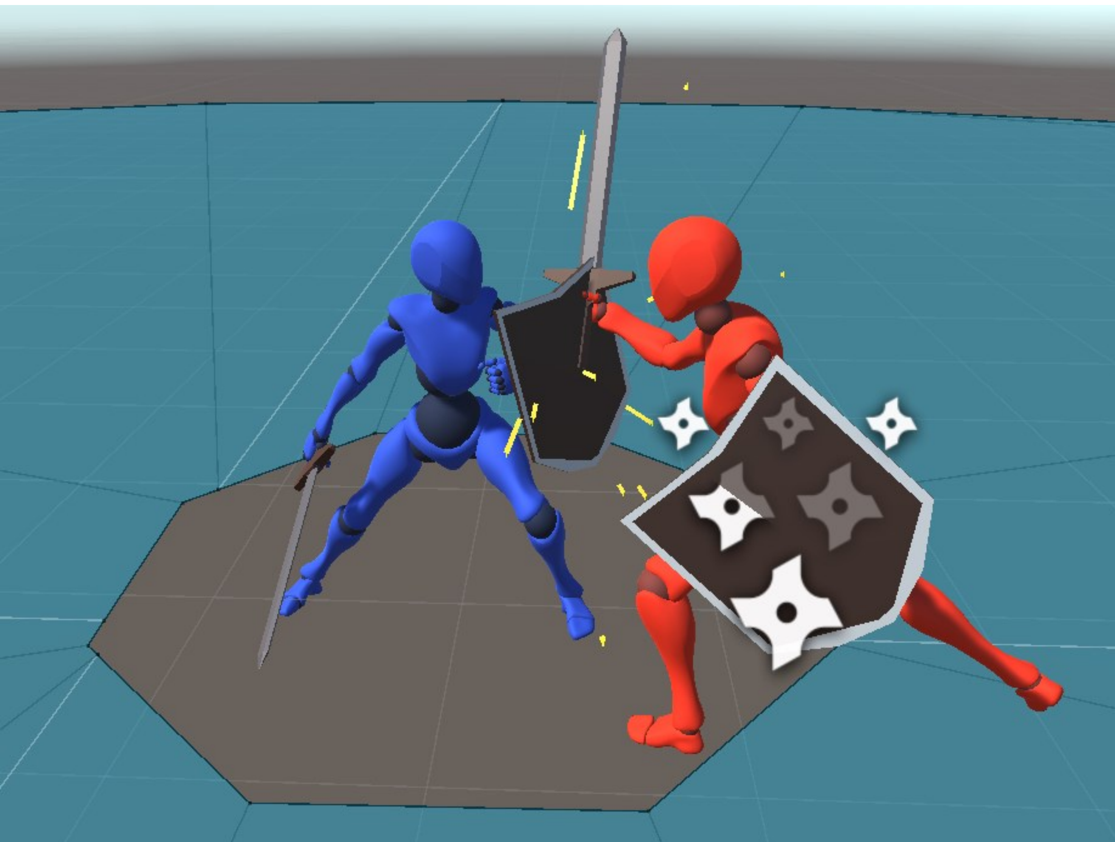
Ojanperä Lauri

Vihollisen tekoälytilojen ja animaatioiden synkronointi Unityssä



Tradenomi

Tietojenkäsittely

Syksy 2025

Tiivistelmä

Tekijä(t): Ojanperä Lauri

Työn nimi: Vihollisen tekoälytilojen ja animaatioiden synkronointi Unityssä

Tutkintonimike: Tradenomi (AMK), tietojenkäsittely

Asiasanat: Ohjelmointi, Peliala, Animaatio, Unity, Synkronointi, Tekoäly

Tässä opinnäytetyössä kehitettiin kolmannen persoonan kuvakulmasta oleva keskiaikainen taistelupeli ja keskitytiin vihollisen tekoälytilojen sekä animaatioiden synkronointiin. Hyvin tehty synkronointi animaatioiden sekä tekoälyn välillä on tärkeää, sillä muuten peli tuntuu epäreilulta tai näytöllä tapahtuva liike ei vastaa täysin pelaajan syötettä.

Opinnäytetyön päätavoitteena oli tutkia tekoälytilojen sekä animaatioiden synkronointia peleissä sekä toteuttaa peli, jossa voisi tutkia asiaa käytännössä. Pelissä vihollisen tekoäly, pelaajan syöte sekä animaatiot toimivat loogisesti sekä ennustettavasti. Löytyneitä synkronointiongelmia, kuten ajoitusongelmia, animaatioiden keskeyttämistä, tilakoneen sekoamista sekä häivytyssiirtymiä tutkittiin sekä testattiin käytännössä.

Projekti tehtiin käyttämällä Unity -pelimoottoria, animaatiot haettiin Mixamo-verkkosivulta sekä joitain 3D-malleja tehtiin Blender-ohjelmalla. Animaatioita ohjattiin käytännössä täysin ohjelmoinnin kautta Unityn visuaalisen tilakoneen sijasta. Tämä salli suuremman kontrollin animaatioiden toistolle, sekä helpotti virheenetsintää suuresti.

Tulokseksi tuli projekti, jossa on toimiva sekä reagoiva taistelumekaniikka. Animaatiot toistuvat silloin kuin niiden pitää, ja ovat synkronoituna hahmojen liikkeisiin. Opinnäytetyön tulosten valossa voidaan todeta, että koodipohjainen animaatiotilojen hallinta voi helpottaa niiden integroimista peliin. Virheenetsintä myös helpottuu, varsinkin jos projekti kasvaa suureksi.

Abstract

Author(s): Ojanperä Lauri

Title of the Publication: Synchronizing Enemy AI States and Animations in Unity

Degree Title: Bachelor of Business Administration, Business Information Technology

Keywords: Programming, Game Industry, Animation, Unity, Synchronizing, Artificial Intelligence

In this thesis, a third-person medieval fighting game was developed, focusing on synchronizing enemy AI states and animations. A well-made synchronization between animations and AI states is important, otherwise the game feels unfair, or the events on the screen do not fully correspond to the player input.

The main objective of this thesis was to research the synchronization of AI states and animations and develop a game in which this topic could be examined in practice. In the game enemy AI, player input and animations work logically and predictably. Found problems with synchronization, such as timing issues, animation interruption, state machine malfunctions, and blending transitions were studied and tested in practice.

The project was made using the Unity game engine, animations were downloaded from the Mixamo-website, and some 3d models were made in Blender. Animations are practically entirely controlled through programming, rather than Unity's visual state machine. This allowed for greater control over animation playback and significantly eased debugging.

The result was a project featuring a functional and responsive combat system. Animations play when they are supposed to and are synchronized with the characters' movements. Based on the results of the thesis, it can be concluded that code-based management of animation states can make it easier to develop a game. Furthermore, bug fixing relevant problems become easier, especially if the project grows large.

Sisällys

1 Johdanto 1

2 Animaatioista ja animoinnista 2

2.1 Animaatioiden historiaa videopeleissä 2

2.2 Animaation hallinta 3

2.2.1 Mecanim 4

2.2.2 Animaatioiden ohjaaminen ohjelmoinnin kautta 5

2.2.3 Valinta näiden kahden välillä 6

3 Ohjelmoinnin ja animaatioiden synkronointi 7

3.1 Tilakone ..... **Virhe. Kirjanmerkkiä ei ole määritetty.**

3.2 Ohjelmoinnin ja animaation synkronointi ..... **Virhe. Kirjanmerkkiä ei ole määritetty.**

3.2.1 Ajoitusongelmat 7

3.2.2 Hahmon liikuttaminen animaatiolla ja ohjelmoinnilla 7

3.2.3 Tilakoneen sekoaminen 8

3.2.4 Animaatioiden häivytysongelmat 9

3.2.5 Animaatioiden keskeyttämisongelmat 9

3.2.6 Pelaajan syötteen ajoitus sekä puskurointi 10

4 Projektin tekeminen 11

4.1 Aloitus 11

4.2 Pelaajahahmon liikkuminen 12

4.3 Pelaajan kamera 12

4.4 Animaatiot 12

4.5 Hitboxit 15

4.6 Particle System 16

4.7 Sivuttain kävely 17

4.8 Inverse Kinematics 19

5 Yhteenveto 23

Lähteet 24

Symboliluettelo

Cutscene	Pelissä esiintyvä kohtaus, jossa pelaaja voi liikkua joko hieman tai ei ollenkaan. Käytetään usein tarinankerronnassa.
Frame	Yksi kuva animaatiosarjassa. Näitä nopeasti toistamalla saadaan aikaan liikettä.
Tilakone	Järjestelmä, joka vaihtaa tilojen välillä sille asetettujen sääntöjen ja syötteen perusteella.
Unity	Pelimoottori, jolla voi rakentaa pelejä sekä sovelluksia.
Mecanim	Unityn animaatiojärjestelmä, joka hallitsee animaatioita.
Hitbox	Näkymätön alue pelissä, joka tarkistaa, osuvatko objektit toisiinsa.
Objekti	Elementti pelissä, kuten esine tai hahmo.
Tekoälyhahmo	Hahmo pelissä, joka suorittaa sille ohjelmoinnissa määriteltyjä toimintoja.
Particle System	Unityn komponentti, jolla voi luoda partikkeleihin perustuvia efektejä.
Boolean	Muuttuja, jolla on vain kaksi tilaa: tosi tai epätosi.
Inverse Kinematics	Tekniikka, jolla lasketaan hahmon nivelten kulma niin, että haluttu kehonosa saavuttaa halutun paikan.
Rig	Hahmon luuranko, joka ohjaa raajojen ja nivelten asentoa.

1 Johdanto

Opinnäytetyön aiheena on rakentaa peli, jossa tekoälyvihollinen ja pelaaja taistelevat toisiaan vastaan ja synkronoida animaatiot mahdollisimman hyvin koodin kanssa. Aihe on tärkeä, sillä se vaikuttaa suoraan pelikokemukseen. Puutteellisesti synkronoidut pelit turhauttavat pelaajaa, ja hyvin synkronoidut vuorostaan parantavat pelikokemusta.

Peli on keskiaikainen kolmannen persoonan taistelupeli, jossa pelaaja sekä tekoälyvihollinen taistelevat toisiaan vastaan. Erityisesti keskityn animaatioiden ja pelilogiikan synkronointiin. Pelaaja voi esimerkiksi lyödä vihollista, torjua iskuja, väistää niitä tai ottaa osumia, ja kaikki tähän liittyvät animaatiot toistuvat sulavasti ja synkronoituna pelin logiikan kanssa.

Opinnäytetyössäni dokumentoin pelin tekemisen eri vaiheita, löydettyjä ratkaisuja sekä vastaantulleita ongelmia. Työ toteutetaan *Unity*-pelimoottorilla. [1.] Valitsin tämän, koska olen käyttänyt sitä eniten projekteissa tähän asti ja se on tutuin. Tarvitsemani animaatiot hankin *Mixamo*-sivustolta. [2.] Valitsin tämän, koska aikaisemmissa projekteissa se on osoittautunut toimivaksi ja käteväksi ratkaisuksi. Tarvitseminani malleina käytän Mixamon animaatioiden mukana tulleita, tai mallinnan itse omani *Blender*-ohjelmalla [3.].

2 Animaatioista ja animoinnista

Tässä luvussa esitellään keskeisiä käsitteitä opinnäytetyön aiheeseen liittyen. Käydään läpi animaatioiden hallintaa, tekoälyä peleissä sekä muita asiaan liittyviä käsitteitä.

2.1 Animaatioiden historiaa videopeleissä

Grafiikka, ja jatkeena animaatiot, ovat nykyään erittäin tärkeä osa videopelejä. Tyylejä on monia, mutta jokaisessa pelissä on grafiikkaa jonkin verran. Animaatiot ovat hieman eri asia, ne voivat olla erittäin isossa osassa videopeliä, kuten esimerkkinä *Red Dead Redemption 2* [4.], jossa cutscenet ja videopeli siirtyvät toistensa välillä saumattomasti sekä animaatioiden laatu on muutenkin erinomaista. *Red Dead Redemption 2* sisältää cutsceneja noin 20 tunnin edestä. [5.] Vastapuolen esimerkkinä voitaisiin ottaa *RimWorld* [6.], jossa animaatioiden määrä on erittäin vähäinen. Tynan Sylvesterin mukaan, *”To me, Rimworld graphics are as important as typeface in a novel”* [7, 13:50] Animaatioita korvataan efekteillä sekä ohjelmoinnilla. Tämä on toisaalta artistinen valinta, mutta se myös helpottaa pelin optimointia ja pienentää kokoa.

1950-luvun lopulla, kun ensimmäiset videopelit tulivat julkisuuteen, olivat ne erittäin yksinkertaisia. Esimerkkinä *Tennis For Two* [8.] Toki tuolloin yksinkertaisuus oli puhtaasti laitteiston rajoitusten seurausta. Laitteiston kehitys jatkui, ja 1972 julkaistiin legendaarinen *Pong* [9.] Pelit pysyivät suhteellisen yksinkertaisina, kunnes teknologia oli kehittynyt tarpeeksi.

1980-luvulla videopelit alkoivat sisältää animaatioita. Ensimmäisten joukossa oli *Donkey Kong* [10.], jossa oli sprite-animaatioita [11.] sekä loppuvuosikymmenellä tullut *Prince Of Persia* [12.], jossa animaatiot olivat rotoskoopattuja [13.] 1990-luvulla 3D-teknologia oli kehittynyt huomattavasti, ja vuosikymmenen lopulla julkaistiin *Super Mario 64* [14.], jossa liikkuminen sekä animaatiot toistuivat sulavasti 3D-ympäristössä [15.]

Vuosituhannen vaihteen jälkeen videopelilaitteiden teho on kasvanut, minkä seurauksena videopelien laatu on parantunut huomattavasti. Nykyään animaatioita voidaan jopa luoda simuloinnin kautta tai hyödyntämällä liikkeenkaappausteknologiaa, jolla näyttelijöiden liikkeet saadaan tallennettua pelissä käytettäväksi animaatioksi. Näin voidaan havaita, että animaatioiden käyttö

sekä laatu ovat lisääntyneet suuresti videopeleissä, mikä on aiheuttanut myös ongelmia, joita ei yksinkertaisimmissa peleissä ollut.

2.2 Animaation hallinta

Animaatioita täytyy hallita peleissä, jotta ne toistuvat oikeaan aikaan ja niin kuin ne on tehty toistumaan. Kun pelejä tehtiin niiden alkuaikoina, niiden graafinen esittäminen sekä ohjelmoinnillinen toiminta olivat yksinkertaisimpia. Nykyaikana, varsinkin 3D-peleissä, näiden kummankin monimutkaisuus on noussut suuresti. [16, s21]. Hahmot pelissä tekevät monimutkaisia liikkeitä ja toimintoja, ja nämä täytyy synkronoida hyvin, jotta peli näyttää ja tuntuu hyvältä pelata.

2.2.1 Tilakone

Tilakone on ohjelmointimalli, jossa ohjelmointi toistaa vain yhtä tilaa kerrallaan. Tilakone siirtyy tilasta toiseen, kun jokin ehto täyttyy. Konkreettisena esimerkkinä voidaan ottaa tekoälyhahmo, joka vartioi tiettyä aluetta. Hahmo on niin kauan lukittuna vartiointitilaan, kunnes jokin ehto muuttaa sen pois siitä. Jos hahmo kuulee pelaajan, siirtyy se etsintätilaan. Jos hahmo näkee pelaajan, siirtyy se hyökkäystilaan. Jos tekoälyhahmo hukkaa pelaajan, palaa hän vartiointitilaan.

Mecanimin visuaalinen tilakone eroaa tästä klassisesta mallista siten, että siinä voidaan käyttää kerroksittaista animaatioiden toistamista. Esimerkiksi hahmon ylä- ja alaruumista voidaan hallita erikseen, joten samalla kun alaruumis toistaa kävelyanimaatioita, yläruumis voi vaihtaa aseen hallinta-animaatioiden välillä. [17, s22-23.]

Ylä- ja alaruumiin kerroksittaista animaatiota voidaan myös ohjata ohjelmoinnilla. Tämä ei yleensä ole kustannustehokasta, ellei pelissä ole asioita, jotka vaativat tarkempaa kontrollia, kuten esimerkiksi deterministisiä elementtejä tai uniikin muotoisia hahmoja. Esimerkiksi ihmishahmoa käyttäessä Mecanim tarjoaa yksinkertaisen tavan eritellä ylä- ja alaruumiin animaatiot.

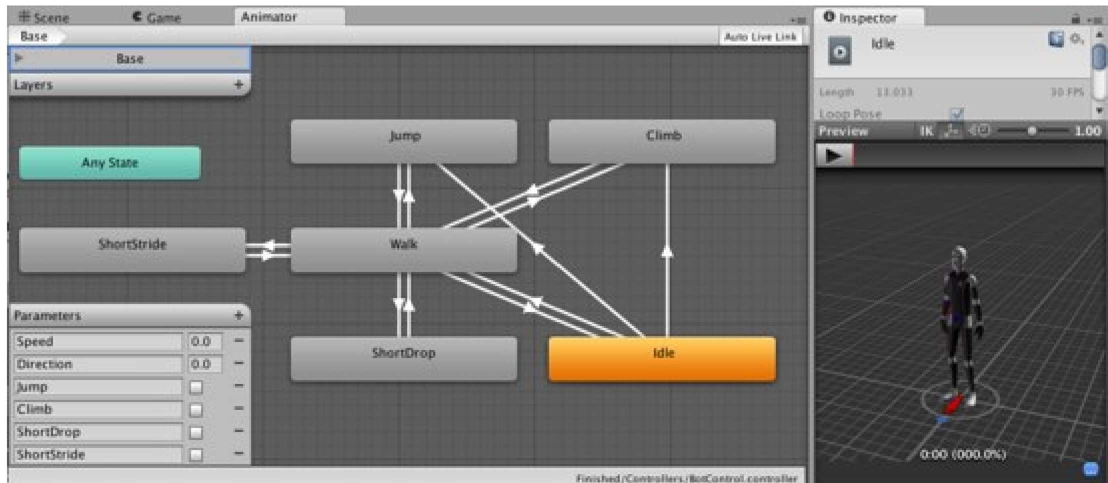
2.2.2 Mecanim

Unityssä on oma hallintatyökalu animaatioille, Mecanim [18.] Mecanim on Unityn kokonaisvaltainen ja rikas animaatiojärjestelmä. Mecanimissä on visuaalinen tilakone, mutta sillä on eroavaisuuksia klassisen tila-automaatin kanssa. Animaatiota voi myös hallita ohjelmoinnin kautta.

Mecanim tarjoaa:

- Helpon työskentelypohjan sekä animaatioiden asettamisen ihmismäisille hahmoille.
- Animaatioiden siirtämisen (retargetoinnin), millä voi siirtää animaatioita hahmomallien välillä.
- Yksinkertaisen prosessin animaatioiden toistamista varten.
- Kätevän esikatselutyökalun animaatioihin, siirtymisiin sekä niiden vuorovaikutukseen. Tämä helpottaa animaatioartistien työskentelyä, koska he eivät ole niin riippuvaisia ohjelmoijista.
- Eri ruumiinosien animoinnin erilaisilla logiikoilla.
- Monimutkaisten interaktioiden hallinnan visuaalisella työkalulla.

Mecanim pystyy paljoon. Animaatioiden hallinnasta tässä kontekstissa puhuttaessa keskitytään pelkästään viimeiseen osaan, eli visuaaliseen työkaluun. [Kuva 1.]



Kuva 1. Mecanimin visuaalinen ohjelmointityökalu vasemmalla sekä animaatioiden esikatseluikkuna oikealla.

2.2.3 Animaatioiden ohjaaminen ohjelmoinnin kautta

Unityä käyttäessä animaatiot täytyy kuitenkin laittaa Mecanimin tarjoamaan Animator -ikkunaan, käytetään Mecanimin visuaalista työkalua animaatioiden ohjaamiseen tai ei. Animaatioiden siirtymää ja toimintaa ohjataan ohjelmoinnin kautta. [Kuva 2.]

```
10 references
void ChangeAnimationState(string newState) {

    // stop the animation from going into loop
    if (currentState == newState)  return;

    else {
      _animator.CrossFadeInFixedTime(newState, 0.25f);
      // set the current state
      currentState = newState;
    }
}
```

Kuva 2. Esimerkki yksinkertaisesta ohjelmointipätkästä, jolla voidaan vaihtaa animaatiota.

Kun animaatioita ohjataan itse ohjelmoinnin kautta, löydetään useita hyviä puolia. Esimerkiksi animaatioita on joustavampaa hallita, koska useita parametrejä voi hienosäätää, mikä ei onnistu tai on hankalampaa Mecanimin visuaalisella työkalulla.

Jos animaatioiden määrä projektissa kasvaa suureksi, on niiden hallinta visuaalisella työkalulla vaikeaa. Animaatioiden ohjaaminen ohjelmoinnin kautta on näin ollen paremmin skaalautuvaa suuriin projekteihin.

Tärkeänä osana on virheiden korjaus. Tämä on huomattavasti helpompaa ohjelmoinnin kautta kuin visuaalisessa ympäristössä. Virheiden etsintä sekä korjaus on ohjelmoijien perustyötä, joten se onnistuu sujuvasti koodiympäristössä.

2.2.4 Valinta näiden kahden välillä

Mecanimin tarjoama visuaalinen tilakone on hyvä työkalu animaatioiden ohjaamiseen, jos animaatioiden määrä pysyy pienenä, tekijä on riittävän perehtynyt työkaluun ja on hyvä visuaalisessa ohjelmoinnissa. Jos tekijällä on yhtään taustaa ohjelmoinnissa, suosittelisin suuresti ohjaamaan animaatioita ohjelmoinnin kautta. Projekti on skaalautuvampi ja helpompi ylläpitää.

Kokeiltuani molempia vaihtoehtoja suoritan projektin jälkimmäisellä, eli ohjaan animaatioiden toistumista itse ohjelmoinnin kautta. Muutkin ovat huomanneet tämän olevan parempi vaihtoehto. [19, s8]: *”A well-designed state machine enhances modularity, maintainability, and scalability of the game's codebase, facilitating easier updates and feature additions”.*

3 Ohjelmoinnin ja animaatioiden synkronointi

Animaatio sekä ohjelmointi täytyvät olla synkronoituna, muuten pelikokemus heikkenee. Esimerkiksi jos pelaaja lyö tekoälyvihollista, mutta vihollinen ei ota ollenkaan vahinkoa tai sen ottaminen viivästyy. Tai jos vihollinen lyö pelaajaa ja pelaaja yrittää torjua sen oikealla hetkellä, mutta ohjelmoinnissa tehtävä hyökkäys tapahtuukin pienellä viiveellä animaatiosta tehden torjumisesta vaikeaa.

Näiden muutaman esimerkin avulla nähdään, että synkronointi on tärkeää. Käydään läpi vielä yleisiä synkronointiongelmia, mitä peleissä tulee vastaan, ja kuinka ne olisi mahdollista korjata.

3.1.1 Ajoitusongelmat

Otetaan esimerkiksi animaatiotapahtuma; kranaatin heittäminen. Jos animaatio toistaa heiton, mutta kranaattiobjekti materialisoituu ja irtoaa kädestä liian aikaisin tai liian myöhään, on animaatiotapahtuma väärässä kohtaa animaatiota tai animaatio ei toistu omalla nopeudellaan.

Tämä tekee pelaajan itsensä toiminnasta sekä tekoälyvihollisten toiminnan ennakoimisesta vaikeaa. Tämän saisi korjattua tarkasti asettamalla oikean framen animaatiotapahtumalle ja tarkistamalla, miltä se näyttää pelissä. Tähän voi myös vaikuttaa animaatioiden häivyttämisestä johtuva ajanlisäys.

3.1.2 Hahmon liikuttaminen animaatiolla ja ohjelmoinnilla

Jos hahmon liikkuvuutta ohjataan sekä ohjelmoinnin että animaatioiden kautta, voivat ne rikkoa pelikokemuksen. Esimerkkinä voi käydä niin, että animaation jälkeen hahmo ei palaakaan oikeaan kohtaan, vaan on hieman eri paikassa kuin sen pitäisi. Toinen yleinen ongelma on, että hahmo liikkuu eteenpäin animaation voimin, mutta hahmoon sidottu hitbox on jäänyt alkuperäiseen kohtaan. Jos hahmoa liikutetaan ohjelmoinnilla ja liikkumisanimaatio toistuu paikallaan, niin täytyy liikkumisen nopeus sekä animaation toistonopeus synkronoida, ettei hahmo näytä juoksevan paikallaan.

Helppo ratkaisu tähän on, että painottaa vain toisen tavan käyttöä, niin ristiriitaongelmia tulee vähemmän, mutta se ei ole aina mahdollista. Tietenkin, jos pelissä ei ole ollenkaan animaatioita, liikkumista ohjataan vain ohjelmoinnilla. Melkein kaikissa cutsceneissä sen sijaan hahmoja ohjataan animaatioiden kautta.

Visuaalisesti tärkeissä liikkeissä hahmoja liikutetaan yleensä animaatioiden kautta. Esimerkkinä *Dragon Age: Origins* [20.] pelissä tehdyt viimeistelyliikkeet, joissa kaksi hahmoa lukitaan synkronoituihin toisiaan vastaaviin animaatioihin [21, 10:06] Ensimmäisen persoonan räiskintäpeleissä taas hahmojen ohjattavuus sekä niiden reagointinopeus on yleensä tärkeämpää. Useammat pelit sekoittavat kumpaakin tyyliä, riippuen käyttökohteesta.

Hahmojen kävely, juokseminen ja vastaavat liikkeet toteutetaan yleensä ohjelmoinnin kautta, mutta tarkkuutta vaativat tai tiettyä toimintaa vastaavat animaatiot taas toteutetaan animaation kautta. Esimerkiksi seinällä kiipeämistä ohjataan ohjelmoinnin kautta, mutta reunan yli kiipeäminen taas tapahtuu animaation kautta, sillä se on tarkka sekä usein toistettava liike.

3.1.3 Tilakoneen sekoaminen

Yleinen ongelma on, että kaksi toisiinsa liittyvää animaatioita kilpailee siitä, kumpi saa toistua. Esimerkkinä kävelyanimaation vaihtuminen juoksuanimaatioon voi hyppiä kummankin animaation välillä. Tämä johtuu yleensä siitä, että animaatioiden vaihtamisen aiheuttamalla parametrillä ei ole toleranssia. Esimerkkikorjauksena kävelyanimaatio voisi toistua vain, jos nopeus on isompi kuin 0, mutta alle 0,4. Juoksuanimaatio taas alkaisi vasta, jos nopeus on isompi kuin 0,5. Toleranssin laittamisen lisäksi tämä saadaan korjattua myös alemmassa kappaleessa mainittavalla häivyttämisellä.

Tilakone voi myös seota, jos ehdot täyttyvät kahdella eri tilalla samaan aikaan. Helppo esimerkki olisi, että kävely- sekä juoksuanimaatioilla on sama ehto, että hahmon nopeus on isompi kuin nolla. Tämän takia tilojen ehdot täytyy olla tarkasti määriteltyjä, että vain yksi tila voi olla aktiivisena kerrallaan. Jos peli kasvaa isoksi, ehtojen määrä kasvaa ja näin ollen myös mahdollisuudet tilakoneen sekoamiseen. Tämä huomioon otettuna on erittäin tärkeää, että ehdot sekä siirtymislogiikka animaatioiden välillä on ajateltu loppuun asti.

3.1.4 Animaatioiden häivytysongelmat

Jos pelissä on useita animaatioita, jotka siirtyvät toistensa välillä, kuten kävely sekä juoksu, on sulavamman näköistä häivyttää ne toisiinsa. Sen sijaan että animaatio vaihdetaan täysin parametrin täyttyessä, häivytetään se nopeusparametrin mukaan pikkuhiljaa seuraavaan animaatioon. Näin saadaan pelihahmon nopeuden muutos kerrottua pelaajalle visuaalisemmin.

Tämä toteutetaan häivyttämällä animaatiot toisiinsa, esimerkiksi nopeusparametrin ollessa puolivälissä kävelemistä ja juoksemista, on animaatio niiden kahden väliltä. Animaatioille voi antaa painoarvoja (*weight*) asteikolla 0–1, eli kuinka paljon animaatiot vaikuttavat lopulliseen häivytettyyn animaatioon. Oletuksena molemmat animaatiot vaikuttavat saman verran. Esimerkkinä jos kävelyanimaation painoarvo olisi 0,1, ja juoksuanimaation 0,9, siirtyisi hahmo kävelyanimaatiosta melkein heti juoksuanimaatioon.

Tämä liittyy myös edellä mainittuun ajoitusongelmaan. Jos häivytyksen parametreja ei ole tarkasti asetettu, saattaa vihollinen esimerkkinä häivyttää juoksuanimaatiota näkyvästi sitä seuranneen hyökkäysanimaation aikana. Tämä näyttää epärealistiselta, ja laskee pelikokemuksen mukavuutta. Jos hyökkäysanimaatioon on sidottu animaatiotapahtumia, kuten vahingon tekeminen, senkin suorittaminen viivästyy, koska animaation sisäiseen aikajanaan vaikuttaa edellisen animaation häivytys.

Normaalissa käytössä tästä ei tule suurta aikaeroa, mutta pienetkin eroavuudet laskevat pelikokemusta. Hyökkääminen voi tuntua esimerkiksi tahmealta. Animaatioita häivyttäessä häivytyksen kestoa ohjataan siirtymisaika (*transition duration*) -parametrilla. [Kuva 2. näkyvä 0.25f (0.25 sekuntia) on häivytyksen kesto.]

3.1.5 Animaatioiden keskeyttämisongelmat

Animaatioita keskeytettäessä täytyy olla tarkkana, että myös niihin liittyvä ohjelmointipuoli pysäytetään sekä muutetut arvot nollataan takaisin alkuperäisiin. Esimerkiksi jos vihollinen lyö pelaajaa ja pelaaja onnistuu torjumaan sen, vihollisen lyömisanimaatio keskeytetään kilpeen osumi-

sen jälkeen. Lyönnin animaatioon sidottu animaatiotapahtuma voi olla lauennut jo, ja sen käynnistämä logiikka voi kuitenkin toistua vielä tehden pelaajaan vahinkoa. Tämä saa pelin tuntumaan rikkinäiseltä ja epäreilulta.

Tämän voisi korjata esimerkiksi laittamalla animaatiotapahtuman alkuun tarkistuspiste, että tapahtuman laukaisema animaatio varmasti toistuu vielä. Toinen mahdollinen ratkaisu olisi tarkistaa, onko tilakoneessa animaatiota vastaava tila aktiivisena, kuten hyökkäystila. Jos ehdot eivät täyty, vahingon tekemisen suoritus pysähtyy.

3.1.6 Pelaajan syötteen ajoitus sekä puskurointi

Vaikka äskeisessä luvussa varoiteltiin animaation keskeyttämisen vaaroista, on se todella tärkeä työkalu. Äskeisen passiivisen esimerkin lisäksi voitaisiin ottaa vielä toinen; oletetaan että pelaaja on lyömässä vihollista, mutta huomaakin että vihollinen aloitti myös lyönnin mutta hieman ennen pelaajaa. Jos pelaaja ei pysty keskeyttämään hyökkäystä ja vaihtaa torjumiseen tai väistää, laskee tämä pelimukavuutta.

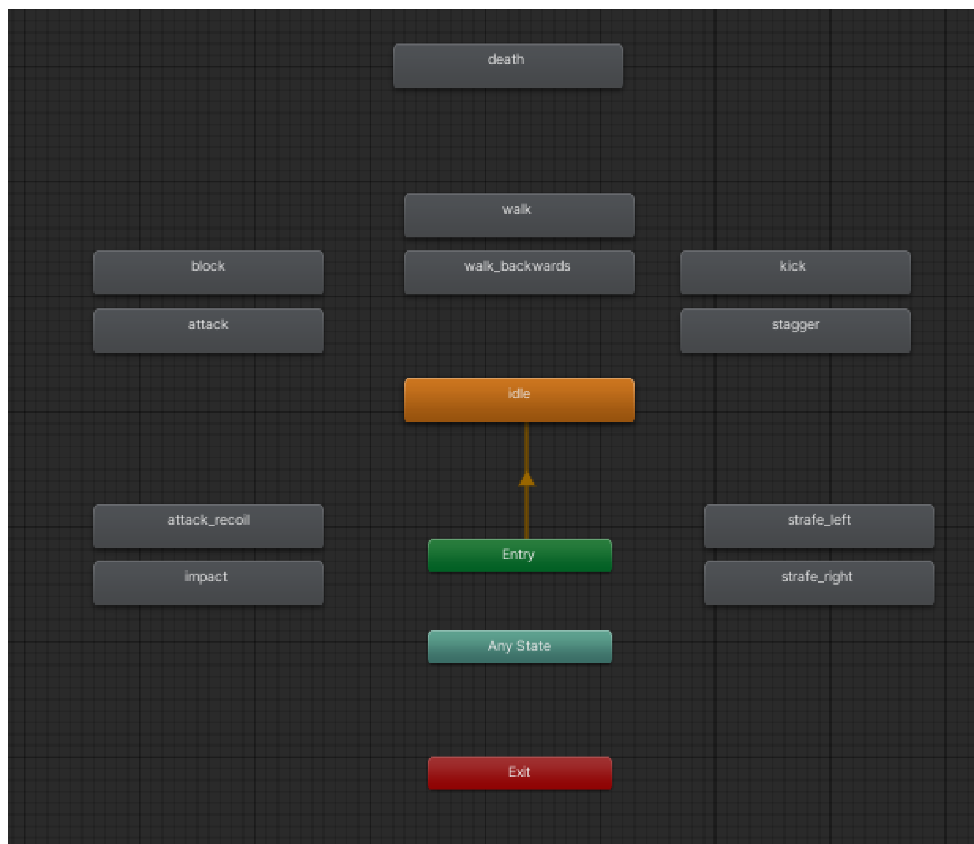
Tähän liittyy myös animaatioiden puskurointi. Jos animaatio on sellainen, jota ei pelimekaniikoiden vuoksi voi keskeyttää, on tärkeää ottaa viimeisin pelaajan syöte talteen, ja toistaa se heti kun mahdollista. Esimerkiksi jos pelaaja ottaa vahinkoa, lukitaan hahmo *stagger*-animaatioon sen loppuun asti. Jos kuitenkin tämän aikana pelaaja yrittää tehdä toista liikettä, kuten torjua, täytyy se toistaa heti, kun edellinen animaatio on päässyt loppuunsa.

4 Projektin toteutus ja työvaiheet

Tässä luvussa seurataan projektin toteutusta alusta loppuun.

4.1 Aloitus

Ensin loin tyhjän projektin Unityyn, käyttäen 3D (Built-In Render Pipeline) pohjaa. Etsin Mixamo-sivustolta aiheeseen sopivia animaatioita. Löydettyäni muutaman hyvän animaation, latasin ne sekä siirsin ne pelimoottoriin. Saatuani kaikki animaatiot pelimoottoriin, loin pelaajahahmon. Seuraavaksi lisäsin pelaajahahmolle Animator-komponentin ja siirsin animaatiot hahmon Animator-ikkunaan. [Kuva 3.]



Kuva 3. Animator-ikkuna, jossa animaatioita ilman visuaalisia siirtymiä. *Idle*-animaatio toistuu oletuksena.

4.2 Pelaajahahmon liikkuminen

Tein pelaajalle yksinkertaisen liikkumisen. Pelaaja sekä hahmo ovat katseeltaan lukittuna toisiinsa. Pelaaja pystyy liikkumaan kohti vihollista, poispäin siitä sekä kävelemään sivuttain vasemmalle tai oikealle.

Löysin sopivan kävelyanimaation, joka toimi hyvin kävellessä eteenpäin. Taaksepäin käveltäessä se kuitenkaan ei toiminut. Ongelman sain korjattua kopioimalla eteenpäin kävelyn animaation, ja asettamalla kopion toistonopeudeksi -1. Näin uusi animaatio toistuu käänteiseen suuntaan, näyttäen taaksepäin kävelyltä.

1. Tietyt animaatiot pysäyttävät hahmon liikkeen. Ne ovat *kick*, *block*, *attack*, *stagger* sekä *impact*. Kun animaatio on toistettu, hahmo voi liikkua jälleen. Tämän voisi yhdistää liikkumiseen, jos ala- ja yläruumiin animoisi erikseen. Tässä käytettäisiin animaatioiden tasoja (animation layer masks). Tätä en kuitenkaan lähtenyt tekemään.

4.3 Pelaajan kamera

Kamera on yksinkertainen. Kameran sijainti pelaajasta sekä rotaatio on asetettu manuaalisesti valmiiksi, ja sen on pelaajan lapsiobjekti, joten sen kääntyy pelaajan mukana. Pelaaja itse on lukittu katsomaan vihollista *LookAt*-funktiolla. Vihollisella on sama funktio, ja näin vihollinen ja pelaaja ovat lukittuna katsomaan toisiaan.

4.4 Animaatiot

Ohjaan animaatioita itse ohjelmoinnin kautta, käyttämättä Unityn visuaalista työkalua. [Kuva 4.]

```
10 references
void ChangeAnimationState(string newState) {

    bool isAction = actionStates.Contains(newState);   1

    // makes these able to override other animations
    if(isDoingAction && !isAction) { return; }          2

    // stop the animation from going into loop
    if (currentState == newState)  return;          3

    if (isAction) { isDoingAction = true; }   4

    // add delay to prohibit idle/run animations from taking over
    if(isAction) {
        StopAllCoroutines();
        StartCoroutine(AttackAnimationDelay(newState));   5
    }
    else {
        _animator.CrossFadeInFixedTime(newState, 0.25f);
        // set the current state
        currentState = newState;          6
    }
}
```

Kuva 4. Kuvakaappaus koodista, joka vaihtaa animaatioita.

2. Ensin funktioon syötetään uuden animaatiotilan nimi. Sen jälkeen luodaan booleani, joka tarkistaa onko uusi animaatiotila actionstate. Actionstatet ovat niitä animaatiotiloja, jotka voivat keskeyttää jo toistuvan animaation. Esimerkiksi hyökkäysanimaatio voi keskeyttää kävelyanimaation.

3. Tällä koodirivillä tarkistetaan, onko samanniminen animaatiotila toistumassa jo. Tämä estää sen, että sama animaatio ei toistu uudestaan ja uudestaan, vaikka se täyttääkin ehdot toistumiseen. Kävelyanimaatio sekä muut jatkuvaa toistoa tarvitsevat animaatiot ovat itsestään asetettu jatkuvasti toistuvaksi.

4. Jos newStaten tuoma animaatiotila on actionState, asetetaan *isDoingAction*-booleani todeksi, jotta passiiviset animaatiot eivät voi yliajaa sitä. Esimerkiksi jos pelaaja on paikallaan ja hyökkää, voisi *idle*-animaatio yrittää toistua, mutta tämä estää sen.

5. Tämä osa koodia toistuu vain, jos newState on tyyppiä actionState. Kaikki jo toistuvat coroutinet pysäytetään. Tämä estää jo toistumassa olevia coroutineita tekemästä asioita, kun ne käynnistänyt animaatio on jo keskeytetty. Tämä myös estää tähän liittyvien bugien synnyn [17, s22]. *AttackAnimationDelay*-funktio varmistaa, että jo toistuva *action*-animaatio ehtii toistua loppuun asti. Vasta sen jälkeen *isDoingAction*-booleani muuttuu taas kielteiseksi.

6. Jos newState ei ollut actionState, häivytetään se jo toistuvaan animaatioon 0,25 sekunnin ajan, ja nykyinen animaation tila päivitetään oikeaksi.

Seuraavassa kuvassa on ohjelmointia hyökkäysanimaatioiden viiveeseen liittyen. [Kuva 5.]

```
1 reference
IEnumerator AttackAnimationDelay(string stateToChange) {   1

    _animator.CrossFadeInFixedTime(stateToChange, 0.25f);        2
    float waitTime = ReturnAnimationLengthByName(foundAnimations, stateToChange);

    var timeToPlayAnimationInOneSecond = 1 / (AttackInterval / waitTime);
    _animator.speed = timeToPlayAnimationInOneSecond;  3

    yield return new WaitForSeconds(AttackInterval);  4
    _animator.speed = 1;

    isDoingAction = false;
    currentState = stateToChange;  5
}
```

Kuva 5. Kuvakaappaus hyökkäysanimaatioille viiveen antavasta koodista.

1. Tein hyökkäysanimaatioiden toistolle oman funktion, koska niiden täytyy toistua loppuun asti. IEnumerator oli tälle kätevä vaihtoehto, koska siinä voi pysäyttää koodin suorituksen määritellyksi ajaksi. Tämä funktion ottaa sisäänsä myös uuden tilan.

2. Ensin uusi tila häivytetään jo toistuvaan animaatioon, samalla tavalla kuin normaalin animaation kanssa. Tällä rivillä myös etsitään uutta tilaa vastaavan animaation kesto animaatiolistasta. Tällä saadaan animaation tarkka kesto.

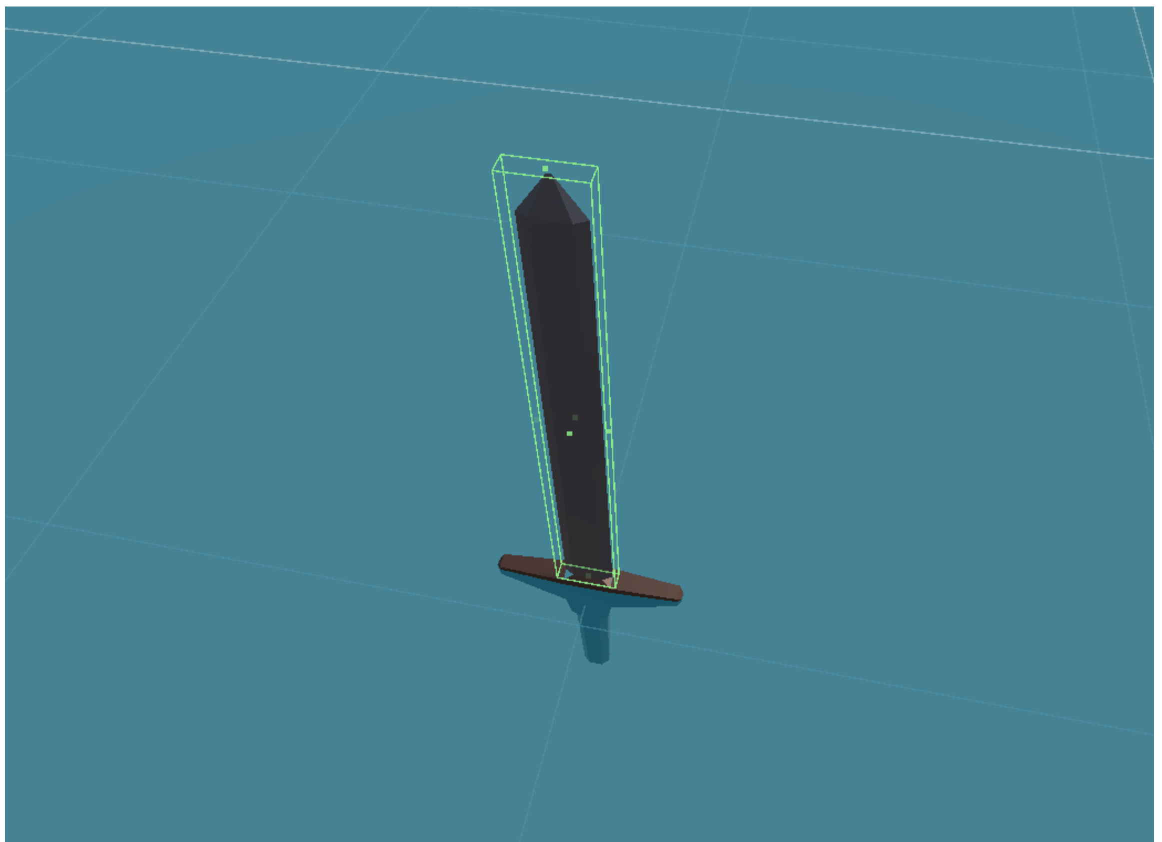
3. Seuraavaksi animaation toistonopeutta muutetaan niin, että se toistuu asetetussa ajassa. Tässä varmistetaan, ettei animaatio toistu liian nopeasti tai liian hitaasti.

4. Seuraavalla rivillä odotetaan sen aikaa, että animaatio on toistunut. Sen jälkeen animaattorin toistonopeus palautetaan vakioon.

5. Lopuksi *action*-booleani palautetaan epätodeksi, sekä nykyinen animaation tila päivitetään taas oikeaksi.

4.5 Hitboxit

Aluksi mallinsin yksinkertaisen miekan sekä kilven peliin, joita mekaniikoiden esittäminen tarvitsee. Aseen sekä kilven hitbox täytyy olla tarkasti asetettu oikean mittaiseksi, että niiden käyttäminen tuntuu realistiselta. [Kuva 6.]



Kuva 6. Hitbox miekassa (vihreä alue).

Yleinen ongelma hitboxeja tehdessä aseiden kanssa on se, että ase osuukin lyöjään itseensä. Tämän saa helposti korjattua sillä, että tarkistaa, onko osutun kappaleen juuriobjekti sama, kuin miekan juuriobjekti. Jos on, miekka osui lyöjään itseensä. Näin ollen koodin suoritus voidaan lopettaa tähän. [Kuva 7.]

Tämän olisi myös voinut ottaa huomioon käyttämällä tasopohjaista törmäysmatriisia (*layer collision matrix*) [22.], mutta totesin tämän olevan riittävä ratkaisu näin pienen skaalan projektiin.

```
Unity Message | 0 references
private void OnTriggerEnter(Collider other) {

    if(other.gameObject.transform.root.gameObject == transform.root.gameObject) { return;}
```

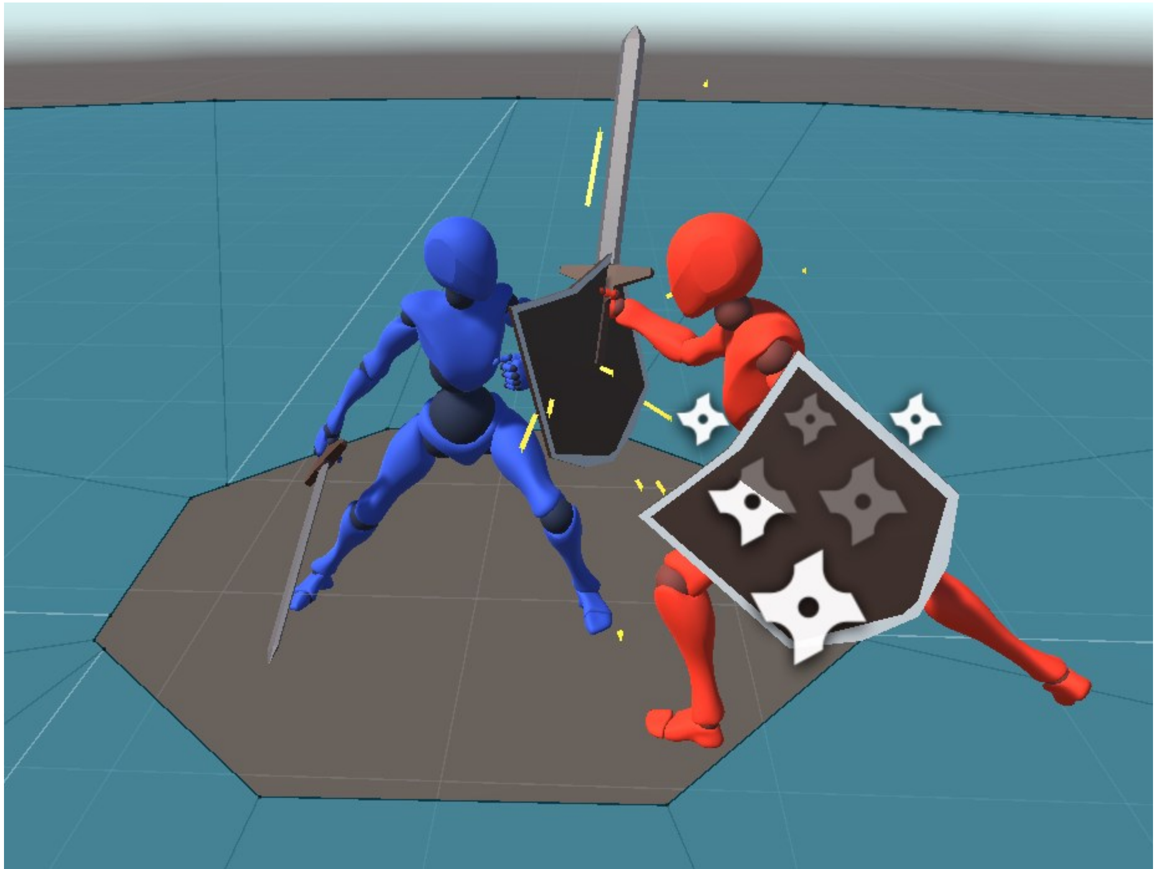
Kuva 7. Jos lyöjä osuu itseensä, koodin suoritus lopetetaan siihen.

Tässä vaiheessa peliin tuli hyökkäys sekä torjuminen, joiden logiikan toteutin miekassa sekä kilvessä olevien hitboxien avulla. Kun hahmo lyö toista hahmoa, seuraukset riippuvat siitä, mihin miekka osuu. Jos hahmo lyö toista hahmoa, mutta toinen hahmo on torjumassa ja miekka osuu kilpeen, lyöjä saa *recoil*-animaation ja torjuja *impact*-animaation.

Jos taas puolustaja ei ollut torjumassa, ja miekka osuu hahmon hitboxiin kilven sijasta, hyökkäysanimaatio toistuu loppuun asti. Samalla vastaanottajan toistuvat animaatiot keskeytetään ja hän saa *stagger*-animaation. Riippumatta puolustajan animaatiosta, jos hyökkääjä potkaisee ja osuu puolustajaan, saa puolustaja *stagger*-animaation.

4.6 Particle System

Kun olin saanut torjumisen sekä hyökkäyksen valmiiksi, paransin pelikokemusta hieman. Tein uuden Particle Systemin, ja laitoin se laukaisemaan tietyn määrän partikkeleita kilvestä poispäin miekan siihen osuessa, simuloiden kipinöitä [Kuva 8.] Tämä antaa visuaalista palautetta pelaajalle ja tekee pelistä realistisemman.



Kuva 8. Partikkeliefekti aktivoituu oikealla hetkellä.

4.7 Sivuttain kävely

Pelaajahahmo sekä tekoälyvihollinen ovat katseelta lukittuna toisiinsa, mutta voivat kuitenkin kävellä sivusuunnassa. Tämän takia peli tarvitsi uuden animaation; *strafe* eli sivuttain kävelyn. Latasin vain vasemmalle kävelyn animaation, josta tein kopion. Asetin kopiolle toistonopeudeksi -1, jonka jälkeen animaatiot olivat valmiina vasemmalle ja oikealle kävelylle. Tämän jälkeen muutin olemassa olevaa liikkumislogiikkaa niin, että se huomioi myös sivusuuntaisen liikkeen, ja toistaa sivuttaiskävelyanimaatiota silloin. [Kuva 9.]

Kuva 9. Pelaajahahmo kävelee vasemmalle, samalla kun vihollishahmo kävelee pelaajaa kohti.

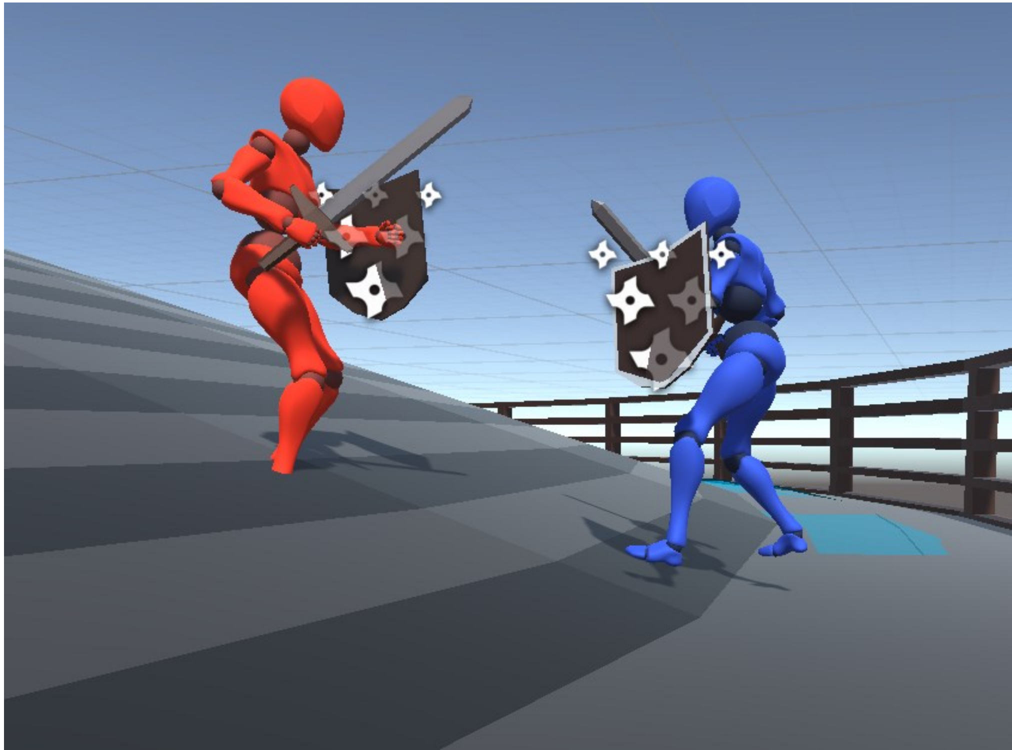
Koodi oikean animaation toistamiselle oli yksinkertainen. Tarkistin hahmon nopeusvektorin x- ja z- akseleiden liikkeen ja sen perusteella valitsin sopivan kävelyanimaation. [Kuva 10.]

```
if (vel.z > 1) {
        // eteenpäin
        ChangeAnimationState(Moving_Forward);
}
else if (vel.z < -1) {
        // taaksepäin
        ChangeAnimationState(Moving_Backward);
}
else if (vel.x > 1) {
        // oikealle
        ChangeAnimationState(Strafing_Right);
}
else if (vel.x < -1) {
        // vasemmalle
        ChangeAnimationState(Strafing_Left);
```

Kuva 10. Oikean animaation valitsemista pelaajan liikesuunnan mukaan.

4.8 Inverse Kinematics

Korvasin tasaisen areenan uudella mallilla, jonka tein Blender-sovelluksessa. Areenassa on nyt korkeuseroja. Tästä päästään yleiseen ongelmaan animaatioissa peleissä, joissa pelialueella on eroja käveltävän alueen korkeudessa. [Kuva 11.]



Kuva 11. Hahmojen jalat eivät mukaudu lattian kulmaan.

Tämä saadaan ratkaistua ottamalla käyttöön *Inverse Kinematics*, jolloin raajoja voi ohjata erillään varsinaisesta animaatiosta. Inverse kinematics eli käänteinen kinematiikka, on animaatio- ja fysiikkatekniikka, jolla määritellään raajan päätepisteen sijainti [23.] Pelimoottori kääntää niveliä automaattisesti niin, että päätepiste saadaan haluttuun paikkaan. Tämän vastakohtana on *Forward Kinematics*, jossa raajan niveliä liikutellaan, ja raajan loppukohta liikkuu niiden mukana. Suurin osa hahmoanimaatioista on toteutettu forward kinematics -pohjaisesti, mutta inverse kinematics lisätään sen päälle usein raajojen hienosäätämiseksi.

Käytin alun perin geneeristä rigi-tyyppiä Unityssä, mutta tämä ei toiminut inverse kinematicsin kanssa. Aikani tätä tutkiessa huomasin, että toimiakseen rigin tyypiksi pitää asettaa humanoidi. Olin ehtinyt poistaa alkuperäiset animaatiopaketit ja kopioinut niistä vain raa'an animaatiodatan, joten jouduin etsimään ne uudestaan. Latasin animaatiot uudestaan, ja tällä kertaan asetin niiden rigityypiksi humanoidin.

Tämän jälkeen inverse kinematic -liittyvä tekevä koodi alkoi toimimaan. Vaihdon yhteydessä syntyi kyllä liuta bugeja, joita korjatessa meni aikansa. Suurin osa tuli kylläkin helposti korjattavista nimeämisongelmista. Jotkut animaatiot toistuivat hieman väärällä rotaatiolla tai liikkuivat pelaajahahmon sijainnista erillään, ja niitä piti hienosäätää. Käytin inverse kinematicsia pelaajahahmon jalkoihin, minkä jälkeen ne mukautuivat sen alla olevan maaston korkeuteen ja kulmaan. Punaiselle hahmolle en operaatiota vielä tehnyt, vertailun vuoksi. [Kuva 12.]

Kuva 12. Punainen hahmo ei käytä inverse kinematicseja jaloissaan, sininen käyttää.

Seuraavana kuvakaappaus koodista, joka ohjaa jalkojen liikettä inverse kinematicsilla. [Kuva 13.]

```
2 references
void FootCalculation(AvatarIKGoal footIK, Transform footTransform, float soleOffset) 1
{
    Vector3 origin = footTransform.position + Vector3.up * 0.5f;
    if (Physics.Raycast(origin, Vector3.down, out RaycastHit hit, 5f, groundLayer)) 2
    {
            Vector3 targetPos = anim.GetIKPosition(footIK);
            targetPos.y = hit.point.y + soleOffset + 0.2f; // 0.2 is extra offset 3
            anim.SetIKPosition(footIK, targetPos);

            Vector3 forward = Vector3.ProjectOnPlane(footTransform.forward, hit.normal);
            anim.SetIKRotation(footIK, Quaternion.LookRotation(forward, hit.normal)); 4

    }
}
```

Kuva 13. Kuvakaappaus koodista, joka ohjaa Inverse Kinematicsin kautta jalkojen liikettä.

1. Tein uuden funktion Inverse Kinematicsien käyttöä jalkoja varten. Funktio ottaa sisäänsä AvatarIkGoalin, transformin sekä offset-muuttujan.

2. Ensin laskin alkupisteen tulevalle Raycastille. Raycasti alkaa 0,5 yksikköä jalasta ylöspäin. Tein näin sen takia, että Raycast alkaa varmasti maanpinnan päältä. Sen jälkeen ammuin Raycastin alaspäin äsken tehdystä alkupisteestä. Jos säde törmää johonkin, sen informaatio tallennetaan Raycasthit *hit*-muuttujaan. Raycastin maksimipituus on 5 yksikköä. Raycast osuu myös vain *Ground*-tasolla oleviin hitboxeihin. Vain objekteille, joiden päällä voi kävellä, on asetettu tämä taso.

3. Hain ensin kohdepisteen. Animaattorista sen sai haettua kätevästi. Sen jälkeen muutin kohdepisteen y-arvoa. Lisäsin siihen jalan pituuden, sekä pienen lisäpituuden, jotta se on tasassa käveltävän tason kanssa. Lopuksi liikutin jalan äsken määriteltyyn kohtaan.

4. Tässä segmentissä muutetaan jalan kääntymistä. Sen täytyy olla myös tasassa käveltävän tason kanssa. Ensin otin jalasta suunnan sen pituussuunnassa ja alla olevasta lattiasta sen kulman. Sen jälkeen käänsin jalan.

5 Yhteenveto

Projektin tavoitteena oli tehdä kolmannen persoonan keskiaikainen taistelupeli, jossa pelaaja sekä tekoälyvihollinen taistelevat toisiaan vastaan. Näiden animaatiot on hyvin synkronoitu koodin kanssa, mikä saa aikaan sulavan sekä realistisen pelikokemuksen.

Pelin tehtiin *Unity*-pelimoottoria käyttämällä, tarvittavat animaatiot hankittiin *Mixamo*-verkkosivulta sekä osa tarvittavista malleista tehtiin itse *Blender*-ohjelmalla. Kaikki pelissä olevat mekaniikat ohjelmoitiin itse, kuten liikkuminen, taisteleminen, animaatioiden vaihtaminen sekä jalkojen ohjaaminen käänteisen kinematiikan kautta.

Pelin tekemisen aikana vastaan tuli odotetusti ongelmia. Internetistä ladattujen animaatioiden integrointi peliin, inverse kinematicsin soveltamisen vaikeus, animaatioiden toistumisen hallinta sekä animaatiotilojen vaihtamisen logiikka aiheuttivat päänvaiva. Nämä ja muut vastaan tulleet ongelmat ratkaistiin ongelmanetsinnällä, koodia uudelleen kirjoittamalla sekä aiheesta lisää lukemalla.

Projekti korostaa animaatioiden sekä tekoälyn synkronoinnin tarvetta. Oikein toistuvat animaatiot parantavat pelikokemusta, väärin toistuvat heikentävät sitä. Tulevissa projekteissa voitaisiin keskittyä edistyneemmän tekoälyvihollisen tekemiseen, ylä- ja alaruumiin animoimiseen erikseen sekä inverse kinematicsin käyttämiseen kaikille raajoille.

Lähteet

1. Unity Technologies. (2025). *Unity* [Pelimoottori]. Unity Technologies https://unity.com

2. Adobe. (2025). *Mixamo*. [Verkkosivu] https://www.mixamo.com

3. Blender Development Team (2025). Blender (Versio 5.0) [Tietokoneohjelma]

4. *Red Dead Redemption 2*. Rockstar. [Videopeli]. (Xbox, Playstation, Pc), 2018

5. RabidRetrospectGames, *RED DEAD REDEMPTION 2 All Cutscenes Movie (Game Movie)* [Verkkosivu]. 2019 [Viitattu 10. maaliskuuta 2025]. Saatavissa: https://www.youtube.com/watch?v=zPlbP0K_GtM

6. *RimWorld*. Ludeon Studios. (2018). [Videopeli]. PC, Steam.

7. Sylvester T. *RimWorld: Contrarian, ridiculous, and impossible game design methods* [Verkkosivu] 2017. [Viitattu 10. maaliskuuta 2025] Saatavissa: https://www.gdcvault.com/play/1024232/-RimWorld-Contrarian-Ridiculous-and

8. Hunter, William (2007-09-10). "The Original Video Game". [Viitattu 9. joulukuuta 2025] [Verkkosivu].

9. Atari. (1972). *Pong.* [Videopeli].

10. Nintendo. (1981). *Donkey Kong.* [Videopeli].

11. Donkey Kong – First Versions [Verkkosivu]. [Viitattu 10. maaliskuuta 2025]. Saatavissa: https://www.firstversions.com/2015/08/donkey-kong.html

12. Brøderbund. *Prince of Persia* [Videopeli]. 1989. Apple II, PC

13. Seth Porges. *How The Original 'Prince Of Persia' Changed Video Game Animation* [Verkkosivu]. 2017. [Viitattu 10. maaliskuuta 2025]. Saatavissa: https://www.forbes.com/sites/sethporges/2017/12/19/how-the-original-prince-of-persia-changed-video-game-animation/

14. Nintendo. (1996). *Super Mario 64*. [Videopeli].

15. Lucas M. Thomas. *The Genius of Super Mario 64.* [Verkkosivu]. 2012. [Viitattu 10. maaliskuuta 2025]. Saatavissa: https://www.ign.com/articles/2011/09/28/the-genius-of-super-mario-64

16. Saarainen, M. (2019). *Game development with Unity: A study on mechanics and user interaction in 3D games* [Master's thesis, **University of Lapland**]. **Theseus.fi**. Saatavissa: https://www.theseus.fi/bitstream/handle/10024/167279/Saarainen_Mirve.pdf

17. Gong, Lechen, *A Real-Time Animation Toolkit with Blending, Root Motion Control, Interruption, Events, and Layering* (2025). Programming Theses and Dissertations. 11. Saatavissa: https://scholar.smu.edu/cgi/viewcontent.cgi?article=1000&context=guildhall_programming_etds

18. Unity Technologies. (2025). *Mechanim Animation System.* Unity Manual. [Viitattu 11. marraskuuta 2025] Saatavissa: https://docs.unity3d.com/462/Documentation/Manual/MecanimAnimationSystem.html¨

19. Pavan P Y, Shanu Gour, & Lalit Kumar P Bhaiya. (2024). *Designing Custom State Machine Instead of Animation State Machine in Unity for Developing 3rd Person Action Game*. *International Journal of Progressive Research in Science and Engineering*, *5*(07), 95–103. [Viitattu 3. joulukuuta 2025] Saatavissa: https://journal.ijprse.com/index.php/ijprse/article/view/1101

20. BioWare. (2009). *Dragon Age: Origins* [Videopeli]. Electronic Arts.

21. NeoGamer – The Video Game Archive. *Behind the Scenes - Dragon Age: Origins [Making of*] [Verkkosivu]. 2022. [Viitattu 10. maaliskuuta 2025]. Saatavissa: https://www.youtube.com/watch?v=yXoteXWdGww

22. Unity Technologies. (2025). *Layer-based collision detection*. Unity Manual. [Viitattu 5. joulukuuta 2025] Saatavissa: https://docs.unity3d.com/6000.2/Documentation/Manual/LayerBasedCollision.html

23. Dave Pagurek. *Simple Inverse Kinematics.* [Verkkosivu]. 2017. [Viitattu 10. maaliskuuta 2025]. Saatavissa: https://www.davepagurek.com/blog/inverse-kinematics/