

**Prosjektrapport**

3D Spillutvikling med Unity

Utviklings-rapport for spillprosjektet «Pepigo»



Utført av Jan Andreas Sletta

# Introduksjon

## Forord

## 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. På mange måter vil jeg kalle dette prosjektet en lang improvisasjon.

## Oppsett

Når det kommer til rapport og dokumentasjon vil jeg legge mer fokus på den faglige beskrivelsen som utviklingsmetoder, dokumentasjon, design og forklaringer av ulike systemer i motsetning til administrative beskrivelse 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. Dette kommer jeg tilbake til i utviklings delen. Samtidig som prosjektet er utviklings-fokusert, er jeg like interessert i å gjøre det forsknings-basert der jeg går grundig inn i de ulike systemene og forklarer hvordan ting funker, hvor fallgruvene ligger og hvordan prosessen var.

Jeg tror at det å bare lage et produkt (et spill) bør ha en større representasjon av prosessen i motsetning til et produkt som for eksempel en nettside som kan snakke for seg selv i IT-mennesker sine øyne. I motsetning vil ikke hvem som helst klare å se hvilken innsats er lagt inn i det å lage et spill. Det er en intern prosess som ikke kan relateres til en større folkegruppe.

# Administrasjon (?)

# Utviklingsmetode

# Utviklingen

## Teknisk

Under utviklingen var jeg innom

Forklare konseptene før jeg tar dem til bruk i utviklingsmetode

### Sammenkobling mellom Unity og C#

Måten C# blir implementert 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, input events, objekter i scenen og kontroll over hver ramme.

«GameObject» er base-komponenten i Unity som kan representere alt fra spiller-karakterer til et kamera, som enten ligger i scenen før runtime 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. 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. Det at 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. 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 implementere 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 animasjoner som 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, 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 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.

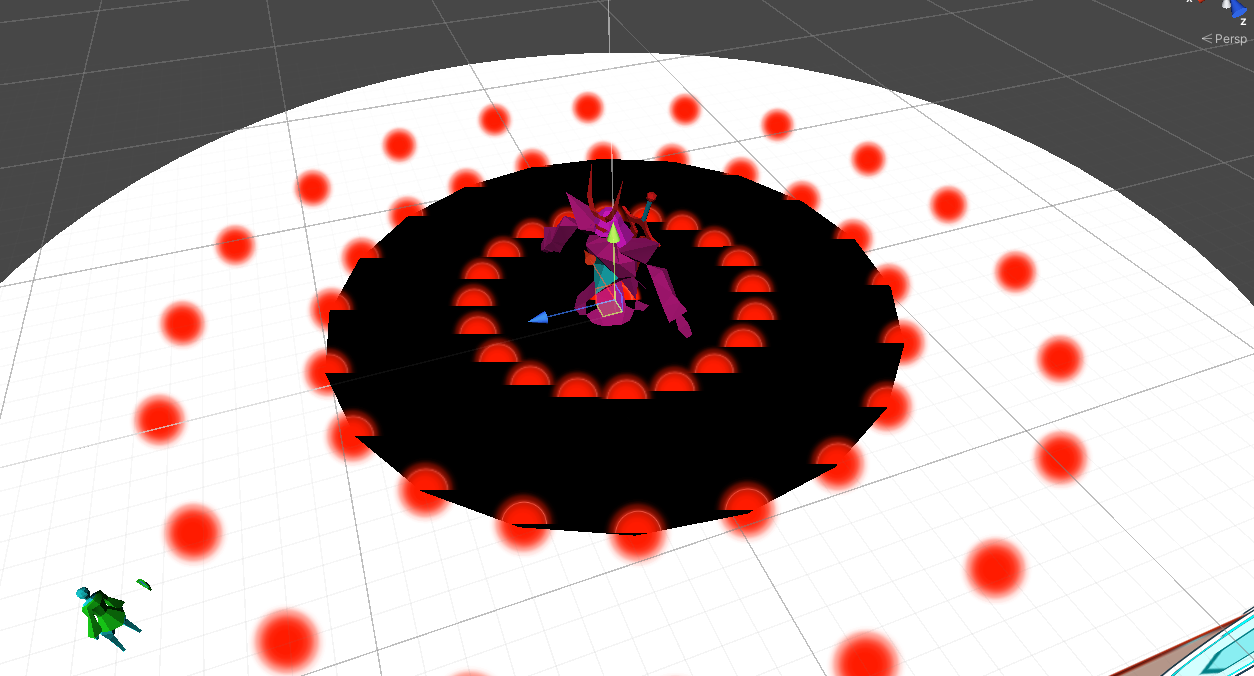
Men heller ikke rigging som kommer med modellen inn til Unity ville alltid oppføre seg på den måten jeg ville. Jeg kom til et punkt hvor jeg hadde lagd relativt kompliserte animasjoner som rulling, og i motsetning til andre animasjoner som ble overført til Blender, ville denne vise seg til å være en helt annen animasjon i forhold til det den var I blender. Dette er delvis fordi Unity og Blender har ulike standarder eller nullpunkt for rotasjoner?????

----------------------------------------------------------------------------------------------------

### 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 setup 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 renderen 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. Derfor ved bruk av statisk animasjon, må vi samtidig si at objektet skal bevege seg på en måte som passer animasjonen ved 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 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 parent 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. Dette blir ofte 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 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 overleve, 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 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. Her traff jeg også på den største feilen i prosjektet som jeg lot ligge i ukesvis før jeg endelig hadde forståelsen rundt Unity og Monobehavior til å løse det.

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.

## 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 tiden det ville ta.

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 beskriver ikke objektets ekte rotasjon i 3d rommet, bare verdier som er lette for oss 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 mere 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 jeg og starte på blanke ark. Denne gangen forestilte jeg meg et topp ned perspektiv der kameraet er statisk og bare følger spilleren. 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 rund plate. 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 «Assets»

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 og andre material. 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 alt selv. Jeg endte opp med å lage alt selv som jeg i utgangspunktet hadde bestemt meg for før jeg fant gratis material 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.

### Midlertidig mestringsfølelse

Jeg kom eventuelt til et punkt der jeg følte alt 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.

### Begrensninger og tidspress

# 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. 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 increasesiden 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 nedover kraft.

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 inebygde 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 sjold 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.

## 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 har ofte fokus på å gjøre den mer interaktiv og dynamisk for ulike skjermstørrelser, men den har også komponenter som knapper, bilder, toggle 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 Photoshop.

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.

## 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 en 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.

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.

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 blend treet for bevegelse. 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.

Scriptene er lagt opp på en lettvin måte der jeg har GameMaster som er et statisk script som alle andre script enkelt kan referere det for å informere om status. Det står for 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

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 ét semester med c# var egentlig bare forståelsen av syntaks, variabler, metoder 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.

# Referanser