle ukedagene fram til innlevering, noe jeg har klart å fullføre og mer. I starten var det vanskelig å treffe 3 timer hver dag med frustrasjon over læring og masse feil som bygget seg opp, men utover slutten satt jeg fulle dager og jobba med ulike deler i prosjektet.

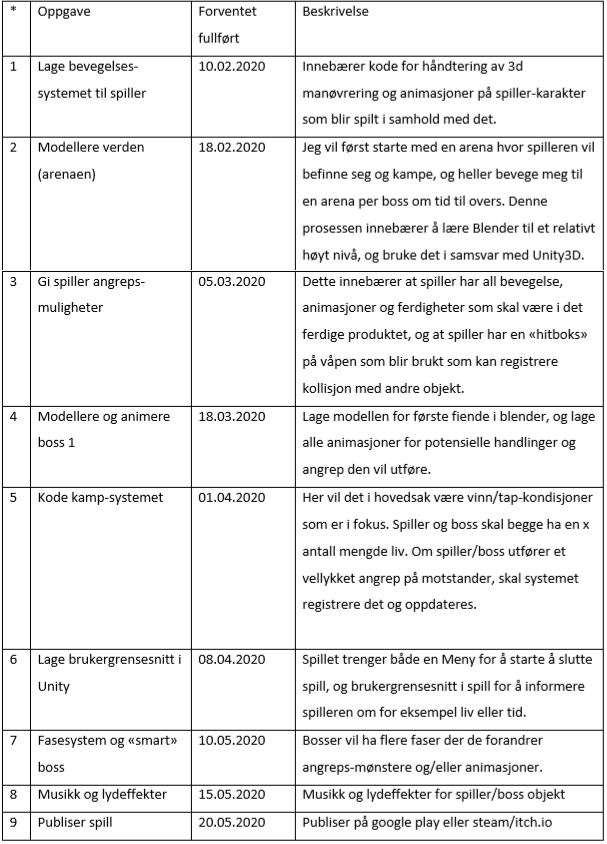


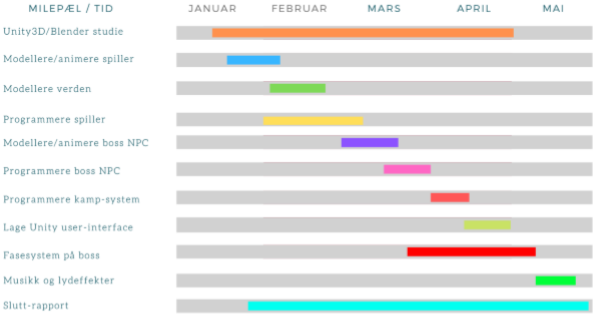
Jeg tror GitHub aktiviteten godt reflekterer prosessen. Hele januar og februar gikk bort til læring der jeg ikke brukte versjons-kontroll, og jeg følte meg ikke komfortabel nok til å starte på prosjektet før i tidlig mars. I april gikk all tiden inn i Blender da jeg måtte virkelig sette meg inn i ting for å modellere/animere fienden, i tillegg til å re-designe spilleren. I slutten av april startet jeg å programmere fienden og fasesystemet, og som jeg forutså, så tok det tid og ble en test for programmerings-ferdighetene mine. I konklusjon var nok Blender den tyngste prosessen i å både lære og lage spill-klare modeller.

## Framdriftsplan

I startfasen av prosjektet var jeg bestemt på å følge framdriftsplanen som den var satt opp i prosjektbeskrivelsen, og mens noe av det klarte jeg å følge (punkt 2 og 7), havnet resten i resirkulering der jeg måtte gjennom flere av stegene flere ganger.

Jeg så tidlig at framdriftsplanen ikke var realistisk da jeg måtte sette spillerkontrollen på vent til jeg visste hva som faktisk krevdes av den. Spillerkontrollen var estimert til å være ferdig 10 februar, og ble ikke helt ferdig før i slutten av april med at jeg gikk tilbake og lagde en ny modell med nye animasjoner mot slutten. Det var en prosess som varte prosjektet ut, på samme måte som de fleste punkt der jeg måtte hele tiden gå tilbake å justere på ting, eller bare sette meg ned for å lære meg noe grundig fordi jeg gjorde det ikke ordentlig og lot det stå åpent for feil. I retrospektiv visste jeg aldri om jeg ville klare å gjøre ferdig prosjektet innen fristen.





## Risiko

Før prosjektstart så jeg for meg diverse risikofaktorer som kunne halte prosjektet og i verste fall hindre meg i å oppnå målene jeg hadde satt. Den største risikoen var helt klart tid og ferdigheter som muligens ikke ville være nok. Jeg hadde aldri tatt for meg et prosjekt på denne størrelsen, og det med nye system jeg måtte lære for å utføre det. En konsekvens av dette ville eventuelt være at jeg ikke ble ferdig, eller at spillet jeg hadde sett for meg krevde for mye og at jeg måtte degradere det til noe utenfor visjonen min.



En risiko som jeg forutså ville ha middels konsekvens, men som jeg ikke tenkte ville skje, var den tekniske risikoen hvor enten datamaskinen skulle slutte å funke eller at prosjektet ble korrupt. Til stor overraskelse skjedde begge delene under utvikling, og hadde jeg ikke vært nøye med versjonskontroll og dytta prosjektet til GitHub flere ganger om dagen, ville mye arbeid gått tapt. Disse risikoene skjedde heldigvis samtidig, da noen filer i prosjektet ble korrupte samme dagen det sa stopp for datamaskinen, så alt jeg mistet var to setninger med kode. Resten av prosjektet utførte jeg på en lånt bærbar pc. Etter det slo det meg inn ekstra mye hvor viktig det er å holde prosjektet oppdatert på GitHub og ikke bare lokalt.

## Versjons-kontroll

For prosjektet brukte jeg GitHub sin GitHub desktop applikasjon som Git-tjeneste for å holde prosjektet under kontroll og redusere risikoen for å miste det. Versjonskontroll har vært veldig nyttig i prosjektet siden det hadde vært vanskelig og holdt oversikt på alle typer filer som går ut og inn av Unity-editoren til tider. Vanligvis har jeg brukt versjons-kontroll til å se forandringer i koden, men denne gangen brukte jeg det hovedsakelig til få en oversikt på hvor diverse filer ligger. Prosjektet ligger åpent på GitHub profilen min (Link i Kilder under Konklusjon).

## Feilsøking og testing

Testing i Unity besto ofte i å gå inn i kjøremodus, spillteste og se om ting funker som de skal. Siden prosjektet er sterkt basert på komponentoppbygning (se komponentbasert oppsett under Systemdokumentasjon), var det viktig å sette variabler til *Public* så man kan se hva som skjer med variabelen under kjøretid i editoren. Strategien rundt feiltesting gikk ut på å finne ut hvilken variabel feilen var sentrert rundt, sette objektet til *Public* og deretter følge med på hva som skjer med referansen/verdien. Denne prosessen ble også fulgt av et Debug.log() kall i koden, som er *Monobehavior* (Se sammenkobling mellom Unity og C# under Utviklingen) sin måte å skrive ut i konsollen på. I mitt prosjekt ble denne prosessen ekstra viktig da grunnlaget for spill mekanikker ligger mye i sjonglering av verdier og referanser.

# Utviklingsmetode

## Læringsmetoder

Lærings-prosessen var muligens det som tok opp den største tiden av prosjektet, og jeg følte at det var en prosess som varte prosjektet ut. For å lære Unity så jeg først og fremst på forskjellige *Youtube*-kanaler for å visuelt se hvordan ting ble satt opp komponentbasert i Unity editoren. Her brukte jeg hovedsakelig kanalen *Brackeys* som dekker for det meste alt man skulle treffe på i Unity på en god oversiktlig måte. Deretter brukte jeg generelt Unity sin egen dokumentasjon rundt *Monobehavior* for bruksmanual på spesifikke metoder innen *Transform* (se Teknisk under Utviklingen) og lignende. Unity har også en seksjon på nettsiden der mennesker kan stille spørsmål rundt noe de sitter fast på, og andre kan svare. Dette ligner på *Stackoverflow* sitt oppsett der den «mest riktige» kommentaren ofte ligger øverst og er lett å se. Jeg tok også i bruk et Unity samfunn på *Reddit* for rask respons på problem man skulle treffe på.

Den vanskeligere lærings-delen har helt klart vært Blender der en kode-bakgrunn ikke hjalp meg noe. Å lære Blender til et punkt hvor jeg var komfortabel med det tok meg ukesvis, og selv om de har mye god dokumentasjon, var det ikke lett å forstå meg på ukjente begrep og konsept. Her tok jeg hovedsakelig i bruk *Youtube*-kanaler som jeg fulgte utrolig nøye, i kombinasjon ved å spørre om hjelp i et Blender samfunn på *Reddit*. Alle disse hjelpemidlene har jeg lagt under referanser i tillegg til *Reddit*-brukeren jeg har opprettet tråder med.

## Programvare for spillutvikling

Til prosjektet tok jeg i bruk spillmotoren Unity, modellering og animasjons verktøyet Blender, kode editoren Visual studio og bilde-redigerings programmet Krita. Unity er gratis for både ikke-kommersielt og kommersielt bruk så ikke det utviklet produktet ikke overstiger 100 000 kr i profitt. Siden planen for spillet var å legge det ut gratis på en plattform var ikke dette noe problem, og jeg fikk tatt i bruk alle verktøy Unity kan tilby under prosjektet. Visual studio er standard kode-editor som Unity ber brukeren om å installere om man ikke har det.

Blender er alternativet for modellering/animasjon som de fleste nye utviklere går for siden det er gratis i forhold til motparten Maya som koster, men er muligens mer profesjonelt og har flere verktøy. Blender ga meg alle verktøy jeg hadde behov for når det kom til å lage forskjellige objekt, legge farge på og animere de.

Et verktøy jeg ikke forventet å få bruk for var et bilde-redigerings verktøy. Spillet har rundt 30 forskjellige effekter som bruker Unity sitt innebygde *Particle-System*. Disse effektene baserer seg ofte på «*sprites*» eller bilder som grunnlag for effekter, men her er det veldig begrensa hva man kan lage med standard sirkler og rektangler. Derfor gikk jeg inn for å lage mine egne «*sprites*» med et bilde-redigerings program. Jeg valgte å gå for Krita framfor Photoshop siden Krita er gratis og tilbyr mye av de samme egenskapene.

## Implementeringsmetode

Spillutvikling går ofte ut på å tenke seg til konsept, også utvikle dem. Vanligvis vil man tenke seg til konseptet og legge opp utviklingen i bestemte deler for å få en oversikt over kompleksitet i forhold til tidsbruk. I mitt tilfelle visste jeg ikke hva konseptet ville være utenom «*Boss-Battle*» og heller ikke hvor komplisert arbeidsprosessen var. Derfor kom jeg fram til en utviklingsmåte der jeg tok faste små steg framover.

Hvis vi tenker oss et fullstendig konsept av å lage en «*Boss-Battle*», inkluderer dette mange store deler som er i direkte relasjon til hverandre. En modell har et fast sett med animasjoner og et script som styrer animasjoner er begrenset til disse. Derfor var det viktig at jeg startet å koble sammen disse relasjonene tidlig. Jeg startet med å sette opp minimumskrav for konseptet i en helhet som innebar en spiller, en fiende, en arena, kollisjonsbokser og angrep. Unity har innebygde spill-objekt som firkanter og andre simple figurer som jeg la opp minimumskrav med. I den tidligste versjonen satt jeg med et rektangel som representerte både spiller og fiende der spiller hadde begrensa muligheter til å bevege seg rundt med et prosjektil for å teste kollisjon. Etter minimumskrav rundt målene jeg hadde satt meg var i boks, startet jeg å bygge på lag for lag med samme prosessen rundt å gjøre det å gjøre neste lag så simpelt som mulig for testing.

## Planen var waterfall

Før prosjektstart var planen å kjøre en «*waterfall*» utviklingsmetode der jeg ble ferdig med én ting først og bevegde meg til neste. Dette var fordi inntrykket mitt av spillutvikling i startfasen var at de tidligere stegene satte grunnlaget for de senere stegene, som for eksempel modellering etterfulgt av animasjon. Jeg trodde det å bruke en «*agile*» utviklingsmetode ville bare sette meg tilbake til starten uten framgang. Derfor brukte jeg mye tid i starten av utviklingen til å forsøke å gjøre alle stegene perfekt før jeg bevegde meg videre. Dette var ikke veldig produktivt siden jeg raskt fant ut at jeg måtte forandre på ting når jeg innså at jeg hadde satt meg for høye krav. Jeg var også redd for å gjøre feil eller ødelegge noe siden jeg hadde gått så nøye inn/brukt så mye tid på et steg at det ville ha fatale konsekvenser.

Etter prosjekt-omstart (se Utviklingen), bestemte jeg meg heller for å prøve en «*agile*» utviklingsmetode der jeg gikk fra modellering til koding i Unity og tilbake i samme økt. Dette gjorde meg etter hvert såpass komfortabel med programmene at alle stegene i spill-utviklingen (modellering, rigging, animasjon, koding, komponent-oppbygging og effekter), ble til én. Tidligere hadde jeg brukt dager til uker på ett steg i prosessen, for eksempel bygge en modell jeg var fornøyd med, mens mot slutten ville jeg ofte gå tilbake til blender-filen, legge på et bein til riggen, animere det og sette det opp mot script i Unity på noen minutter. Jeg tror utviklingsmetoden for mennesker som vil inn i spillutvikling lander automatisk på «*waterfall*» fordi alt er nytt og man trenger en forståelse og bli komfortabel før man sjonglere de ulike stegene. På samme måte kan «*waterfall*» ha større verdi en «*agile*» for veteraner som vet akkurat hvilke prosesser de må gjennom og hva som kreves av grunn stegene.

# Utviklingen

## Teknisk

### Sammenkobling mellom Unity og C#

Måten C# blir iverksatt inn i Unity-motoren er ved å arve klassen *Monobehavior*. *Monobehavior* er base-klassen som alle Unity script bør arve fra for å kunne overskrive viktige metoder som Awake(), Start() og Update(). *Monobehavior* gir oss også tilgang til å manipulere nærmest alt Unity har å tilby via kode, som for eksempel fysikk, trykk-hendelser, objekter i scenen og kontroll over hver «*frame*».

*GameObject* er base-komponenten i Unity som kan representere alt fra spiller-karakterer til et kamera, som enten ligger i scenen før kjøretid eller blir instansiert under kjøring. Alt starter som et tomt *GameObject* på samme måte som en abstrakt klasse ikke har egen konkret kode, og blir bygd opp ved å legge til andre komponenter som eksempel animatører, kollisjonsbokser, renderer og fysikk. Disse komponentene kan da bli kontrollert via script som også er en komponent enten på samme *GameObject* eller hvilket som helst annet med at det er få grenser på hva som er mulig å referere til/fra. Magien skjer når man sier hva som skal skje med spill-objektet via de forskjellige komponentene i script, enten om komponenter skal legges på, lytte etter kollisjon, ta form av et annet grafisk objekt eller bare fjerne spill-objektet fra scenen som er vanlig om objektet er et prosjektil.

Awake(), Start() og onEnable() er de tre hoved metodene hvor vi sier hva slags objekt som skal instansieres, og hva slags forhold de skal ha til hverandre. Disse metodene kan på mange måter erstatte en tradisjonell konstruktør, og gjør dette mulig siden Unity er komponentbasert og trenger sjeldent å bruke parameterliste. Bruken av disse metodene har klare fordeler og ulemper, og uvøren bruk av dem har skapt utrolig mye feil i prosjektet.

Awake() blir kjørt når scenen starter, uavhengig om spill-objektet det ligger på er aktivert eller ikke, så lenge det er i scenen. Her er det vanlig å instansiere alle objekt og fastlegge relasjoner til andre objekt.

Start() blir bare kjørt når spill-objektet scriptet ligger på instansieres. Dette gjør det mulig å kontrollere når metoden kjører siden vi vet at det skjer når vi instansierer spill-objektet scriptet ligger på. Om samme script også har en Awake() metode vil den kjøres ferdig før Start(). I Start() utnytter vi ofte sjansen til å instansiere lokale variabler som spill-objektet av avhengig av. En fallgruve her er ofte å forsøke å bygge opp objekt-relasjoner som fører til «NullPointerExeption», siden Start() ikke er tidsinnstilt til scenestart og derfor ingen garanti for at objektene som kreves er i scenen ved instansiering. Ved bruk av relasjons-kobling i denne metoden bør utvikleren vite den nøyaktige instansiering-rekkefølgen av alle objekt i scenen.

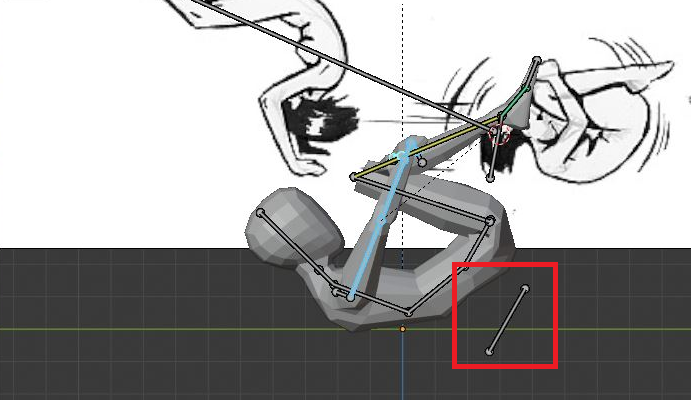
onEnable() kjøres når spill-objektet scriptet ligger på blir satt aktivt. Hvis scriptet også har Awake() og Start() vil disse bli kjørt først. Forskjellen mellom denne og de andre er at onEnable() blir kjørt for hver gang objektet aktiveres mens Awake() og Start() blir bare kjørt én gang for hele scene-tiden. Denne metoden er spesielt egnet for «*Object pools*» der objektene vil aktivere/deaktivere regelmessig, men aldri forlate scenen.

### Teknisk inkompatibilitet

Unity og Blender blir ofte regna som hoved duoen når det kommer til 3D spillutvikling med at begge er gratis og det fins ikke andre gode gratis alternativer til 3D-modellering. Med det sagt, så er ikke dette en tur i parken hvor alt er lagt opp til å fungere slik man vil. Det fins ikke noe sånt som klare veier når det kommer til spillutvikling, alle må brøyte sin egen for å gjennomføre egne ideer. Å kalle Blender og Unity «inkompatible» er ikke langt ifra sannheten når man går djupt nok inn i utviklingen. På et overfladisk nivå med et ensidig spill vil alt man trenger å gjøre være å lage en modell og dra den inn i Unity for et godt resultat. Men når det kommer til innebygd Blender animasjon, farger og modellering kan man lett treffe på hinder som krever en større og helhetlig forståelse av programmene.

Et eksempel på dette er en lang prosess jeg gikk gjennom men som endte opp i en blindvei. Under læring av Blender kom jeg borti mange «*modifiers*» som er noe man bruker på modeller for å modifisere formen eller oppførselen. Mange av de var ferdiglagde animasjonene modifiserte modellen på en måte Unity ikke kunne. Så da tenkte jeg at på samme måte som man manuelt lagde animasjoner ved rigging kunne lett overføres til Unity, kunne jeg også bruke disse, men da tok jeg feil. Innebygde Blender animasjoner krevde å bli lagret i filtypen «*abc*» for å fungere. Samtidig tar ikke Unity «*abc*» filer uten at man først installerer et tillegg som kommer i de nyeste versjonene. Etter en oppgradering av Unity versjon og fungerende animasjon i Unity finner jeg eventuelt ut at «*abc*» filtypen ikke overfører teksturen fra Blender, og modellen kom uten farger. Konklusjonen jeg kom fram til var at jeg måtte begrense Blender til modellering, rigging og simpel tekstur, og heller la Unity ta seg av resten.

Men heller ikke rigging som kommer med modellen inn til Unity ville alltid oppføre seg på den måten jeg ville. Jeg kom til et punkt hvor jeg hadde lagd relativt kompliserte animasjoner som rulling, og i motsetning til andre animasjoner som ble overført til Blender, ville denne vise seg til å være en helt annen animasjon i forhold til det den var I blender. Etter hvert kom jeg fram til svaret som var at blender har ofte problemer med å overføre bein som ikke er direkte i relasjon til andre bein i modellen. For å oppnå en rulleanimasjon måtte jeg brukte andre hjelpe-bein til å rotere modellen, og disse var ikke relatert til beina i modellen.

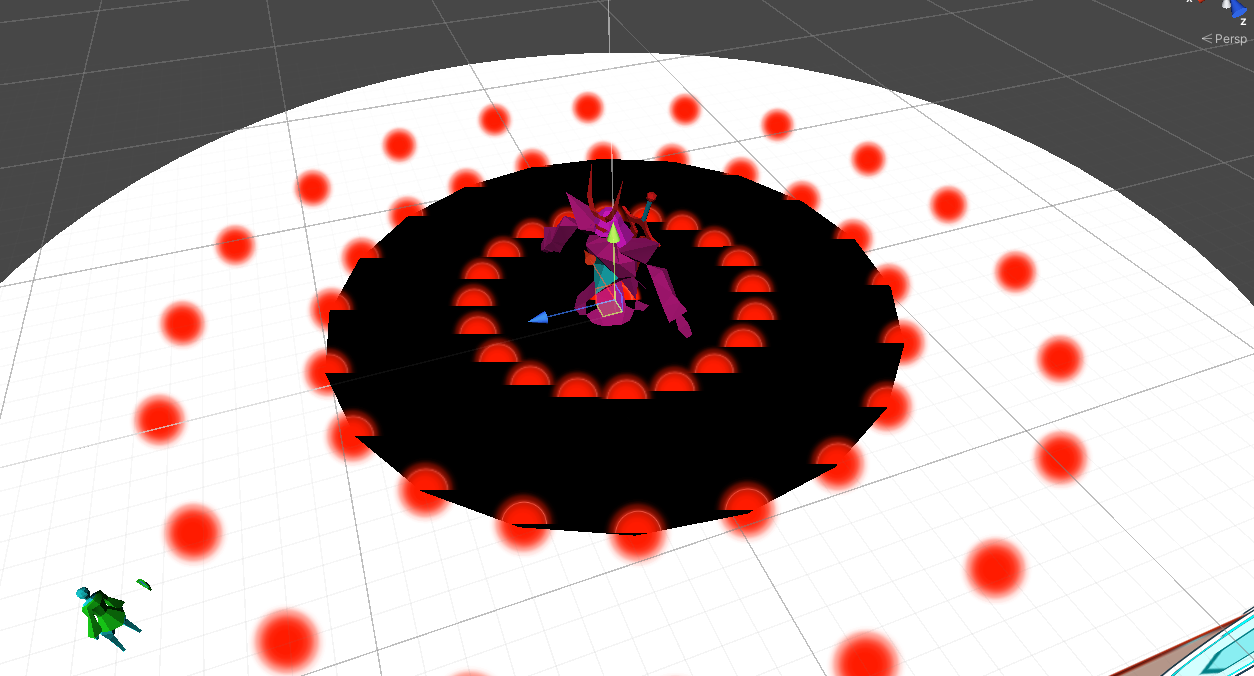


Konseptet rundt rulling og den gamle modellen havnet begge eventuelt i søpla, men jeg tok til meg mye god læring rundt slike hinder og tekniske problem selv om de var frustrerende og tidskrevende.

### Unity hierarkiets ferdigheter

Under utviklinga fant jeg ut at hierarki systemet til Unity er mer enn bare for å vise hva slags objekt som fins i scenen. Hierarkiet er bygd opp av foreldre og barn-objekt, som kan ligne på en objektorientert struktur der barn arver fra foreldre. Hvis vi ser på hva alle spill-objekt i Unity har, så starter alle med komponenten «*Transform*». «*Transform*» består av attributtene posisjon, rotasjon og skala, som er hoved-kontrollene for objekt. Hvis vi lager et barn under et spill-objekt og flytter, skalerer eller roterer foreldreobjektet, vil også barnet følge. Barnet har også en egen lokal instans av «*Transform*» attributtene, så om vi forandrer på disse, vil de ta utgangspunkt foreldre-objektets attributter som basis. Med andre ord, hvis vi flytter foreldreobjektet til x = 200, vil barnets x vise 0 helt til man flytter barnet selv. Naturlig vil ikke barnet påvirke forelderens attributter.

Et godt eksempel på hvordan jeg utnyttet hierarkiet er måten jeg designet et fiende-angrep på. I senere faser starter bossen å sende ut kuler som roterer og sprer seg utover. I utgangspunktet tenkte jeg at jeg måtte gi hver kule et eget script som beskreiv hvordan den skulle oppføre seg, men jeg fant raskt ut at dette ville se unaturlig ut. Derfor kom jeg fram til ideen å bruke en sirkel med et script som sier den skal utvide og rotere seg gradvis. Langs kanten på sirkelen lagde jeg tomme spill-objekt som representerte posisjoner kuler skulle instansieres på, og deretter deaktiverte jeg «*mesh*-*renderer*» komponenten til sirkelen som gjør den usynlig. Resultatet får det til å se ut som de individuelle kulene har et eget liv, men de henger bare på kanten av en usynlig utvidende og spinnende sirkel.



Unity hierarkiet kan også bli brukt til å lette på arbeid og forkorte visse oppsett. Hvis vi har et spiller-objekt som skal bestå av mange komponenter og finner ut at vi må forandre på 3d-modellen og re-importere den til Unity, må vi legge inn alle komponenter og data tilhørende på nytt. Måten vi kan unngå dette på er å konstruere spiller-objektet som en abstrakt klasse der selve modellen er separert fra komponentene, og en re-import vil ikke påvirke andre data enn modellen selv.

### 3D Animasjon

Når det kommer til å animere en 3D-modell, har man generelt to løsninger, statisk animasjon eller «*Root motion*». «*Root motion*» er et nyere konsept som blir brukt i moderne spill mens statisk animasjon starter å bli utdatert. Ved statisk animasjon betyr det at animasjonen blir kjørt uten at nullpunktet til objektet forandrer seg. Det betyr at om objektet har en gå-animasjon, vil animasjonen kjøre uten at objektet nødvendigvis forandrer posisjon i forhold til 3d-rommet. Derfor ved bruk av statisk animasjon, må vi samtidig si at objektet skal bevege seg på en måte som passer animasjonen ved hjelp av script. Ved bruk av «*Root motion*» gir vi all kontroll av bevegelse til animasjonen. Om vi animerer et objekt til å flytte seg en meter framover i blender, vil objektet også bevege seg en meter på samme måte i 3d-rommet i Unity. Dette åpner muligheten for veldig realistiske bevegelse-mekanikker i som er mer tydelige i moderne spill.

I dette prosjektet valgte jeg å bruke statisk animasjon ved at jeg traff på flere problemer ved bruk av «*Root motion*». De fleste av problemene kan generaliseres ned til det at Unity har et eget fysikk-system (Transform), som gjør det vanskelig å holde styr på posisjonen til Unity objektet i forhold til blender-modellen som eier «*Root motion*» delen. Litt nærmere forklart, så har man Unity-objektet som er foreldre til Blender-modellen. Det vil si at manipulasjon av Unity-objektet vil også gjøre det samme med Blender-modellen i forhold til posisjon/rotasjon, men om Blender-modellen har «*Root motion*» vil den selv forandre posisjon/rotasjon uavhengig av Unity-objektet og vil eventuelt skape en forskjell i posisjon/rotasjon mellom objektene. Det er naturligvis mange løsninger rundt dette, men for dette prosjektet følte jeg ikke at «*Root motion*» ga meg noen fordeler.

### Kode-prinsipp under Monobehavior

Selv om prinsipp som *MVC* ikke er engasjert i Unity koding, så har det oppstått andre prinsipp som bygger på den komponent-baserte strukturen av behov for bedre ytelse under kjøring. En av disse prinsippene blir oftest kalt «*Object-pooling*» som er et konsept basert på å bruke de samme objektene om igjen I stedet for å lage nye. Dette er mulig fordi *GameObject* klassen har metoder for å aktivere og deaktivere objekt i scenen, som betyr at vi kan midlertidig kan nærmest slå av ytelsen til et objekt til vi får bruk for det igjen. Dette konseptet er engasjert i spill-programmering siden det å instansiere eller fjerne objekter i kjøretid kan drastisk påvirke ytelsen til et spill og i verste fall påføre «*Lagg*».

Hvis vi tar for oss et spill-konsept som «*Bullethell*» der hundrevis av objekter blir instansiert i scenen der spilleren må unngå de for å ikke ta skade, kan vi se hvorfor dette konseptet kan bli viktig. Instantiate() og Destroy() er tunge operasjoner, og om vi kjører de hundrevis av ganger hver ramme, kan vi risikere og ikke kunne klare å kjøre gjennom Update() metoden 30/60 ganger før sekundet er over.

Løsningen på dette er å bruke Awake() metoden som blir kjørt ved starten av kjøretid før alle andre metoder, uavhengig av om scriptets tilhørende *GameObjekt* er aktivt eller ikke. Her tenker vi oss fram til et maksimalt antall objekter spillet kommer til å bruke og bruker det antallet som kondisjon i en for-loop. Deretter bruker vi Instantiate() for å instansiere objektet og legger på nødvendige komponenter vi vet objektet skal ha om objektet ikke allerede er en «*prefab*» som er et ferdig-innstilt spill-objekt. Til slutt legger vi det instansierte objektet inn i en liste og deaktiverer det. Nå har vi en liste med det maksimale antall objekter vi trenger, og ved behov henter vi ut et og et objekt og aktiverer det. Dette går full sirkel ved at hvert av disse objektene har et script som sier at det skal deaktiveres ved kollisjon eller andre kondisjoner.

Jeg måtte bruke dette konseptet i prosjektet mitt med at jeg bruker et høyt antall objekt i scenen på samme tid i tillegg til en kort livstid. Jeg kom fram til at jeg trengte «*Object-pooling*» ved at jeg merket ytelsen ble dårlig når jeg bare instansierte og ødela objekt etter behov. Unity har funksjonen «Profiler» som kan se hvordan spillet yter under kjøring, som hjalp meg med å se dette.

## Prosessen

### Hvordan ideen og realiteten kolliderte

Da jeg startet hadde jeg en klar visjon over hva jeg ville utføre og klare innenfor tiden jeg hadde, men jeg undervurderte prosessen på utrolig mange felt. Jeg tenkte at så lenge jeg kunne lage en modell i Blender og ha en solid bakgrunn og forståelse av C# at å lage et relativt simpelt spill i Unity ville være lett.

I visjonen hadde jeg sett for meg et spill fra tredje-persons perspektiv der spilleren skulle kampe mot en fiende med forskjellige egenskaper. Jeg vil uten tvil si at det ferdige spillet er et produkt som fortsatt følger den originale ideen, men på mange måter måtte jeg også legge ned flere uskrevne ideer når det kom til kamera, kampsystem, verdenen og fienden på grunn av ren vanskelighetsgrad og tidspress.

For det første fant jeg ut av kamera-koding og få det til å gjøre det jeg ville, viste seg til å være utrolig komplisert og var ikke verdt å investere mer tid i enn jeg allerede hadde gjort. Unity kameraet er basert på tidligere forklart «*Transform*» som er verdier som beskriver et objekts rotasjon, posisjon og størrelse i en 3d-plass. og disse verdiene vises i Unity editoren som x, y og z. Alle «*Transform*» verdier kan manipuleres i kode. Jeg ville at kameraet skulle kunne roteres rundt spiller både på x og z aksen, og koder meg fram til en slik funksjon. Det jeg ikke visste var at de verdiene som vises i editoren ikke beskriver objektets ekte rotasjon i 3d-rommet, bare verdier som er lette for mennesker å forstå. Dette gir problemer om man prøver å direkte manipulere rotasjons-verdiene i editoren med at de ikke representerer de faktiske tallene. Den ekte rotasjonen blir beskrevet gjennom kvaternioner, som Monobehavior har egne metoder for å manipulere. Jeg følte det var et hull som ikke var verdt å gå ned i for en såpass liten del av prosjektet.

Den andre veggen jeg traff relativt tidlig var hvor vanskelig det ville bli å gjennomføre spillerkontrollen på den måten jeg originalt hadde tenkt. I utgangspunktet ville jeg at spilleren skulle kunne bære på et nærkamp-våpen som for eksempel et sverd og kunne slå fienden. Dette forsøkte jeg å gjøre ved å legge en kollisjons boks på våpenet og la en animasjon styre banen til sverdet, noe som viste seg til å være utrolig uresponsivt. Eventuelt fant jeg ut at de fleste spill utviklet i Unity som bruker nærkamp lager system rundt «*Raycasting*» som bruker usynlige linjer i et 3d-rom for å oppdage kollisjon, og gjør det mer robust i motsetning til å basere all kollisjon på om et objekt treffer et annet. Dette kombinert med animasjoner får det til å se ut som objektet spilleren holder er det som treffer. Jeg bestemte meg for å sette dette på pause og starta heller å lete etter andre måter å utvikle kampsystemet på.

### Prosjekt-omstart

Jeg bestemte meg eventuelt for å legge ned det jeg hadde for å starte på blanke ark. Denne gangen forestilte jeg meg et topp-ned perspektiv der kameraet er statisk og bare følger spilleren for å slippe kamera-programmering. Jeg valgte også å begrense spiller-angrep til prosjektiler, selv om jeg originalt hadde sett for meg våpen og angrep for nærkamp. Jeg lagde et system der prosjektil-basert kamp ga mening, og la mer fokus på én unik spill-mekanikk framfor flere.

Dette konseptet åpnet muligheten for en mindre frustrerende utviklings-prosess der jeg fikk bruke tiden min på å kode fram ideer i stedet for å dykke ned i avanserte teknikker. Jeg slapp også å tenke på å utvikle et bevegelses-system som måtte ta hensyn til forskjellige høyder i terrenget siden arenaen består av en flat plate. Jeg startet også på nye modeller for spiller og fiende som hadde mindre kanter (lav polygon modellering). Dette bygger på det å gjøre det lettere for systemet å kjøre, mindre kompleksitet i rigging og kameraet ville uansett være langt nok unna til man ikke ser modellene med nøyaktighet. Det starta raskt å ligne på et spill jeg faktisk kunne gjøre ferdig innen tidsfristen med de nye begrensningene jeg la ut for meg selv.

### Fristelsen av å bruke gratis ferdiglagde modeller

Unity har en butikk der spill-utviklere kan både legge ut og kjøpe forskjellige ferdiglagde «*Assets*» som for eksempel 3d-modeller, effekter og kode. Noe av dette er gratis for utviklere å ta og bruke i egne prosjekt, til og med kommersielle spill som tjenes penger på. Jeg erfarte raskt at mange av 3d-modellene som var lagt ut gratis for bruk i butikken var ganske høy kvalitet på i forhold til det jeg selv kunne lage. Derfor var fristelsen stor når jeg så muligheten for å hoppe over flere utfordringer når det kom til å lage spill-klare modeller som var en stor del av prosjektet. Tar jeg vekk prosessen ved å ikke bare lære Blender(3D-modellering) i prosjektet, men også mestre det til et punkt hvor jeg kan forestille meg noe og lage det, kunne jeg lagt mer fokus på kode og flere mekanikker i spillet. Dette var en realitet jeg måtte vurdere siden det å bruke bare én ferdiglagd modell ville bety at jeg gikk bort ifra den originale visjonen av å lage alle modeller selv.

Jeg endte opp med å lage alt selv som jeg i utgangspunktet hadde bestemt meg for før jeg fant gratis modeller i så høy kvalitet. Dette valget bygger også på ideen om å kunne dokumentere de forskjellige systemene og prosessene som man må gjennom for å lage et spill mye grundigere og riktigere. Jeg tror prosjektet undergraves en god del om jeg velger å ta snarveier når jeg ser dem.

### Begrensninger og tidspress

Jeg kom eventuelt til et punkt der jeg følte at jeg hadde mestret Unity og Blender, og utviklingen gikk unna. Jeg klarte å gjennomføre ting på veldig kort tid uten feil, og spillet lignet mer og mer på noe jeg kunne kalle et bachelorprosjekt. Jeg hadde fått gjennomført de avgjørende elementene som ville definere spillet og måla mine som spillkontroll, animasjoner, fiende fase-system og spill-loop. Jo nærmere slutten jeg kom jo flere gode ideer poppa opp i hodet som diverse «*powerups*» til spiller som jeg nå hadde ferdigheter til å utvikle, men ikke tid. Jeg hadde fortsatt en lang vei foran meg med GUI, bedre modeller og generelt gjøre at spillet ser bra ut. De fleste prosessene i spillutviklinga tok lengre tid enn jeg hadde tenkt i utgangspunktet.

# Systemdokumentasjon

## Spillerkontroll

Grunnlaget til spillerkontrollen er bygd på en «*wasd*» bevegelse der jeg lagrer hvilken verdi x og z aksene har og legger disse inn i en «*Vector3»* variabel. «*Vector3» er et objekt som* representerer x, y og z verdier*.* X og z-verdiene blir funnet ved hjelp av *Monobehavior* sin Input.GetAxisRaw() metode som lytter etter tastatur-trykk. Spilleren har en «*RigidBody*» som er Unity sin fysikk-komponent der jeg kan kontrollere for eksempel kraft i en retning som blir beskrevet av «*Vector3»* variabelen. I utgangspunktet valgte jeg å slå av tyngdekraften siden prosjektet ikke hadde bruk for det, men etter hvert fant jeg ut at det var lurt for å låse spilleren til bakken og hindre mulige kjøretid feil. Derfor bruker jeg også y verdien i «*Vector3»* variabelen til å legge på en kontinuerlig minkende verdi for å representere en tyngdekraft.

Spilleren har to forskjellige modus (angreps-modus og forsvars-modus), og med disse måtte jeg utvikle to ulike bevegelses system. Mens forsvars-modus er veldig standard der spilleren er vendt og løper i samme retning, kodet jeg angreps-modus til at spilleren kan sikte og vende seg i retning til musepeker samtidig som å kunne gå i alle retninger. Her bruker jeg forsvars-modus «*wasd*» bevegelse, men legger på rotasjon av spiller-objektet ved hjelp av *Monobehavior* sin Physics.Raycast() metode. Metoden kaster en usynlig stråle til en plass i 3d-rommet som jeg finner ved å hente x og z-plassen til musepeker. Deretter sier jeg at spiller-objektet skal rotere seg mot den plasseringen. Disse to modusene blir vekslet mellom gjennom lytting etter om spiller holder inne høyre museklikk eller ikke.

Angreps-modus er bygd opp av 9 steg som hver for seg har et eget «*Particle-System*» for en visuell effekt (som er Unity sitt innebygde system for effekter), samtidig som angreps-kraften har en høyere multiplikator for hvert steg. Det betyr at jo lengre spilleren holder seg i angreps-modus, jo raskere vil angreps-kraften øke. Dette balanseres ved at spilleren ikke får nye skjold så lenge angreps-modus er aktivt. Spiller-objekt scriptet har variabler som teller opp og nullstiller seg for hvert steg når de når tidsgrensen jeg har satt, og aktiverer tilhørende «*Particle-System*». Prosjektilet som spilleren fyrer av, tar til seg angreps-kraften og kutter relasjoner til spiller. Dette funker da som et objekt fienden må «pakke ut» og finne float variabelen for kraft i script-komponenten til prosjektilet, om den skulle treffe. *Monobehavior* har egne metoder OnCollisionEnter() og onTriggerEnter() for å høre etter om noe treffer kollisjons-boksen til objektet.

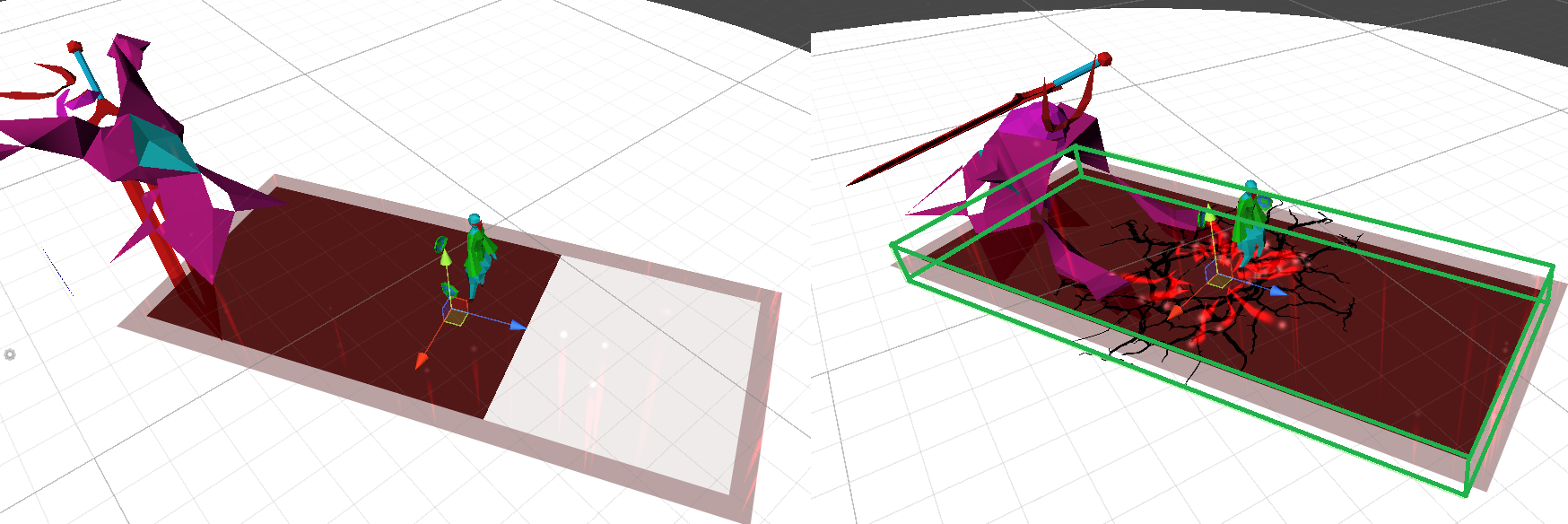
## Fiende AI

Jeg bestemte meg for å programmere fienden til å være så lett som mulig på systemet, så jeg unngikk å bruke metoden Update() som kjører opp til 60 ganger i sekunder. Dette er fordi en fasebasert fiende vil nødvendigvis ikke kreve å bli fortalt hva den må gjøre like ofte som for eksempel en spiller-kontroll som hele tiden må lytte etter taste/museklikk. Det vil heller være smartere å gi den en beskjed om å gjøre en oppgave, så kan den heller opplyse systemet når den er ferdig med oppgaven og få en ny. På den måten bruker ikke objektet mer enn akkurat det den trenger.

Derfor baserte jeg fiende-kontrollen på et system av «*coroutiner*» der fienden baserer angrepsmønster og faser på et system som styrer seg selv i motsetning til en Update() metode og et klyster av if-sjekker. «*Coroutiner*» er metoder som kjøres parallelt med annen kode, i stor likhet med tradisjonelle tråder. I rutinene bruker vi linjen «yield return new WaitForSeconds()» for å fortelle metoden hvor lenge den skal vente på hvilken plass. Under denne ventetiden bruker den betydelig mindre ytelse om man skulle brukt en «while» loop i Update() metoden.

Dette er bygd opp ved at når scenen starter, kjøres PhaseMachine() rutinen i fiende-scriptet som varer helt til scene-slutt med tre sekund pause mellom hver gjennomgang. Her sjekker den hvilken fase fienden er i, og kaller doCoroutine()-rutinen som tar inn en liste med referanser til forskjellige rutiner som hører til fasen. Denne oppfører seg på en måte der den kjører lista med rutiner i en for-loop, men med ventetid og en sjekk som hindrer at for mange rutiner kan kjøre samtidig.

Fiende-angrep er bygd opp på noe jeg vil kalle en veldig «hjemmelagd» måte. Kulene er tidligere forklart, men den andre typen angrep fienden har er av den klassiske sonebaserte «Ikke stå i flammene» der det indikeres på bakken hva/hvor det er farlig å stå. Her har jeg brukt Unity sitt UI- system, som vanligvis blir brukt til å vise knapper og lignende på skjermen statisk, til å plassere bilder på bakken. Dette er gjort mulig med Unity sitt alternativ til å gjøre et UI-objekt om til «*World* *Space*». Inne i disse bildene fyller jeg ett til bilde gradvis opp for å indikere når effekten vil gi utslag å skade spilleren via script. Denne UI-effekten i seg selv er bare visuell, og for å oppdage om den faktisk treffer spiller, har jeg brukt en kollisjons boks. Problemet var at om jeg la en kollisjons boks på effekt-objektet, ville boksen treffe spiller hele tiden uavhengig av når indikatoren viste treff. Derfor har jeg kollisjons boksen deaktivert helt til indikatoren er fylt opp, og dermed aktiverer jeg den og sjekker om spiller står innenfor.



Fiende-programmering ble generelt en prosess der jeg prøvde å implementere forskjellige teknikker jeg ikke allerede hadde vært borti for å lære mest mulig framfor å bygge videre på det originale konseptet der fienden bare har flere variasjoner av «*Bullethell*».

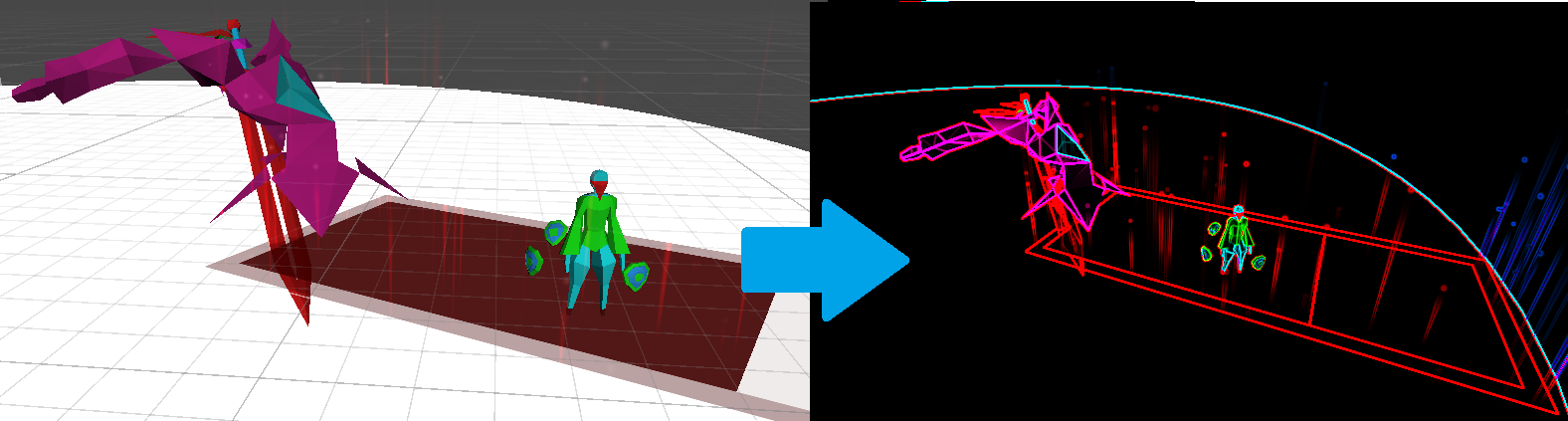
## GUI, resolusjon og grafikk

Unity har et godt innebygd system for oppbygning av GUI, og all GUI er bygd opp av panel inne i andre panel. Alle GUI-objekter har naturligvis også mulighet til å bruke komponenter i likhet med vanlige spill-objekt. Disse komponentene er ofte GUI spesialisert og kan ha fokus på å gjøre den mer interaktiv og dynamisk for ulike skjermstørrelser, men den har også komponenter som knapper, bilder, veksling og nedtrekks funksjoner. Alle slike komponenter har mulighet til å referere for eksempel en onClick() metode i et script, som jeg har tatt i bruk for å styre brukeren rundt. Disse komponentene har jeg også lagt på litt design ved hjelp av Krita.

I menyen har brukeren diverse valg som å stille på resolusjon, grafikk eller gå i vindu modus. Dette ble gjort relativt enkelt ved at disse innstillingene er innebygd i Unity og alt man trenger å gjøre er å kalle på de via refererende metoder. Men det er ikke selvsagt at disse innstillingene er like neste gang brukeren starter spillet, eller scene-skifte i det hele tatt. Derfor måtte jeg ta i bruk «*PlayerPrefs*» som er Unity sitt system for lagring av innstillinger. «*PlayerPrefs*» kan bli brukt til lagring generelt, men dataen er lett å finne og er lett å redigere, så det er ikke alltid like lurt å lagre noe sånt som spiller-framgang om sjangeren skulle være et RPG. «*PlayerPrefs*» blir brukt ved at man lagrer en verdi ved hjelp av en nøkkel(string). Hver gang brukeren forandrer på innstillingene, blir det lagra en ny verdi inn i tilhørende nøkkel, som deretter blir henta på spill/scene oppstart og sjekker om verdien samsvarer med de nåværende innstillingene. I «*PlayerPrefs*» lagrer jeg alle innstillinger i tillegg til score siden spillet bare er lokalt og spilleren konkurrerer bare mot seg selv.

## Bilde-effekt

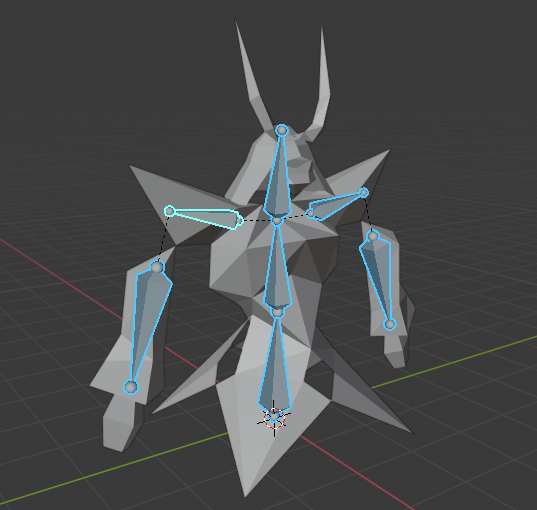
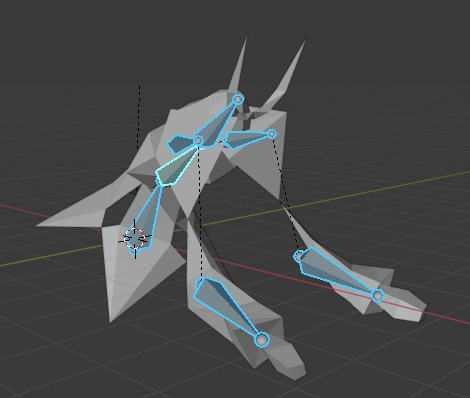
Bilde-effekter er brukt i spill for å gi spillet et annet utseende eller følelse. Dette blir ofte brukt i spill for å gi «*bloom*» eller dybde følelse. Dette blir gjort ved å gi kameraet et script som fungerer som en slags linse som beskriver hvordan den skal oppfatte bildet. Ideen var å lage en bilde-effekt for et geometrisk neon utseende, men jeg innså det å lage en bilde-effekt var et prosjekt i seg selv. Så jeg bestemte meg for å se etter en effekt i Unity butikken, og jeg endte opp med å finne en bilde-effekt fra en tredje-parti utvikler som passet inn perfekt med visjonen jeg ville gå for etter litt justering. Men det var ikke uten å ofre noe. Med at bilde-effekt var den siste prosessen jeg gikk igjennom, var resten av spillet allerede ferdig, og alt jeg hadde bygd hittil fikk et annet utseende. Jeg hadde brukt flere dagers arbeid på å designe alle effektene etter et farge-system som effekten ikke får fram.



Bilde-effekten er også delvis grunnen til at jeg bestemte meg også for å re-designe både spiller og fiende til å ha færre kanter, da effekten legger vekt på kantene i modellen, og for mange kanter så ut som farge-kliss. Effekten førte også til at jeg ga spilleren et alternativ i menyen til å slå av de fleste objektene som utgjør verdenen, med at det kan virke distraherende. Det sier litt om hvor bestemt jeg var i å bruke effekten til og med etter å ha måtte gi opp så mye annet, men grunnlaget i spill-loopen, spiller ferdigheter, fienden og det meste jeg hadde gjort var fortsatt den samme.

## Modellering og animasjon

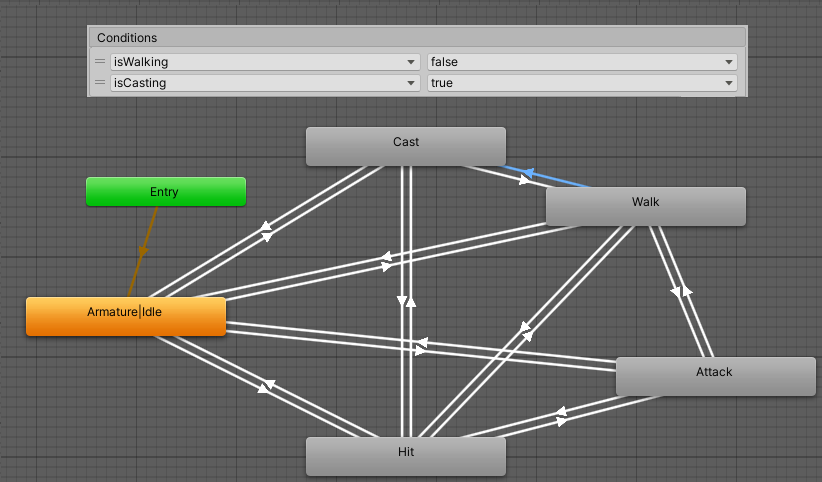
Prosjektet består hovedsakelig av tre modeller, arenaen, spiller og fiende. Arenaen er en statisk modell som bare har et rotasjons-script på seg, mens spiller og fiende modell er mer kompliserte ved at de har et rig med animasjoner på seg.

Dette er en relativ lang prosess i Blender der man designer modellen, bygger opp en bein-struktur til modellen, legger vekt på hvert bein som sier noe hvilke deler av modellen som skal påvirkes av beinet, og til slutt lager animasjoner når man har kontroll over og kan bevege modellen som en dokke. Etter riggen er satt opp kan man eksportere modellen inn til Unity som en «*fbx*» fil, og animasjonene ligger klare for å bindes til eventuelle script. Med at jeg gikk for en lav polygon stil, trengte jeg ikke så mange bein for å kunne kontrollere og animere alle delene av modellen.

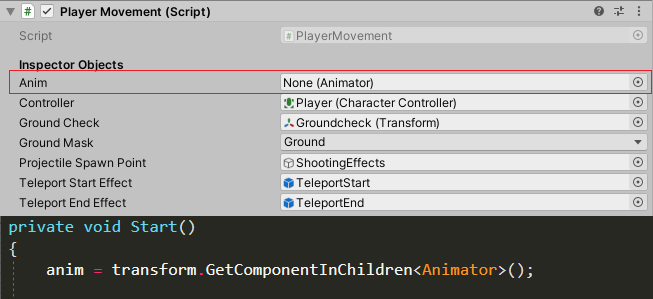
Unity har et node-system for animasjon der man lager kondisjoner for hvilke animasjoner som skal kjøres til hvilken tid. For spiller har jeg brukt «*Blend tree*» der man kan legge inn flere like animasjoner som stå stille og løpe animasjoner med en float variabel. I tilfellet variabelen er 0, vil «*idle*» animasjonen kjøre, og om den er 1 vil «*run*» animasjonen kjøre. Om den derimot er 0.5, vil Unity blande de to animasjonene for å få et godt midtpunkt mellom å stå stille og løpe. Dette er dessverre ikke veldig tilsynelatende ved bruk av tastatur som alltid vil dytte float variabelen til enten 0 eller 1 i løpet av kort tid.

Fienden er mer fokusert rundt avfyrings-animasjon som blir kjørt i samsvar med hvilken rutine fienden er i. Dette er animasjon som ikke trenger flere kondisjons-sjekker, bare en aktivering i et script og noden vil miste fokus når animasjonen er ferdig og gå tilbake til animasjon som stemmer mest i situasjonen. Her har jeg satt opp angreps-animasjonen og ta-skade-animasjonen som avfyrings-noder.



## Komponentbasert oppsett

En stor del av prosjektet er satt opp komponentbasert i Unity editoren. Det betyr at mesteparten av alle spill-objekt-referanser ligger i editoren og ikke i script. Man kan se dette ved at objekt i script er deklarert, men ikke nødvendigvis initialisert. I de aller fleste tilfeller er det mulig å referere spill-objekt fra script, men jeg valgte å gå fullt komponentbasert så jeg kan se visuelt i editoren under kjøretid hva som skjer med objekt-referanser og unngå «*NullPointerExceptions*» på en mye mer oversiktlig måte.



Her er ikke animatøren initialisert i editoren siden jeg gjør det i Start() metoden i scriptet. Animatør-komponenten ligger på et barn av spiller-objektet. Det å si «*transform*» blir det samme som «*this*» vanligvis, siden alle script i utgangspunktet er barn av *Monobehavior*.

Scriptene er lagt opp på en lettvin måte der jeg har «*GameMaster*» som er et statisk script som alle andre script enkelt kan referere for å informere om status. Det har i oppgave å styre spill-status som går ut over å vite om spillet kjører eller ikke, i tillegg til å ha metoder for å navigere scener ved følge av GUI trykk.

# Konklusjon

## Et ambisiøst prosjekt

Da jeg skrev prosjektbeskrivelsen og satt meg selv ambisjonen om å ikke bare lage et 3d spill, men et fullstendig et som jeg ville utgi på en spill-plattform, visste jeg at jeg kanskje satt meg for høye mål. Jeg visste ikke hvor mye tid og innsats de ulike prosessene krevde fra meg når jeg skrev den, og jeg visste heller ikke om jeg kom til å nå målet mitt med et ferdig utgitt spill. Jeg har opp igjennom semestrene alltid tatt sjansen til å lage noe spill lignende der det var mulig, om det var en obligatorisk Java oppgave eller en nettside ved bruk av JavaScript. Det var ofte vi fikk oppgaver i å lage simple spill med «*EasyGraphics*» i tidlige Java fag, og det som virka kjedelig for andre studenter var spennende for meg. Helt siden jeg lærte min første linje med kode var alltid motivasjonen i bakgrunnen til å bli bedre det at jeg kanskje en dag ville få muligheten til å lage et ordentlig spill i skolesammenheng.

## Måloppnåelsen

Målene fra prosjekt beskrivelsen var som følger;

1. Spilleren skal ha fullstendig kontroll over karakteren sin, og kunne bruke diverse våpen og

egenskaper.

2. Flere bosser (AI) skal designes med ulike angreps-mønstre. (Hvor mange kommer an på tid

og kompleksitet)

3. Landskap og ulike andre objekt som våpen og fiender må modelleres.

4. Animasjoner for spiller, fiender og andre objekt.

5. Spillet skal overvåke seg selv. Det betyr at koden skal ha tidsgrenser og transaksjoner

mellom de ulike delene av spillet så ting ikke stopper opp eller går for fort.

6. Et User-Interface for meny og i spill.

7. Lyd-effekter (Bør ha)

8. Musikk (Bør ha)

9. Utgi det ferdige spillet

Selv om jeg måtte begrense de originale måla, føler jeg ikke at det ferdige prosjektet er noe mindre eller dårligere enn det jeg først beskrev. Det var aldri realistisk for en nybegynner og lage et spill av typen «*Sekiro*» på noen måneder, og det jeg endte opp med er unikt og bra på sin egen måte. Det eneste åpenbare jeg måtte tråkke ned på, var punkt 1 som måtte begrenses ned til at spiller ikke kan bruke flere våpen og har bare én egenskap. Men egenskapen er gjennomtenkt, godt utviklet og er hoved-mekanismen i spillet. Jeg fullførte alle hoved-konseptene fra beskrivelsen på en god og robust måte, og spillet ligger utgitt og ferdig på en spill-plattform. Jeg kunne ikke vært mer fornøyd med prosjektet og min egen innsats.

## Erfaringer og utbytte

I utviklingen av spillet tok jeg bruk av *Monobehavior* biblioteket der jeg kunne, noe som naturligvis førte til bruken av mange ukjente metoder. Den eneste åpenbare fordelen jeg hadde med bakgrunn i fire semester med Java og et semester med C# var egentlig bare forståelsen av syntaks, variabler, metoder, klasse-oppbygning og generelle konsept som tilhører alle kode-språk. Prosjektet inkluderte en stor variasjon av forskjellige ting der rå C# koding var en relativt liten del av det. Dette førte til ukesvis med frustrasjon og sitte fast på problemer. Men jo lengre tid noe tar å løse, jo raskere løser man den neste gang som er noe jeg helt klart fikk banka inn under utviklingen. Jeg fikk erfart at det å lage et spill fra bunn av er en vanskelig, tidskrevende og frustrerende prosess, men som gir en utrolig god følelse når man først får det til.

Det er ikke så stor fordel å sitte igjen med et bachelorprosjekt i spillutvikling da det ikke bygger på så mange prinsipp og fag som jeg har hatt opp gjennom disse tre åra. Jeg fikk naturligvis heller ikke erfaring med prosjektarbeid som gruppe og bruk av sprinter og fysisk samarbeid. Men den konklusjonen kom jeg fram til før jeg starta og det var fortsatt ikke tvil om hva jeg ville gjøre. Dette prosjektet har lært meg Blender som ikke har noe med IT å gjøre, men jeg kjenner på et større utbytte av det enn mange andre fag jeg har tatt. Jeg kan modellere, rigge, animere og fargelegge modeller, noe jeg allerede klør i fingra etter å ta i bruk i mitt neste spill-prosjekt som blir en hobby prosjekt fremfor et skole-prosjekt uten en tidsgrense.

Prosjektet har også gitt meg en grundig gjennomgang av Unity der jeg lærte det grunnleggende ved oppsett av spill-objekt og komponenter, men også interne Unity system som «*Particle System*» som jeg tok i bruk utrolig mye, brukergrensesnitt, lyseffekter og scripting med c#. Jeg har fått repetert C# selv om jeg ikke nødvendigvis har tatt i bruk tradisjonelle C# prinsipp som for eksempel forenklet *«get/set»* -metoder i forhold til Java som fortsatt sitter igjen i kjernen min etter 1 år. Det å koble C# sammen med *Monobehavior* har vært veldig interessant da jeg måtte skifte tankegangen min fra tradisjonell Java programmering der referanser går gjennom konstruktører og parametere, til en ren visuell referanse. I de få tilfellene jeg måtte referere en komponent manuelt i et script, endte jeg opp med lange kjeder av referanser gjennom forskjellige spillobjekt som hjalp meg med å forstå hvordan komponentsystemet var bygd opp. Så selv om prosjektet ikke var så relevant til faget som jeg skulle ønske det var, setter jeg pris på de erfaringene jeg fikk av det, og spillutvikling er uten tvil en hobby jeg kommer til å fortsette med.

## Format

Jeg hadde originalt tenkt at spillet skulle designes for Android siden formatet rundt en «*Boss-Battle*» med ideen om å overleve så lenge som mulig passer bedre som et mobilspill. Med det i tankene, designet jeg spillet slik at modellene hadde et lavt antall kanter og utnyttet «*Object-pools*» og optimalisert kode for bedre ytelse. Det jeg ikke gjorde, var å designe spillet med tanke på begrensningene mobil har til kontroll. Et mobilspill blir ofte spilt med to tomler, mens jeg hadde designet det på en måte hvor noen situasjoner trengte tre. Uten å tilbake-utvikle konseptet visste jeg ikke hvordan jeg ville få det til å bli en god spillkontroll, så jeg bestemte meg for å utvikle det ferdig som et pc-spill.

## Utgivelse og publisitet

Planen var å legge ut spillet på den største plattformen for spill, Steam, men etter å ha starta prosessen fant jeg ut at dette var en lang prosess som både ville ta lang tid og mye arbeid. Steam er en plattform som krever mer enn at man bare laster opp en fil. Det er en prosess som innebærer å først betale for å publisere, få godkjenning for lovlig innhold, og legge til prestasjoner i spillet for å støtte systemet de har med nivå og merker. Derfor fant jeg en annen plattform ved navn Itch.io der mange «*indie*» spill-utviklere holder til. Denne var mye mer lettvin ved at man trengte bare registrering og en filopplastning av spillet.

Ved utgivelse lagde jeg en tråd i Unity 3d samfunnet på Reddit og reklamerte for spillet i håp om å få noen til å prøve det ut og gi tilbakemeldinger.

# Referanser

GitHub prosjektet:

<https://github.com/JavaBeginner66/Bachelor-project>

Reddit bruker:

<https://www.reddit.com/user/Java_beginner66>

Dokumentasjon av kode er nærmere gjort i script (på engelsk): <https://github.com/JavaBeginner66/Bachelor-project/tree/master/Assets/Scripts>

Link til det utgitte spillet:

Reddit tråd for publisitet

# Kilder

Kildebruk gjennom prosjektet har gått ut på å se hvordan andre gjør ting, og prøve å forandre på det så det passer min egen kontekst. Scriptene inneholder ikke direkte referanser eller kopiert kode, bare inspirasjon og hjelp til å få det til å funke inn i prosjekt-sammenhengen.

Unity documentasjon:

<https://docs.unity3d.com/Manual/index.html>

Unity questions/answers

<https://answers.unity.com/index.html>

Unity3d reddit

<https://www.reddit.com/r/Unity3D/>

Blender documentasjon:

[https://docs.blender.org/manual/en/latest/#](https://docs.blender.org/manual/en/latest/)

Blender reddit

<https://www.reddit.com/r/blender/>

Youtube-kanaler som stod for startfasen av læring av Unity:

<https://www.youtube.com/user/Brackeys>

<https://www.youtube.com/channel/UC9Z1XWw1kmnvOOFsj6Bzy2g>

Blender-serie som hjalp meg med det grunnleggende

<https://www.youtube.com/watch?v=MF1qEhBSfq4&list=PLa1F2ddGya_-UvuAqHAksYnB0qL9yWDO6>

Blender-serie som hjalp meg med prosessen rundt modellering, rigging og animasjon

<https://www.youtube.com/watch?v=aAO4C_8y0w8>

Bilde effekt pakke fra Asset Store

<https://assetstore.unity.com/packages/vfx/shaders/fullscreen-camera-effects/snapshot-shader-collection-146666>