KARELIA-AMMATTIKORKEAKOULU
Tietojenkäsittelyn koulutus

Niku Räsänen

AALTOPOHJAINEN VIHOLLISGENERAATTORI UNREAL ENGINESSÄ

Opinnäytetyö
Toukokuu 2021

OPINNÄYTETYÖ
Toukokuu 2021
Tietojenkäsittelyn koulutus

Tikkarinne 9
80200 JOENSUU
+358 13 260 600 (vaihde)

Tekijä
Niku Räsänen

Nimeke
Aaltopohjainen vihollisgeneraattori Unreal Enginessä

Toimeksiantaja
Kollektiwe Oy

Tiivistelmä

Tämä opinnäytetyö on toiminnallinen työ, jossa toteutettiin Unreal Engine 4 -pelimoottorilla tekoälyagentteja rakentava, synnyttävä ja ohjaava järjestelmä. Työn tarkoituksena oli järjestelmän toteuttamisen lisäksi myös tutkia, kuinka yksittäisestä tekoälyagentista saadaan toteutettua toisistaan poikkeavia persoonallisia instansseja.

Opinnäytetyön tietoperustassa esitellään järjestelmän toteuttamiseen käytettävä pelimoottori sekä sen sisältämät tekoälyn rakentamiseen käytettävät järjestelmät ja komponentit. Tietoperustassa tutkitaan myös näiden järjestelmien ja komponenttien yhteistyötä. Opinnäytetyössä esitellään agentin sisältämien muuttujien asettaminen sattumanvaraiseksi ja se, miten näitä muuttujia hyödynnetään personoinnin saavuttamiseksi. Työssä esitellään myös agentin liikkumiskeinojen, itsesuojelun, sekä hyökkäyksen toteuttaminen.

Toteutettu järjestelmä tuo peliin tuntumaa, jossa tekoälyagentteja olisi toteutettu useita erilaisia sen sijaan, että yksittäisestä agentista luodaan instansseja. Järjestelmä toteuttaa onnistuneesti tehtävät, joita varten se luotiin. Opinnäytetyön lopussa pohditaan toteutus- ja toimintatapoja, joilla toteutettu järjestelmä olisi ollut parempi. Järjestelmän jatkokehitykseen otetaan työssä kantaa teoreettisella tasolla.

Kieli
suomi

Sivuja 56
Liitteet 0
Liitesivumäärä 0

Asiasanat
Unreal Engine 4, tekoäly, tekoälyagentti, kontrolloitu sattumanvaraisuus

	THESIS **May 2021** **Business Information Technology** Tikkarinne 9 80200 JOENSUU FINLAND + 358 13 260 600 (switchboard)

Author
Niku Räsänen

Title
Wave-based Enemy Generator in Unreal Engine 4

Commissioned by
Kollektiwe Oy

Abstract

This thesis focuses on developing a system in Unreal Engine 4 which initializes, builds, and guides intelligent agents. In addition to the system, this thesis also examines how to create distinctive instances of a singular intelligent agent using blueprints.

The thesis first introduces Unreal Engine 4 and its components that are needed to create the agent generating system. The thesis also investigates how these components behave, what their function is, and how they co-operate with each other. This thesis presents how to randomize variables of the intelligent agent and how to use these variables to personalize instances of the agent. The thesis also introduces the creation of movement, self-preservation and attacking logics of the agent.

The system creates the effect of having many different agents in the game instead of multiple instances of the same agent. The system successfully completes the required tasks. Different actions, implementations and approaches that might have yielded subjectively better results are examined at the end of this thesis. Further development of the agent generating system is pondered theoretically.

Language Finnish	Pages 56 Appendices 0 Pages of Appendices 0

Keywords
Unreal Engine 4, artificial intelligence, intelligent agent, controlled randomness

Sisältö

1 Johdanto 5
2 Unreal Engine 6
2.1 Blueprintit 6
2.2 C++-Luokat 6
2.3 Unreal Editor, plug-init ja marketplace 7
3 Tekoäly Unreal Engine 4 pelimoottorissa 8
4 Tekoälyagentit 9
4.1 Tekoälyagentin keho 9
4.2 Tekoälynagentin sielu 10
4.3 Tekoälyagentin aistien simulointi 10
4.4 Tekoälyagentin aivot 10
4.5 Tekoälyagentin muisti 15
4.6 Tekoälyagentin järjestelmien yhdistäminen 15
5 Tekoälyn ympäristö 15
5.1 Ympäristössä navigointi 15
5.2 Ympäristön tutkinta 16
6 Työskentelyn koordinointi toimeksiantajan kanssa 17
7 Aaltopohjaisen vihollisgeneraattorin toteuttaminen tekoälyn näkökulmasta 17
7.1 Vaatimukset 17
7.2 Peliprojektin valmiiden osien hyödyntäminen 18
7.3 Tekoälyagentin alustaminen ja personointi 19
7.4 Tekoälyagenttien synnyttäminen pelimaailmaan 28
7.5 Tekoälyagenttien tehtävät 32
7.6 Agentin liikkuminen 36
7.7 Tekoälyagenttien elinkaaren päättyminen 44
7.8 Tekoälyagenttia tukevat muut toiminnot peliprojektissa 44
8 Integrointi 48
9 Jatkokehitysideat 48
9.1 Taidot 48
9.2 Pelaajan puolella taistelevat hahmot 49
9.3 Aallot 50
9.4 Vahingon aiheuttaminen 50
9.5 Pisteytys ja pelaajalle osoitettava Data 50
9.6 Tekoälyagentin aistit 51
10 Pohdinta 51
10.1 Onnistumiset ja haasteet 52
10.2 Vaihtoehtoiset toteuttamistavat 52
Lähteet 55

1 Johdanto

Opinnäytetyössä toteutetaan toimeksiantona peliprojektiin tekoälyjärjestelmä käyttäen Unreal Engine 4 -pelimoottoria. Järjestelmään kuuluvat tekoälyagentti, sen toimintaa ohjaavat komponentit, sekä agentteja synnyttävä entiteetti. Toteutettava tekoälyagentti toimii peliprojektissa vihollishahmona, joten sille rakennetaan vihollishahmoille ominaisia tehtäviä.

Tekoälyagentille rakennetaan järjestelmä, jolla yhdestä samasta tekoälyagentista voidaan synnyttää toisistaan eroavia instansseja. Agentin rakentamisessa hyödynnetään ohjattua sattumanvaraisuutta, jolla on tarkoitus saada aikaan suuri määrä erilaisia mahdollisia instansseja vihollishahmosta. Sattumanvaraisuuden tehtävänä on tuoda jokaiseen eri pelikertaan jotain uutta, vaikka peli onkin sama.

Vihollishahmoja synnyttävä entiteetti rakennetaan synnyttämään vihollishahmoja aaltopohjaisesti. Aaltopohjaisuuteen toteutetaan myös sattumanvaraisuutta agenttien vaikeudessa ja määrässä.

Opinnäytetyön teoriaosuudessa tarkastellaan työssä käytettävää Unreal Engine 4 -pelimoottoria, sekä sen sisältämiä tekoälyjärjestelmiä ja -komponentteja. Järjestelmien ja komponenttien tehtäviä ja yhteistyötä pyritään kuvaamaan vertaamalla niitä ihmisen kehon, sielun ja mielen tehtäviin, jossa mieli on jaettu aivoiksi ja muistiksi. Toteutettu järjestelmä esitellään tekoälyagentin elinkaaren mukaisessa järjestyksessä, alkaen alustamisesta ja loppuen tekoälyagentin kuolemiseen. Lopuksi vastataan siihen, että mitä olisin tehnyt toisin ja miten jatkaisin peliprojektia tekoälyn kontekstissa.

2 Unreal Engine

Unreal Engine on Epic Gamesin kehittämä ja julkaisema pelimoottori. Pelimoottori sisältää pelin tekemistä varten käytettäviä hyödyllisiä työkaluja ja komponentteja. Unreal Engineä käytetään erikokoisten toimijoiden toimesta, ja se sopiikin sekä yksittäisen kehittäjän, että suurien kehitystiimien käyttöön. Unreal Engine on suosittu valinta pelimoottoriksi sen järjestelmäriippumattomien kehitysmahdollisuuksien, sekä sen skaalautuvuuden vuoksi. Myös mahdollisuus täysin visuaaliseen ohjelmistokehitykseen vetää uusia kehittäjiä puoleensa. (Lee 2015.) Unreal Enginessä ohjelmoidaan käyttäen blueprinttejä, tai C++-kieltä.

2.1 Blueprintit

Blueprintit ovat Unreal Enginen sisältämä visuaalisen ohjelmoinnin ja skriptauksen menetelmä. Blueprintit perustuvat olio-ohjelmoinnin periaatteisiin, eli luokkapohjaiseen jaotteluun, jossa yksi blueprint vastaa yhtä luokkaa. Täten blueprintit voivat olio-ohjelmoinnin periaatteiden mukaisesti periytyä toisesta luokasta. Itse luoduille blueprinteille valitaan luomisen yhteydessä mihin tarkoitukseen blueprinttiä ollaan luomassa, joka tapahtuu valitsemalla luotavalle blueprintille parent blueprint. Erilaisia valmiita parent blueprinttejä on satoja kappaleita, aina pelaajahahmosta pelimoodiin tai pelin tallentamisen blueprinttiin. Nämä parent blueprintit sisältävät komponentteja, jotka ovat hyödyllisiä tai välttämättömiä itse luodun blueprintin toimimiselle tehtävässään. Parent blueprinteillä on myös toisistaan eroavia asetuksia, esimerkiksi replikointiin, näkyvyyteen, ja niiden interaktioon muiden blueprinttien kanssa liittyviä asetuksia. (Unreal Engine 2021a.)

2.2 C++-Luokat

Unreal Enginen ohjelmointikieli on C++. Kieli toimii myös blueprinttien rakennuspalikkana, joten kaikki valmiit parent blueprintit ja komponentit ovat c++-kielellä toteutettuja. Myös itse toteutetut blueprintit ovat kokonaisuudessaan c++-kieltä

visuaalisen ulkomuotonsa alla. Kaikkia Unreal Enginessä olevia toiminnallisuuksia ja funktiota ei ole vielä avattu käytettäväksi blueprinteissä, vaan ne ovat hyödynnettävissä vain käyttämällä C++-ohjelmointia. Unreal Enginessä ei ole tukea muille kielille (Versio 4.26). Muita kieliä on mahdollista käyttää vain käyttämällä ulkoisen tekijän luomaa koodikirjastoa tai komponenttia. (GamesFromScratch 2016.)

2.3 Unreal Editor, plug-init ja marketplace

Unreal Editor on pelimoottorin sovelluksen nimi. Sovellus sisältää monta erilaista näkymää ja työkalua pelisovelluksen toteuttamista varten. Näkymiä ovat esimerkiksi pelimaailman näkymä ja tiedostojen selainnäkymä. (Cordone 2019.) Tekoälyn kannalta tärkeää tietää editorista ovat projekti- ja sovellusasetukset. Projektiasetuksista löytyy oma alavalikko tekoälyn asetuksille. Sovellusasetuksien kautta otetaan käyttöön eksperimentaalisella tasolla olevat järjestelmät, joita tekoälyä käyttävässä projektissa voidaan hyödyntää.

Editorin avulla rakennettu pelisovellus käynnistetään editorissa, tai rakennetaan (build) editorin ulkopuoliseksi sovellukseksi, joka voidaan käynnistää ilman editoria. Täten suurin osa sovelluksen testaamisesta suoritetaan suoraan editorin kautta sen ollen helpoin vaihtoehto nopealle yksittäisten toiminnallisuuksien testaamiselle. Projektin laajamittainen testaus kuitenkin on syytä toteuttaa käyttäen joko itse luotuja testausluokkia tai editorista lisättäviä plugineja. Käyttämällä plug-inejä on mahdollista toteuttaa automaattista testausta. (Unreal Engine 2021g.)

Editorista on myös tärkeä tietää blueprint editor -näkymä. Blueprint editorissa blueprintteihin rakennetaan niiden toiminnallisuus, eli niihin lisätään tarvittavat komponentit, muuttujat sekä funktiot. Blueprint editorissa on kolme eri näkymää, jotka ovat Viewport, Construction Script sekä Event Graph. Viewportista näkee miltä blueprint tulee näyttämään sovelluksessa, eli kyseessä on blueprintin visuaalinen representaatio. Construction Script ja Event Graph ovat solmupohjaisia graafeja, joihin rakennetaan kyseisen blueprintin toiminnallisuus. Construction Script on hyvä työkalu alustamista varten, koska se suoritetaan sillä hetkellä,

kun kyseinen blueprint-luokka luodaan. Tämä tarkoittaa joko blueprintin lisäämistä maailmaan editorissa, tai sen syntymistä luodussa sovelluksessa. Event Graphiin saadaan lisättyä solmuja, jotka käynnistyvät jonkun asian eli eventin tapahtuessa. Eventtejä ovat esimerkiksi kollisio blueprintin sekä jonkun muun ympäristön toimijan tai esineen välillä, hiirellä klikkaaminen tai näppäimen painallus. (Romero & Sewell 2019.)

Epic Games Launcher on Epic gamesin oma distribuutiota varten tarkoitettu sovellus, jota kautta hallinnoidaan Epic Gamesin julkaisemia tai omistamia pelejä ja sovelluksia. Launcherista löytyy Unreal Engine Marketplace, josta löytyy Unreal Engineen käytettäväksi kolmannen osapuolen koodikirjastoja, komponentteja ja assetteja. Marketplace voidaan käynnistää myös editorin kautta. Marketplacen sisältö on joko ilmaista tai maksullista. Tekoälyyn liittyvää sisältöä on sadoin kappalein. Marketplacestä löytyvän sisällön lisääminen omaan projektiin on hyvin helppoa ja nopeaa. (Cordone 2019.)

3 Tekoäly Unreal Engine 4 pelimoottorissa

Tekoäly koostuu tekoälyagenteista, sekä tekoälyagenttien ympäristöstä. Unreal Enginessä tekoälystä puhuttaessa on tärkeä ymmärtää, että tekoälyagentteja voi olla samassa pelimaailmassa monia erilaisia, jotka käyttävät omia järjestelmiään. Tekoälyagentteja voi olla myös monia instansseja samasta tekoälyagentista. Saman tekoälyagentin instanssit voivat olla toisistaan hyvinkin erilaisia, mikäli tekoälyagentin infrastruktuuri on rakennettu sellaiseksi. Terminä tekoälyagentin ympäristö ei ole vakiintunut, joten puhuttaessa sen ympäristöstä voidaan tarkoittaa kaikkea pelimaailmassa olevaa ympäristöä, tai yksittäisen tekoälyagentin toimintaympäristöä.

4 Tekoälyagentit

Tekoälyagentti terminä on käsitteellinen yksikkö, joka tarkoittaa käytännössä sitä, että tekoälyagentti voi teoreettisesti olla täysin abstrakti olio. Tekoälyagentti on siis entiteetti, joka tekee päätöksiä toimintaympäristössään, jotka vievät sitä kohti tekoälyn tehtävän toteuttamista tai tavoitetta. Toimintaympäristöä tekoälyagentti havainnoi jonkun syötemekanismin avulla. (Lappalainen P 2020.) Usein kuitenkin tekoälyagenteista puhuttaessa pelimaailman kontekstissa tarkoitetaan hahmoa tai esinettä, joka toimii tekoälyä käyttäen. Tekoälyagentti tällaisessa tapauksessa tarkoittaa sitä kokonaisuutta, joka sisältää tekoälyn, sekä sen fyysisen olomuodon. (Unreal Engine 2021e.)

Unreal Engine 4 viiteympäristössä tekoälyagentti koostuu kokonaisuudesta, joka sisältää useita eri järjestelmiä ja komponentteja, vaikkakin teoreettisesti kaikki on mahdollista toteuttaa käyttäen vain blueprinttejä, tai itse luotuja c++-luokkia. Yleinen infrastruktuuri tekoälyagentille koostuu neljästä järjestelmästä. Näitä neljää järjestelmää voidaan miettiä tekoälyn kehona, sieluna, aivoina sekä muistina. Keho tarkoittaa pelaajalle tai ohjelman käyttäjälle näkyvää osaa tekoälyagentista. Sielu tarkoittaa agenttia ohjaavaa entiteettiä. Aivot tekevät tekoälyagenttia koskevat päätökset. Muisti säilyttää tekoälyn tarvitseman informaation. (Souza 2020.)

4.1 Tekoälyagentin keho

Tekoälyagentin keho eli pelaajalle näkyvä representaatio tekoälyagentista rakentuu Unreal Enginessä tekemälle sille oma blueprint. Riippuen siitä millaista tekoälyä ollaan rakentamassa, valitaan sille sopiva parent blueprint. Keholliselle tekoälyagentille parhaat vaihtoehdot ovat Pawn tai Character -blueprintit. Pawn on blueprint, joka on mahdollistaa ottaa kontrolloitavaksi käyttäen jotain kontrolleria. Character on enemmän toiminnallisuuksia sisältävä pawn blueprint, johon on lisätty agentin polygonmallia varten Static tai Skeletal mesh -komponentti, törmäystarkasteluun vaadittava kollisiokomponentti, sekä agentin liikkumisen kontrolloimista varten CharacterMovement-komponentti. (Unreal Engine 2021c.)

Mikäli toteutettavassa tekoälyssä tarvitaan näitä komponentteja, on syytä valita parent blueprintiksi character.

4.2 Tekoälynagentin sielu

Tekoälyn sieluna toimii Unreal Enginessä AIController -niminen järjestelmä. AIController on blueprint, joka ottaa haltuunsa tekoälyn kehon, liikuttaa kehoa hyödyntäen kehon liikkumisen komponenttia, sekä toimii linkkinä tekoälyagentin kehon ja aivojen välillä. Vaikka AIControlleria verrataan tekoälyagentin sieluun, olisi oikeampi kuvaus sille tehtäviensä perusteella keskushermosto. Unreal Enginen tapauksessa sielu usein ottaa vastuun tekoälyagentin aistien simuloinnista. (Unreal Engine 2021b.)

4.3 Tekoälyagentin aistien simulointi

Tekoälyagentin aistien simulointiin käytetään AIPerception tai Pawn Sensing -komponentteja. AIPerception on eritoten tekoälykäyttöön valjastettu komponentti, kun taas Pawn Sensing on yleisemmän tason havainnointiin tarkoitettu komponentti, jota voi käyttää myös hahmot, joilla ei ole tekoälyä käytössään. Haluttu komponentti liitetään osaksi AIControlleria, tai joissain harvinaisissa tapauksissa osaksi tekoälyagentin omaan blueprinttiin. AIPerception voidaan konfiguroida hyödyntämään joko näkö, kuulo tai tuntoaistia. AIPerception -komponentti mahdollistaa funktioden ja eventtien käynnistämisen aistien muutosten perusteella. Esimerkiksi voidaan luoda funktio, joka käynnistyy sillä hetkellä, kun komponentti havaitsee äänen tapahtuvan. Komponentin avulla voidaan myös hallita aistien voimakkuutta. Tällä voidaan vaikuttaa siihen, miten kauas agentti näkee tai kuulee. Komponentin avulla voidaan myös vaikuttaa aistialueeseen ja sitä kautta luoda efektejä, joissa tekoälyagentti esimerkiksi kuulee vain yhdestä suunnasta tulevat äänet. (Unreal Engine 2021d.)

4.4 Tekoälyagentin aivot

Tekoälyn aivoista puhuttaessa tarkoitetaan komponenttia tai järjestelmää, joka ohjaa tekoälyagenttien tekemiä päätöksiä. Muutkin tekoälyn osat kuin sen aivot

osallistuvat päätöksen tekoon, mutta enemmänkin informaatiota tuottavassa roolissa. Erilaisia lähestymistapoja päätöksenteon ohjaukseen on useita. Yksinkertaisimmillaan aivoina voi olla joukko ehtolausekkeita. Vaihtoehtona yleensä pelimoottoreissa on ehtolausekkeiden lisäksi äärellinen tilakone ja käytöspuu.

Sami Lahtinen (2020) kertoo opinnäytetyönsä teoriaosuudessa erilaisista tekoälyarkkitehtuureista peleissä. Tilakoneiden ja käytöspuiden lisäksi Lahtinen kertoo tavoite-, sekä hyötypohjaisesta tekoälyarkkitehtuurista. Nämä kaksi arkkitehtuuria ottavat vastaan itselleen kaikki mahdolliset tekoälyagentin tehtävät, ja erinäisten parametrien ja ehtojen kautta tekee päätöksen toteutettavasta tehtävästä. Erona näillä arkkitehtuureilla on se, että tavoitepohjainen arkkitehtuuri valitsee suoritettavaksi tehtäväkseen absoluuttisesti tehokkaimman tehtävän sen hetken kontekstissa, kun taas hyötypohjainen valitsee tehtävän, jolle ohjelmoija on asettanut suurimman numeerisen hyötyarvon. (Lahtinen 2020.)

Äärellisessä tilakoneessa on ennalta määritetty määrä erilaisia tiloja, joiden välillä siis tekoälyagentti vaihtelee kuitenkin siten, että vain yksi tila on aktiivisina kerrallaan. Tilat yleensä tarkoittavat tekoälyagentin sen hetken tekemistä, mielentilaa tai jotain vastaavaa, jonka muutoksilla halutaan vaikuttaa tekoälyagentin käytökseen. (Ögren 2020.)

Käytöspuu on näistä järjestelmistä raskain ja sen vahvuudet tulevatkin esille tekoälyn kompleksisuuden kasvaessa. Käytöspuuta voidaan miettiä nimensä mukaisesti puun muotoisena komponenttina. (Ögren 2020.) Puusta kasvaa oksia, eli käytöspuun haaroja. Näille oksille asetetaan painoarvo, joiden mukaan tekoäly suorittaa käytöspuuhun sijoitettuja tehtäviä. Suuremman painoarvon omaavat solmut pyritään suorittamaan joko ensimmäisenä, tai ainoana tehtävä riippuen käytöspuun konfiguraatiosta.

Unreal Enginessä näistä on yleisesti käytössä käytöspuu, vaikkakin tekoälyagentin päätöksiin voidaan vaikuttaa myös ehtolausekkeilla tekoälyagentin blueprintissä. Käytöspuu koostuu Unreal Enginessä taskeista eli tehtävistä, komposiittisolmuista, sekä edellä mainittuihin liitettävistä suplementeista. Suplementit ovat muihin solmuihin liitettäviä lisäosia, joilla tuodaan solmuille ehtoja tai ohjeita niiden suoritukseen.

4.4.1 Tekoälyagentin aivojen rakennuspalikat

Käytöspuu sisältää kolme erilaista komposiittia Selector, Sequence ja Simple parallel. Komposiitit ohjaavat käytöspuun läpikäyntiä. Selector suorittaa sen alle asetettuja tehtäviä ja komposiitteja, siten että se lopettaa haarojensa suorituksen sen jälkeen, kun solmu saa lapsisolmultaan onnistumisilmoituksen. (Unreal Engine 2021i.) Selector siis suorittaa lapsisolmujaan niin pitkään, että kaikki sen lapsisolmut ovat palauttaneet epäonnistumisilmoituksen tai siihen, että jokin lapsisolmuista palauttaa onnistumisilmoituksen. Käytännössä tämä tarkoittaa sitä, että prioriteetilla pienemmät haarat eivät koskaan aktivoidu, ellei ensimmäinen lapsisolmu palauta suorituksestaan epäonnistunutta tulosta.

Sequence suorittaa lapsisolmujaan niin kauan, että joko kaikki sen lapsihaarat on suoritettu tai jokin haaroista palauttaa epäonnistuneen tulokseen (Unreal Engine 2021i). Käytännössä siis päinvastoin kuin selector. Sequenceä voidaan ajatella nimensä mukaisesti sekvenssinä. Esimerkiksi tekoälyagentti voi ensimmäisessä lapsisolmussa laskea itselleen uuden sijainnin, johon liikkua. Seuraavassa solmussa liikutaan kyseiseen pisteeseen. Tällaisessa tilanteessa sequenceä käyttämällä voidaan asettaa toinen solmu ehdolliseksi ensimmäisen tehtävän onnistumisesta.

Simple parallel on komposiitti, jonka tarkoitus on mahdollistaa käytöspuun haarojen läpikäynnin kaksi haaraa kerrallaan. Toisen haaran on oltava vain yhden tehtävän mittainen. Hyvä esimerkki simple parallel -solmun käytöstä on tilanne, jossa tekoälyagentti liikkuu kohti sille määriteltyä maalipistettä ja sen liikkeen aikana vihollinen ampuu kohti pelaajaa keskeyttämättä liikkumista simple parallel -solmun käytöstä.

4.4.2 Tekoälyagentin aivojen ohjaus

Tekoälyagentin aivoja ohjataan komposiittien lisäksi Decorator ja Service -nimisillä solmujen suplementeilla. Suplementit liitetään komposiitti- tai tehtäväsolmuihin. Nämä suplementit tuovat lisämahdollisuuksia käytöspuun läpikäynnin ohjaamiseen. (Unreal Engine 2021h.)

Decoratoreita voidaan miettiä käytöspuun ehtolausekkeina. Decoraattoreita voidaan sijoittaa käytöspuuhun komposiittien yhteyteen rajoittamaan puun haarojen läpikäyntiä, siten muodostaen ehdollisia tehtäviä käytöspuun haaraksi. Tälläinen haara suoritetaan vain, mikäli decoratoriksi asetettu ehto toteutuu. (Unreal Engine 2021h.) Ehdoksi voidaan asettaa tekoälyn muistin sisältämän muuttujan arvo, tai jokin muu Unreal Enginen valmiiksi sisältämistä decoraattoreista. Ennalta rakennettuja decoraattoreita ovat esimerkiksi Cooldown, joka asettaa haaran epäaktiiviseksi käyttäjän määrittämäksi ajaksi, tai Loop joka asettaa haaran läpikäytäväksi tietyn määrän kertoja.

Servicet ovat aikaperusteisia suplementteja. Jos Service-suplementti on liitetty johonkin solmuun, suoritetaan kyseinen solmu aina tietyn ajan välein. Servicen toiminnallisuus on riippuvainen siitä, että kyseinen haara on suorituksen alla muutenkin. (Unreal Engine 2021h.) Servicejä käytetään lähinnä siis tekoälyagentin muistissa sijaitsevien arvojen seurantaan ja päivitykseen. Unreal Engine sisältää kaksi valmiiksi tehtyä Serviceä. Default Focus on suplementti, joka mahdollistaa tekoälyn muistissa olevan muuttujan välittämisen AIControllerille. Run EQS Query suorittaa ympäristötutkinnan spesifioidulla aikavälillä ja palauttaa ympäristötutkinnan tuloksen käytöspuun käytettäväksi. Servicejä voi myös tehdä itse. Yleinen käyttötarkoitus kustomoidulle servicelle on tekoälyagentin muistin muuttujien seuranta ja tarkastelu.

4.4.3 Tekoälyagentin tehtävät

Käytöspuu sisältää tehtäväsolmuja, jotka ovat käytöspuun suoritettavia funktiota. Yksinkertaisia tehtäviä on jo valmiiksi, kuten esimerkiksi MoveTo ja Wait -funktiot. Näillä yleisesti käytetyillä valmiilla tehtävillä tekoäly osaa liikkua ja suunnistaa kohti niille annettua maalipistettä tai maalikohdetta, tai odottaa tekemättä mitään. Muita valmiiksi Unrealista löytyviä tehtäviä ovat FinishWithResult, jolla voidaan lopettaa läpikäytävä haara ja palauttaa haluttu lopputulos. MakeNoise tekee äänen käyttäen tekoälyagenttia. Tähän ääneen voi muut agentit tai hahmot reagoida käyttämällä aikaisemmin mainittuja havainnointikomponentteja. PlayAnimation pyörittää halutun animaation tekoälyagentilla. RotateToFaceBBEntry muuttaa tekoälyagentin rotaatiota siten, että tekoälyagentti ”katsoo”

kohti tekoälyn muistissa olevaa muuttujaa. MoveDirectlyToward -tehtävä liikuttaa tekoälyagentin suoraa viivaa kohti maalipistettä tai maalikohdetta ja jättää kaiken navigaation täysin huomioitta. PushPawnActionilla voidaan valita joku Unreal Enginen sisältämä tehtävä, joka välitetään tekoälyagentin kontrollerilla ja pakotetaan kontrolleri suorittamaan kyseinen tehtävä. Tätä käytetään, jos tarvitaan referenssi suoritetusta tehtävästä tekoälyagentin kontrollerille. Run Behaviour -tehtävällä voidaan suorittaa yhdellä tehtävällä alikäytöspuuta. Käyttämällä tätä tehtävää voidaan käytöspuun kokoa pienentää ja sitä kautta helpottaa käytöspuun seurantaa ja muokkausta. Run Behaviour -tehtävästä on dynaaminen ja ei dynaaminen versio, joiden ero on se, että dynaamisessa versiossa tehtävän suorittamaa alikäytöspuuta voidaan vaihtaa toteutetun sovelluksen ollessa käynnissä, kun taas ei dynaamisessa versiossa suoritettava alikäytöspuu on lukittu siihen mikä se on sovelluksen käynnistyessä. (Unreal Engine 2021f.)

Monimutkaisemmat tehtävät tehdään itse luomalla. Itse tehtyjä tehtäviä ei ole rajoitettu määrällisesti tai laadullisesti, josta syystä käytöspuuhun saa tehtyä tarvittaessa todella raskaita ja kompleksisia funktiotakin. Raskaat ja kompleksiset funktiot ovat kuitenkin huono idea käytettäväksi käytöspuussa performanssiongelmien, sekä tehtävien epäonnistumisen hallinnan vaikeuden takia. Mikäli pitkä, raskas funktio pitää abortoida kesken sen käynnissä olemisen jää tekoäly tilaan, jonka huomioonottaminen muualla koodipohjassa voi tuottaa suuria hankaluuksia.

Itse luotavissa tehtävissä mahdollisia funktion aloittavia Eventtejä ovat Receive Tick, Receive Execute ja Receive Abort -eventit. Receive Tick tarkoittaa jokaista hetkeä, kun käytöspuu päivittyy. Recieve Execute Event käynnistyy, kun kyseistä tehtävää kutsutaan käytöspuusta käsin. Receive Abort Event käynnistyy silloin, kun käytöspuussa ylemmältä haaralta tulee keskeytyskäsky alemmalle haaralle. Kustomoiduissa tehtävissä on tärkeä muistaa lopettaa tehtävä käyttäen Finish Execute -solmua, koska muuten käytöspuu jää ikuiseen silmukkaan. Ikuinen silmukka tapahtuu, koska käytöspuun näkökulmasta kyseinen tehtävä on kesken. Tällä solmulla myös voidaan valita palauttaako tehtävä onnistuneen vai epäonnistuneen viestin käytöspuulle.

4.5 Tekoälyagentin muisti

Tekoälyn aivojen avuksi Unreal Engine tarjoaa muistin erikseen. Muistia vastaava komponentti on nimeltään Blackboard. Se sisältää kaikki käytöspuun tarvitsemat ja käyttämät muuttujat. Käytöspuusta voidaan lukea tai muuttaa Blackboardin sisältämiä muuttujia käyttäen tehtävien sisältämiä solmuja Get Blackboard value as <muuttujan tyyppi> ja Set Blackboard value as <muuttujan tyyppi>. Tällä tavoin voidaan esimerkiksi päivittää kohdetta, johon kohti tekoälyagentti liikkuu kesken pelin. (Unreal Engine 2021b.) Blackboardia voidaan täten kuvailla muuttujia sisältävänä tietokantana, jota käytöspuu hyödyntää.

4.6 Tekoälyagentin järjestelmien yhdistäminen

Tekoälyagentin kokonaisarkkitehtuuri rakentuu seuraavanlaisesti. Tekoälyagentin blueprint linkataan AIControllerille valitsemalla blueprintissä AIController Class oikeaksi AIControlleriksi. AIController yhdistetään käytöspuuhun kontrollerin Event Graphissa. Tämä tapahtuu käyttäen solmua Run Behaviour Tree, johon valitaan kyseisen tekoälyagentin käytöspuu. Käytöspuun Details-valikosta valitaan oikea Blackboard. Jos kansiorakenteessa on samassa kansiossa vain yksi käytöspuu ja yksi Blackboard yhdistyvät ne automaattisesti. Yksinkertaisimmillaan tekoälyagentin eri järjestelmien linkitys tapahtuu näin yksinkertaisesti. (Souza 2020.)

5 Tekoälyn ympäristö

5.1 Ympäristössä navigointi

Unreal Enginessä Tekoälyagentin reitin etsinnälle ja navigoinnin konfiguroinnille käytetään NavMesh nimistä navigaatiokomponenttia. NavMesh perustuu open-source pohjaiseen navigaatiokomponenttiin nimeltä ReCast. NavMesh koostuu NavMeshBoundVolumesta, Navigaatiolle relevanteista blueprintestä, ympäristön itsenäisistä komponenteista, sekä NavMeshBoundVolumen välisistä linkeistä. (Zieliński 2015.)

NavMeshBoundVolume on nelikulmio, jonka peittämän alueen sisälle tekoälyagentin mahdolliset liikkumispinnat ja reitit lasketaan. Mikäli tekoälyagentin ympäristö ei ole navigaationelikulmion sisällä ei silloin myöskään tekoälyagentin reitinlaskenta toimi, eikä agentti osaa liikkua kyseisessä ympäristössä. On mahdollista käskeä agentti liikkumaan myös paikkoihin, jotka eivät ole NavMeshBoundVolumen alaisia, mutta koska reitinlaskenta ei ole käytössä voi agentti jäädä jumiin, mikäli suorimman reitin edessä on este. NavMeshBoundVolumeja on mahdollista olla yhdessä maailmassa useampi ja niiden väliset linkit ovatkin tarkoitettu siihen, että kaksi tai useampi NavMeshBoundVolumea voidaan yhdistää toisiinsa (Zieliński 2015). NavMeshBoundVolume ei osaa ottaa huomioon lentäviä, tai maan alla liikkuvia agentteja, koska se laskee vain maan pinnalla kulkevat reitit.

Navigaatiolle relevantit blueprintit ja ympäristön itsenäiset komponentit ovat navigoitavassa ympäristössä sijaitsevia asioita, jotka vaikuttavat navigaatiolaskentaan. Ne voivat toimia joko esteinä, jolloin niiden lävitse tai päältä ei voi navigoida, tai osana navigoitavaa aluetta.

5.2 Ympäristön tutkinta

Ympäristön tutkintaan Unreal Enginessä käytetään Environment Query Systemiä eli EQS-järjestelmää. Systeemiä käytetään keräämään dataa ympäristöstä ja pisteyttämään ympäristöä erilaisten testien perusteella. Nämä testit voivat esimerkiksi mitata etäisyyttä tekoälyagentista pelaajan hahmoon tai sijaintiin, joka on talletettu vektorimuuttujaan. Testejä voi suorittaa useita päällekkäin yhdessä tiedustelussa. Ympäristön tutkinnan testi pisteyttää ja laskee parhaan tai parhaat sijainnit, toimijat tai rotaatiot, jotka se on laskenut käyttäen haluttuja parametrejä. EQS on järjestelmänä vielä eksperimentaalinen. Järjestelmää käyttääkseen täytyykin käydä asettamassa se editor preferences -asetuksista päälle. (Lowing 2017.)

6 Työskentelyn koordinointi toimeksiantajan kanssa

Toimeksiantajana opinnäytetyössä toimii Kollektiwe Oy. Työskentelen opinnäytetyön parissa yhdessä yrityksen kanssa läheisessä kontaktissa. Opinnäytetyö toteutetaan jo olemassa olevaan peliprojektiin. Peliprojektin kokonaisuuden suunnittelusta vastaa toimeksiantaja, mutta tekoälyn toteutuksen suunnittelen pääosin itse.

Opinnäytetyö toteutettiin kokonaan etätyöskentelyn menetelmillä, joten koordinointiin käytettiin yrityksen käyttämää kommunikaatiosovellusta. Sovelluksen avulla pystyttiin suunnittelun ja osan toteutuksen apuna käyttämään näytön jakamista.

Opinnäytetyö toteutettiin erillisenä pakettina irralleen muusta projektista. Tähän päädyttiin kahden syyn takia. Ensimmäinen syy on projektin jo valmiiksi suuri koko. Suuret projektit hidastavat editorin toimintaa, jota kautta työskentelynopeus eritoten toiminnallisuuksien testauksen parissa olisi kärsinyt. Toinen syy on se, että peliprojekti on VR-sovellus, ja henkilökohtaisesti en omista VR-laitteita. Tekoälyn toiminallisuus on täysin samanlainen VR-laseja käyttäessä kuin normaalissa sovelluksessakin, joten tekoälyä rakentaessa sovelluksen tyypillä ei ole merkitystä.

7 Aaltopohjaisen vihollisgeneraattorin toteuttaminen tekoälyn näkökulmasta

7.1 Vaatimukset

Opinnäytetyön vaatimukset kokonaisuuden osalta on saada aikaan pelillinen paketti, jota on mahdollista käyttää demokäytössä. Opinnäytetyön ulkopuoliset pelilliset mekaniikat, joita demoversioon tarvitaan, on joko jo toteutettu tai toteutetaan opinnäytetyön ohessa nopeimmalla mahdollisella tavalla. Projektiin halutaan kaikkialle sattumanvaraisuutta, joka näkyy myös tekoälyssä, ja niiden synnyttämisessä.

Tekoälyagenttien infrastruktuuri tulee olla sellainen, että tekoälyagentin visuaalinen proseduraalinen rakentaminen myöhemmin on mahdollista. Projektista löytyy jo asegeneraattori, jolla rakennetaan pelaajan käyttämä ase proseduraalisesti osista. Tekoälyagenttien infrastruktuurin tulee seurata samaa mallia projektin yhtenäisyyden säilyttämiseksi, sekä tulevan työn helpottamiseksi.

Vihollisten synnyttämisen tulee toimia aaltopohjaisesti, jolla demoversiossa on alku ja loppu. Aalto loppuu kaikkien vihollisagenttien tultua tuhotuksi. Jokaisen aallon tulee olla edellistä vaikeampi. Aaltorakenteeseen täytyy rakentaa logiikka, jossa joka kymmenes aalto sisältää pomovihollisen. Pomoa itsessään ei kuitenkaan vielä toteuteta.

7.2 Peliprojektin valmiiden osien hyödyntäminen

Opinnäytetyön kattaman osuuden ollessa osa suurempaa peliprojektikokonaisuutta täytyy se ottaa huomioon suunnitellessa tekoälyn toteutusta, ettei päällekkäisiä ominaisuuksia tulisi. Peliprojektin sisältämiä ominaisuuksia, joita tulee omassa toteutuksessani huomioida ovat pelaajahahmo, pelaajan käyttämät aseet, sekä niiden sisältämät logiikat kuten vahinkopisteet, sekä projektiilin liikkumisen komponentit. Hyödyntämällä projektin jo valmiita komponentteja säästetään aikaa tekoälyn ulkopuolisten ominaisuuksien toteuttamiselta, jotka täytyisi testaamista varten joka tapauksessa tehdä.

Projekti asettaa myös rajoitteita tekoälyn toteuttamiselle. Tekoälyagenttien tulee pystyä hyödyntämään Unreal Enginen sisältämää vahinkolaskentaa, tekoälyagenttien alustaminen tulee tapahtua yhden blueprintin sisällä ja koko tekoälyn kokonaisuus tulee toteuttaa käyttäen Unreal Enginen pääasiallisia tekoälyjärjestelmiä, joita on tässä työssä käsitelty. C++-luokkia luodaan vain tapauksissa, jossa toiminnallisuutta ei saa toteutettua blueprintein. Vaatimukset toteutukselle asettaa projektin aikaisempi toteutus, suunniteltu jatkokehitys sekä toimeksiantajan toivomukset.

7.3 Tekoälyagentin alustaminen ja personointi

Toimeksiantaja toivoo tekoälyagenttien olevan persoonallisia hahmoja, jotka ovat toisistaan erilaisia käytökseltään ja toiminnoiltaan. Personointi tapahtuu alustamalla hahmot ennen niiden synnyttämistä maailmaan erilaisin muuttujin. Nämä muuttujat vaikuttavat tekoälyagenttien käyttäytymiseen joko suorasti hahmon sisältämien komponenttien attribuuttien arvojen muuttamisella, tai vaihtoehtoisesti vaikuttamalla käytöspuun läpikäyntiin. Kaikkeen alustamiseen halutaan kontrolloitua sattumanvaraisuutta, jossa sattumanvaraisuudelle annetaan joko tarkat reunaehdot, tai sitä ohjataan kohti haluttua lopputulosta kuitenkaan lopputulosta täysin määrittämättä.

Haluttuun lopputulokseen päästään suunnittelemalla agenttien eroavaisuudet ja ne muuttujat, joilla näihin eroavaisuuksiin päästään. Agenteille arvotaan ennen niiden syntymistä agenttityyppi, arkkityyppi, taito sekä liikkumistapa. Agentin alustaminen tapahtuu tekoälyagentin Blueprintin Construction Graphissa. Sieltä kutsutaan kaikkia alustamiseen luotuja funktiota, jonka seurauksena agentti alustetaan sattumanvaraiseksi juuri ennen sen syntymistä maailmaan.

7.3.1 Agenttien personointiin käytettävät muuttujat

Agentit vaativat useita eri muuttujia, että niistä saadaan persoonallisia. Yksittäisen muuttujan lisääminen tekoälyagentille ei vielä tee tekoälyagentista persoonallista, vaan muuttujia tarvitaan enemmän. Agenttien erot saadaan aikaan seuraavilla muuttujilla.

HP tarkoittaa elämäpisteitä, eli kuinka paljon vahinkoa agentti kestää. Vihollisagentin HP arvosta vähennetään aina sen ottaessa vahinkoa, joka tarkoittaa opinnäytetyössä sitä, kun pelaaja osuu ammuksella tekoälyagenttiin. HP:n arvon tipahtaessa nollaan agentti poistetaan maailmasta.

Speed määrittää tekoälyagentin liikkumisnopeuden. Liikkumisnopeus välitetään suoraan alustamisen yhteydessä Character Movement -komponentille, joka ohjaa tekoälyagenttien liikkumista NavMeshillä. Komponentti voi vaikuttaa vain

NavMeshillä kulkeviin tekoälyagentteihin, joten lentäville agenteille liikkumisnopeus välitetään käytöspuun sisältämiin tehtäviin käytettäviksi.

Damage, Attack Cooldown, Attack Range ja Projectile Speed ovat tekoälyagentin hyökkäykseen vaikuttavia muuttujia. Damage tarkoittaa tekoälyagentin hyökkäyksien aiheuttamaa vahinkoa pelaajaa kohtaan. Attack Cooldown tarkoittaa aikaa tekoälyagentin hyökkäyksien välillä. Attack Range tarkoittaa etäisyyttä, jonka sisältä agentti voi hyökätä kohti pelaajaa. Projectile Speed on muuttuja, jolla määritetään agenttien hyökkäyksien projektiilien nopeus. Damage ja Projectile Speed välitetään agentin ampuman projektiilin blueprintille. Siitä eteenpäin vahinkoa applikoidaan pelaajahahmoon käyttämällä Apply Damage -solmua Event Hit –eventin yhteydessä, jossa varmistetaan, että projektiilin osunut toimija on pelaajan hahmo. Projectile Speed välitetään tekoälyagentin projektiilin laukausemiseen tarkoitetulle blueprintille. Attack Cooldown, sekä Attack Range välitetään käytöspuulle käytettäväksi. Attack Cooldownia käytetään Wait Blackboard Time -tehtävän parametrinä, kun taas Attack Rangelle rakennetaan itse tehty service, jolla seurataan, että onko vihollinen edennyt riittävän lähelle pelaajaa.

Dodge Chance määrittää prosentuaalisen mahdollisuuden agentin väistöliikkeelle. Tekoälyagentti voi väistää pelaajan ampumia ammuksia sillä hetkellä, kun ammus ammutaan. Dodge Chance välitetään käytöspuun tehtävälle, jossa väistäminen toteutetaan.

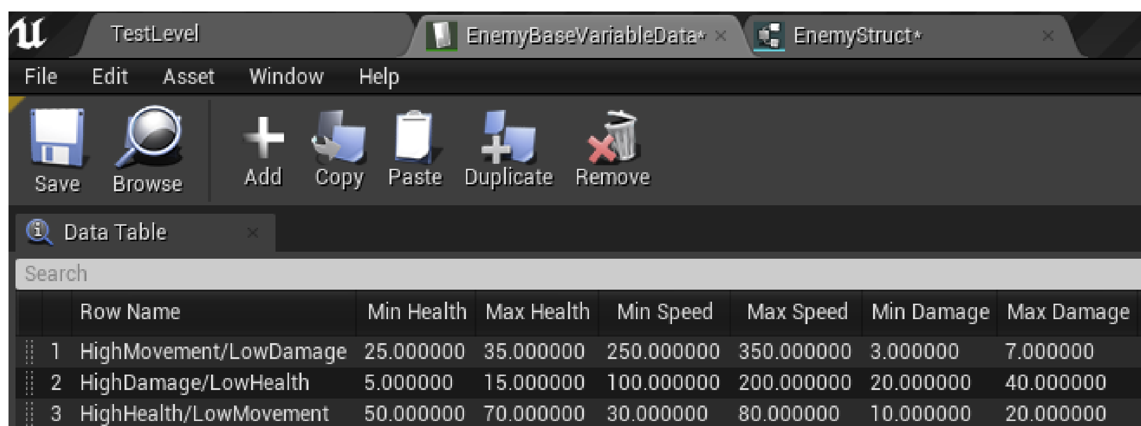
Muuttujien arvojen perusteella tekoälyagentti pisteytetään 1-5 pisteen arvoiseksi. Tätä pisteytystä käytetään vihollisaaltojen vaikeusasteiden kontrollointiin.

7.3.2 Agenttityypit

Agenttityyppejä toteutettiin kolme erilaista. Niiden tehtävä on rajata käytössä olevien muuttujien ala- ja ylärajat, ja tuoda vihollisagentteihin efektiä, jossa agenteilla on selkeästi vahvuudet ja heikkoudet. Niiden on tarkoitus myös toimia

sattumanvaraisuuden rajoitteena, etteivät agentit ole liian eri tasoisia vaikeusasteeltaan. Agenttityyppien implementointi pyrkii mahdollisimman samaan lopputulokseen, kuin erillisten blueprinttien tekeminen erilaisille vihollisille.

Agenttityypit toteutetaan hyödyntäen Datatablea (kuva 1) ja Structia agenttien muuttujien ylä- ja alarajojen vaihtamisen ja visualisoinnin yksinkertaistamiseksi. Datatable on tietokantataulu, jonka rivirakenne tulee jostain valmiista rakenteesta, eli tässä tapauksessa itse luodusta structista. Struct on joukko muuttujia tai dataa, joka on yhdistetty käytön helpottamiseksi yhdeksi kokonaisuudeksi eli structiksi.



	Row Name	Min Health	Max Health	Min Speed	Max Speed	Min Damage	Max Damage
1	HighMovement/LowDamage	25.000000	35.000000	250.000000	350.000000	3.000000	7.000000
2	HighDamage/LowHealth	5.000000	15.000000	100.000000	200.000000	20.000000	40.000000
3	HighHealth/LowMovement	50.000000	70.000000	30.000000	80.000000	10.000000	20.000000

Kuva 1. Agentin muuttujia Unreal Enginen datatablessa.

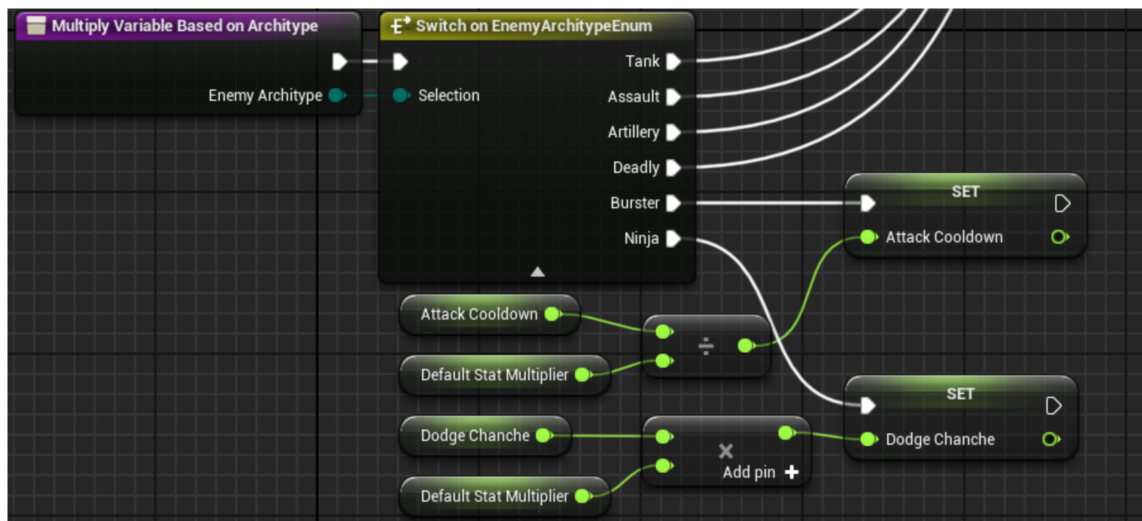
Taulukossa sijaitsevaa tietuetta on helppo lukea ja editoida. Toinen vaihtoehto olisi ollut tehdä kaikki muuttujat blueprintin sisään, joka periaatteessa poistaisi yhden askeleen agentin rakentumisesta, mutta kaiken jatkotyöskentelyn kannalta helpoin tapa on käyttää Datatablea.

Datatable rakentaa automaattisesti sarakkeet jokaiselle structin sisältämälle muuttujalle. Structiin lisättiin kaikki aiemmin mainitut muuttujat kahteen kertaan ylä- ja alarajoja varten. Datatableen lisättiin jokaista suunniteltua agenttityyppiä kohden yksi rivi. Rivien täyttäminen oikeilla arvoilla tehdään datatablessa, koska structissa on vain yhden rivin tietueet, joten jos structissa vaihdetaan jotain arvoa, muuttuu kaikkien rivien arvot.

7.3.3 Arkkityypit

Arkkityyppi on Enumeraattori-muuttuja, jonka tarkoituksena on saada lisättyä yksittäisen agentin vahvuuksia ja sitä kautta tuoda agenttien eroavaisuuksia esille. Enumeraattori on muuttujatyyppi, jota voidaan kuvailla eräänlaisena ennalta täytettynä listana. Esimerkiksi enumeraattori voisi sisältää arvojoukkonaan viikonpäivät, jolloin enumeraattorin arvo olisi tämän hetken viikonpäivä. Enumeraattori ei voi sisältää muita muuttujatyyppejä vaihtoehtoinaan, eikä sen arvojoukkoa voi muokata sovelluksen suorituksen aikana.

Arkkityyppejä toteutettiin kuusi kappaletta, vastaamaan kaikkia muita muuttujia paitsi projectile speedia. Sille ei toteutettu omaa arkkityyppiä, koska vihollisten ampumien projektiilien balansointi siten, että pelaaja voi väistää niitä on hyvin hankalaa eritoten jatkokehityksen kannalta erilaisten projektiilien ollessa mahdollisia. Alustamisvaiheessa vihollisen arkkityypin perusteella kerrotaan tai jaetaan arkkityyppiä vastaavan muuttujan arvoa käyttämällä enumeraattoreille hyvin sopivaa Switch on -solmua (kuva 2).



Kuva 2. Käyttämällä Switch -solmua voidaan edetä ohjelmistossa eri suoritushaaroihin enumeraattorin arvon perusteella.

Opinnäytetyössä arkkityyppi arvotaan jokaiselle yksittäiselle viholliselle. Arkkityyppi kuitenkin halutaan jatkokehityksessä sitoa aaltoihin siten, että arkkityyppien on tarkoitus vaikuttaa 10 aallon ajan ja vaihtua jokaisen pomoaallon yhteydessä luoden efektiä, jossa pomo vaikuttaa omiin alaisiinsa. Esimerkiksi 10

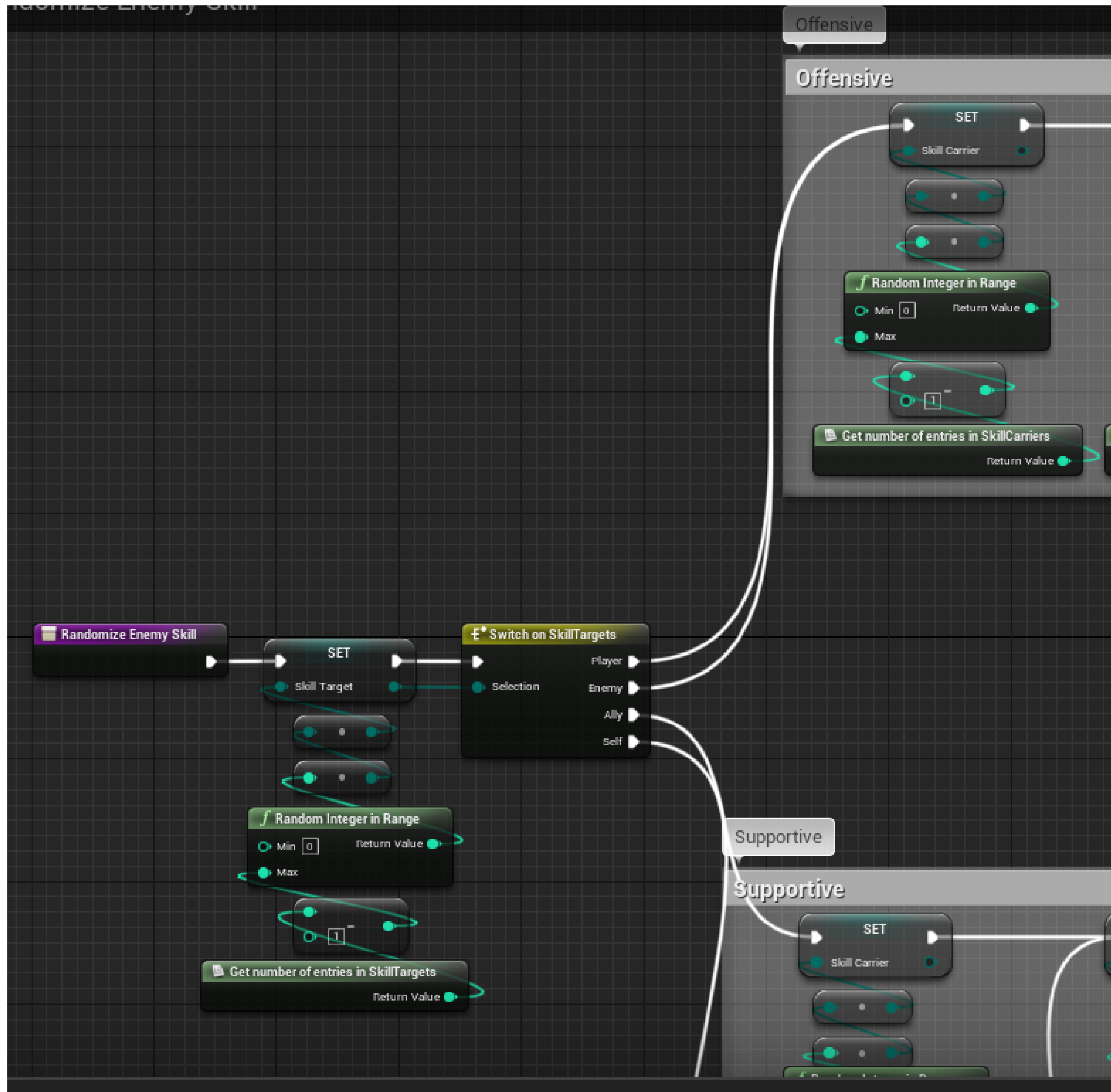
aallon ajan voidaan antaa arkkityypiksi ”Assault”, joka tarkoittaisi jokaisen tekoälyagentin liikkumisvauhdin kaksinkertaistamista sen ajan, kun kyseinen arkkityyppi on aktiivisena.

7.3.4 Taidot

Jokaisella tekoälyagentilla on yksi sattumanvaraisesti arvottu taito. Taidot voivat olla hyökkäyksellisiä tai puolustuksellisia taitoja. Hyökkäyksellinen taito voi olla esimerkiksi normaalista poikkeava hyökkäys, kuten ohjautuva tai räjähtävä projektiili. Puolustuksellinen taito voi olla esimerkiksi vahingoittuneen vihollisagentin vahinkopisteiden poistoa. Taidot toteutetaan vain arkkitehtuurillisella tasolla, eli jokaiselle tekoälyagentille arvotaan taito alustamisen yhteydessä, mutta itse taitoja ei toteuteta.

Taito koostuu taidon kohteesta, taidon kantajasta, taidon efektistä, taidon vaikutusajasta, sekä taidon koosta. Edellä mainituista jokaisesta luotiin oma enumeraattori-muuttujansa. Nämä muuttujat koostettiin yhdeksi structiksi, jotta saadaan enumeraattorit yhdistettyä yhdeksi kokonaisuudeksi. Enumeraattoreita käytetään toteutuksessa, koska halutaan muuttuja, jolla on jo valmiiksi rajoitetut vaihtoehdot arvolle. Listaa tai taulukkoa ei käytetä, koska enumeraattoreilla on helpompi rajoittaa kombinaatiota. Enumeraattori on myös hyödyllinen taitojen tapauksessa niiden ollessa hyviä työkaluja käytöspuun läpikäynnin ohjaukseen. Taitojen tapauksessa käytöspuulle tarvitaan eri haaroja eri taitojen käyttöä varten, sillä jos taidon käyttö olisi yksi tehtävä käytöspuussa, tulisi se erittäin raskaaksi ja pitkäksi tehtäväksi.

Arkkitehtuurillisesti taidon kohde arvotaan ensin ja sitä käytetään rajoittamaan mahdollisuuksia taidon koostumukselle (kuva 3). Tämä tehdään siksi, ettei taidolle tule yhdistelmiä, jossa esimerkiksi kohde olisi pelaaja mutta efektinä olisi Hitpointsien palauttaminen, tai vastaavasti agenttia itseään vahingoittava taito. Taidon kohteen perusteella mennään koodissa hyökkäävään haaraan, tai puolustavaan haaraan. Jatkokehityksessä rajoitteita tulee olemaan enemmän, jolloin mahdollisesti jokaiselle eri vaihtoehtoiselle kohteelle täytyy tehdä oma haaransa, jossa rajoitetaan mahdollisia kombinaatiota.



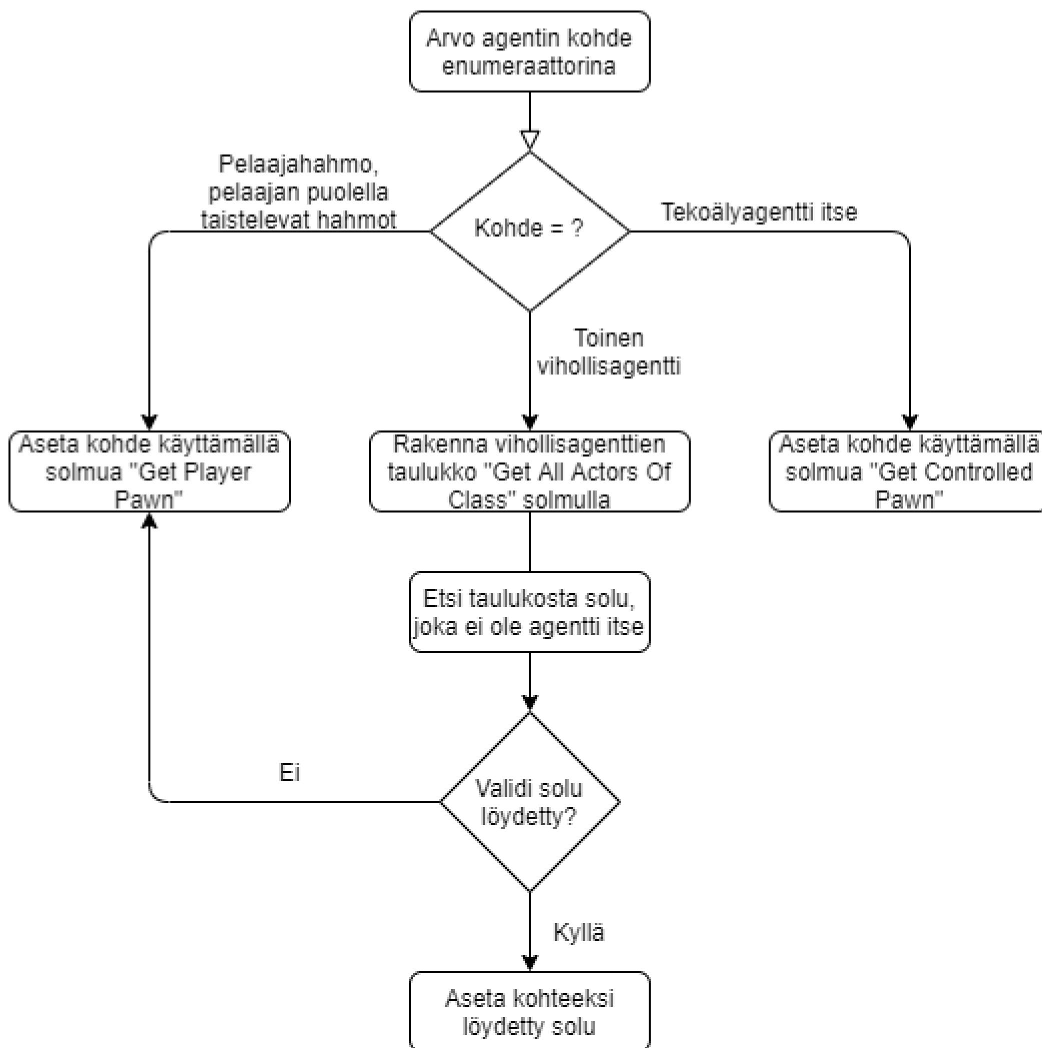
Kuva 3. Tekoälyagentin taitojen kombinaatiota rajoitetaan sen kohteen perusteella.

Tekoälyagentin taidon kohde täytyy konvertoida enumeraattorista actor-tyyppiseksi muuttujaksi, jotta sitä voidaan käyttää käytöspuun tehtävissä antamaan referenssi taidon kohteesta. Actor tarkoittaa jotain objektia, joka voidaan synnyttää tai asettaa maailmaan. Käytännössä kyseessä on siis yleensä jokin blueprint tai itsenäinen komponentti. Muita enumeraattoreita ei tarvitse konvertoida, koska ne eivät ole suoraan toiminnallisuuteen kytköksissä vaan niitä käytetään käytöspuun ohjauksessa.

Konvertointi tapahtuu itse luodulla funktiolla tekoälyagentin AIControllerissa. Funktio alkaa Switch -solmulla, jossa tarkastellaan enumeraattorin arvoa. Jos kohde on pelaaja, pelaajan puolella oleva hahmo, tai vihollisagentti itse on kohteen konvertointi hyvin helppoa. Koska pelaajan puolella olevia hahmoja ei

opinnäytetyön puitteissa toteuteta, asetetaan tekoälyagentin kohteeksi silloin pelaaja. Pelaajan asettamiseksi taidon kohteeksi käytetään Get Player Character -solmua, joka antaa suoraan referenssin pelaajan kontrolloimaan hahmoon. Jos vihollisen kohde on se itse, silloin referenssi itseensä saadaan käyttämällä solmua Get Controlled Pawn, joka antaa kyseisen blueprintin kontrolloiman hahmon referenssin. Koska funktiota suorittaa tekoälyagenttia kontrolloiva AIController, on silloin solmun antama tulos tekoälyagentti itse.

Taidon kohteen ollessa joku toinen vihollisagentti on se hieman monimutkaisempi. Ensin otetaan Get All Actors of Class -solmulla kaikki olemassa olevat instanssit vihollishahmon blueprintistä ja asetetaan ne taulukkoon. Taulukkoa käydään lävitse For Each Loop with Break -silmukalla, jossa yhdessä silmukan iteraatiossa tarkastellaan ensin, ettei kyseessä ole kyseinen funktiota suorittava tekoälyagentti. Seuraavana tarkistetaan, onko käsiteltävä taulukon alkio validi, eli onko tarkasteltava tekoälyagentti olemassa. Mikäli taulukko ei sisällä validia kohdetta, tarkoittaa se sitä, että vihollishahmoja ei ole olemassa. Täten kohteeksi voidaan asettaa pelaaja, ettei tekoälyagentti jää täysin toimettomaksi. Funktio käynnistetään Event Begin Playssä, koska silloin taulukon tarkistus toimii oikein. Get All Actors of Class ei voi palauttaa instansseja vihollisista, joita ei vielä ole. Kyseinen solmu ei palauta haetun luokan instansseja missään tietyssä järjestyksessä, joten taulukon järjestys on jokaiselle eri agentille eri. Muuten ensimmäisen validin alkion sijasta täytyisi ottaa joko sattumanvarainen alkio, tai toteuttaa pidempää logiikkaa tekoälyagentin taidon kohteen valintaan. Kuviossa 1 on kuvattu konvertoinnin logiikka vuokaaviota käyttäen.



Kuvio 1. Vuokaavio taidon kohteen konvertoinnista actor-tyyppiseksi muuttujaksi

7.3.5 Agentin visuaalisuus

Agentin visuaalinen ulkomuoto tullaan rakentamaan myöhemmin proseduraalisesti, joten ajan säästämiseksi opinnäytetyössä käytetään tekoälyagentin keholle Unreal Enginen sisältämiä valmiita yksinkertaisia polygonimalleja eli meshejä. Tuleva visuaalisuuden toteutus kuitenkin täytyi ottaa huomioon. Vihollisten tuleva rakentaminen otettiin arkkitehtuurillisesti huomioon siten, että kaikki vihollisagentit alustetaan ja rakennetaan yhden blueprintin sisällä. Kaikki vihollisagentit pelimaailmassa ovat siis instansseja kyseisestä blueprintistä. Jatkossa kehon eri osat tulevat olemaan omia blueprinttejä tai komponenttejaan, jotka kasataan yhdeksi kokonaiseksi kehoksi tekoälyagentin alustamisen funktioiden yhteydessä.

Agenteille kuitenkin opinnäytön puitteissa toteutettiin fyysisen koon eli Unreal Enginen tapauksessa skaalan muuttamista, jota muutetaan agentille arvotun HP arvon mukaisesti. Mitä enemmän HP:ta agentilta löytyy, sen suurempi sen skaala on.

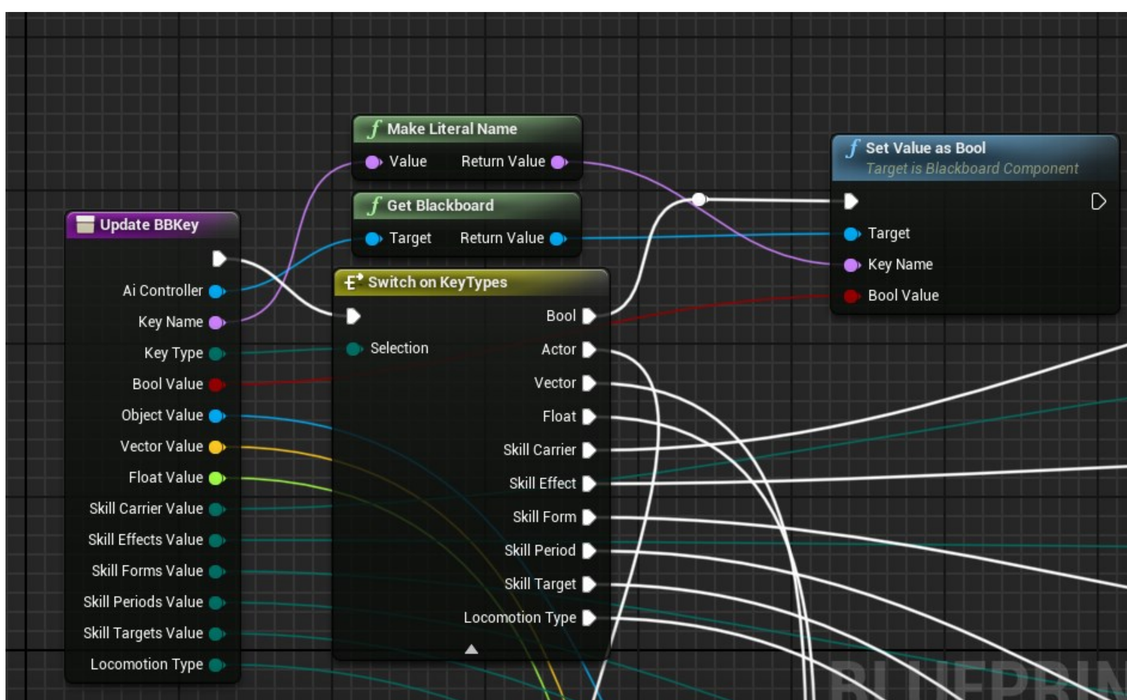
7.3.6 Agentin liikkumistyyppi

Tekoälyagentin alustamiseen kuuluu myös sen liikkumistyypin arpominen. Agentti voi olla joko maata pitkin kulkeva, teleporttia hyödyntävä tai lentävä agentti. Kuten muussakin alustuksessa käytetään vaihtoehtoihin enumeraattori-muuttujaa. Lentävän agentin tapauksessa tarvitaan lisäalustusta, koska lentävä agentti toteutetaan hyödyntämällä Unreal Enginen fysiikanmallinnus järjestelmää. Sitä varten fysiikan simulointi tulee asettaa tekoälyagentin kollisiokomponentilta päälle, sekä asettaa Linear- ja Angular Damping -solmuilla vastustusta käytöspuun tehtävässä lisättävälle työntövoimalle. Gravitaatio asetetaan lentävältä agentilta pois päältä, koska se on huomattavasti helpompi tapa saada toteutettua lentävä agentti kuin se, että tekoälyagentille annettaisiin jatkuvaa alaviistosta suuntautuvaa työntövoimaa. Lopputulos on myös uskottavamman näköinen, sekä vähemmän altis ohjelmointivirheille. Maata pitkin kulkevalla ja teleporttia hyödyntävällä agentilla fysiikkaa ei simuloida ja gravitaatio on voimassa, joten niillä ei tarvita lisäalustusta.

7.3.7 Agentin järjestelmien alustaminen

Agentin alustamiseen kuuluu myös tekoälyagentin käyttämien järjestelmien linkitys. Tekoälyagentin käyttämät järjestelmät linkitetään tässä projektissa kappaleen 4.6 osoittamalla tavalla. Sen lisäksi agentin alustamisen aikana arvotut muuttujat välitetään agentin muistille käyttämällä siihen itse luotua funktiokirjastoa. Unreal Enginen funktiokirjaston sisältämiä funktioita voidaan miettiä globaaleina funktioina. Funktiokirjasto toteutettiin, koska tässä peliprojektissa tekoälyagentin muistin muuttujia päivitetään monesta eri paikasta moneen kertaan.

Kyseistä globaalia funktiota varten tehtiin enumeraattori, joka sisältää kaikki tekoälyagentin muistin sisältämät muuttujat. Funktio sisältää Switch On -solmun, jonka parametrinä luotu enumeraattori toimii. Täten siis enumeraattorin arvon perusteella voidaan viedä funktion toteutushaara eri pisteisiin (kuva 4). Set Value as <muuttujatyyppi> on solmu, joka tekee itse Blackboardin muuttujan arvon päivittämisen. Kaikki muu ympärillä vain mahdollistaa funktion kutsumisen muualta koodissa ja lyhentää sen yhden solmun mittaiseksi. Toimiakseen funktio tarvitsee myös referenssin AIControlleriin, johon käytöspuu ja Blackboard on yhdistetty. Set Value as <muuttujatyyppi> tarvitsee myös päivitettävän muuttujan nimen, joten sekin täytyy tuoda koko funktion parametrinä solmulle.



Kuva 4. Funktiossa käytettävä Set Value As -solmu määritetään Enumeraattorin arvon perusteella.

7.4 Tekoälyagenttien synnyttäminen pelimaailmaan

Tekoälyagenttien syntymiselle vaatimukset olivat seuraavanlaiset. Syntyminen tapahtuu aaltopohjaisesti, jokaisen aallon ollessa edellistä vaikeampi pelaajalle. Joka kymmenennen aallon tulee olla pomoaalto. Koska opinnäytetyön jälkeisen demoversion halutaan olevan pelattava, tarvitaan sille alku ja loppupiste. Alkaminen voi tapahtua näppäimellä kutsuttavasta Eventistä. Tekoälyagenttien syn-

nyttämisen infrastruktuuri tulee olla sellainen, että jatkossa aaltojen väliin voidaan lisätä aikaa, jolloin mitään ei tapahdu. Aaltojen tulee sisältää tietyissä rajoissa olevaa sattumanvaraisuutta agenttien määrän suhteen.

7.4.1 Aaltopohjaisuuden toteuttaminen

Aaltoja on kahden tyyppisiä, normaaliaaltoja ja pomoaaltoja. Pomoja ei ole suunniteltu, eikä niitä toteuta muuten kuin logiikan pohjalta, että milloin sellainen syntyy. Toimeksiantaja suunnitteli aaltorakenteen siten, että joka kymmenes aalto on pomoaalto. Toimeksiantaja myös suunnitteli aaltojen vaikeustasojen-vaikeutumisen tason (kuva 5).

hahmojen vaikeuspisteet:				1	5	
Alkuvaikeuspisteet				20		
vaikeuspisteiden nousu			1,2	10		
wave	vaikeuspisteet		Minimiagentit		Maksimiagentit	
1	20		4	-	20	
2	34		7	-	34	
3	50,8		10	-	51	
4	70,96		14	-	71	
5	95,152		19	-	95	
6	124,1824		25	-	124	
7	159,0189		32	-	159	
8	200,8227		40	-	201	
9	250,9872		50	-	251	
10	311,1846		62	-	311	

Kuva 5. Aaltojen vaikeustason nouseminen kaavalla x = x*1.2 + 10

Kehittäjän näkökulmasta katsottuna kyseinen kaava tulee vielä jatkossa vaihtumaan agenttien maksimimäärän suuruuden takia. 250 agenttia pelimaailmassa yhtä aikaa aiheuttaa performanssiongelmia, vaikka tekoälyagenttien renderöintietäisyyden kanssa pelaisikin aggressiivisesti. Tietysti muitakin tapoja taistella performanssiongelmia vastaan on, esimerkiksi yhtäaikaisten tekoälyagenttien määrän rajoittaminen yhden aallon sisällä. Käytännössä siis toteuttaa aaltoja aaltojen sisälle. Open World -tyyppisessä pelissä myös tekoälyagenttien asettaminen inaktiiviseksi olisi toimiva ratkaisu. Kuitenkin tämän peliprojektin kaltaisessa aaltopohjaisessa ammuskelupelissä tekoälyagentit etenevät aina kohti

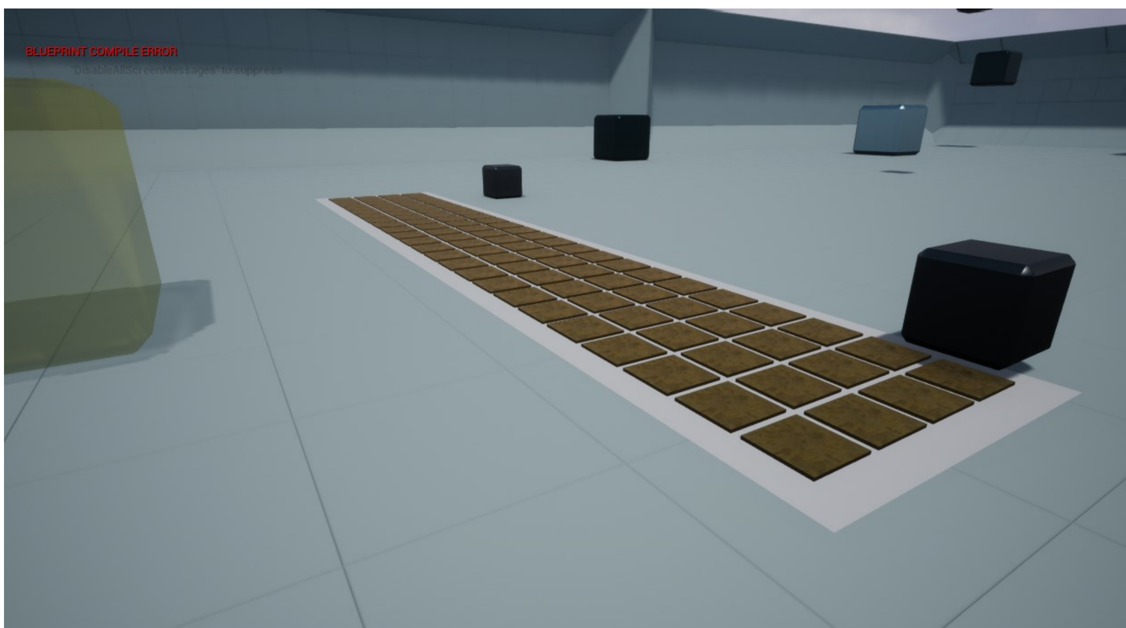
pelaajaa syntymisensä jälkeen, joka tarkoittaa sitä, että tekoälyagentin pitää olla aktiivinen voidakseen suunnistaa kohti pelaajaa, vaikka kyseinen agentti ei vielä pelaajalle edes näkyisi.

Vihollisten synnyttämiselle rakennettiin oma blueprint, jossa kaikki aaltojen synnyttämiseen ja seurantaan liittyvä toiminnallisuus toteutetaan. Vihollisagenttien syntymisen kontrollointiin rakennetaan spawnpointit, eli syntymispisteet. Spawnpointteja käyttämällä ja siihen visuaalisuuden lisäämisellä saadaan efektiä viemäreistä nousevista rotista, joka toimi syntymisen inspiraationa. Käyttämällä spawnpointteja myös vältetään vihollisten liian lähekkäin syntyminen.

Vihollisten syntyminen tapahtuu käyttämällä Spawn Actor -solmua, joka synnyttää yhden toimijan kerrallaan. Tälle solmulle täytyy siis rakentaa silmukka, jotta saadaan synnytettyä useampi vihollisia samanaikaisesti. Vaihtoehtoina silmukalle ovat For, For Each ja While -silmukat. For ja For Each -silmukat olisivat hyviä vaihtoehtoja vihollisten synnyttämiseen, mikäli aallot olisivat jo ennalta täytettyjä. Sattumanvaraisuuden toteuttamiseksi While-silmukka toimii parhaiten. Tekoälyagentteja synnytetään hyödyntäen tekoälyagenttien pisteytystä. Aalloilla on matemaattisella kaavalla laskettu vaikeuden osoittava pistemäärä, josta arvotun tekoälyagentin synnyttäminen vähentää pisteitä. Kun nämä pisteet osuvat nollaan silloin tiedetään aallon kaikkien vihollisten syntyneen. Koska tekoälyagenteilla on jo ennalta määritetty pisteytys, voi silmukka mennä myös alle nolla-arvon yhden agentin arvon verran, mutta tällaisessa tilanteessa, jossa aallot ovat sattumanvaraisuuden alaisia ei sillä ole väliä. Jos yhden ylimääräisen agentin syntyminen olisi suurempi ongelma, täytyisi tekoälyagentin syntymisen yhteydessä rajoittaa arvottavan tekoälyagentin muuttujien alustamista. Kyseinen tapa toteuttaa muuttaisi infrastruktuuria niin paljon, että koko opinnäytetyön arkkitehtuuri jouduttaisiin miettimään uudelleen, koska silloin tekoälyagentin alustus tulisi suorittaa synnyttämisen logiikan jälkeen. Tämä tarkoittaa sitä, että vihollinen pitäisi synnyttää ensin ja alustaa sen jälkeen sopimaan synnyttävän blueprintin asettamien määreiden mukaiseksi.

7.4.2 Syntymispisteiden määrittäminen

Tekoälyagentin spawnpointeille tulee tehdä oma blueprint, koska niiden lisäämiseksi synnyttämisen blueprintin päälle (kuva 6) tarvitsee se tehdä joko manuaalisesti komponentteja lisäämällä blueprintin viewportissa, tai dynaamisesti synnyttämällä blueprinttejä toisen blueprintin päälle. Näistä huomattavasti vähemmän ongelmallinen on synnyttää Begin Play -eventissä spawnpointteja synnyttämisen blueprintin päälle. Laskemiseen hyödynnetään Unreal Enginen 2D Grid Execution Macroa, jolla voidaan laskea itse syöttämiä parametrejä vastaava ruudukko 2-ulotteiselle pinnalle. Spawnpointtien laskemisen ja synnyttämisen yhteydessä lisätään ne taulukkomuuttujaan, jota hyödynnetään tekoälyagenttien synnyttämisessä. Tekoälyagenttien synnyttämisen sijainniksi asetetaan taulukon sisältämä sattumanvarainen spawnpoint.

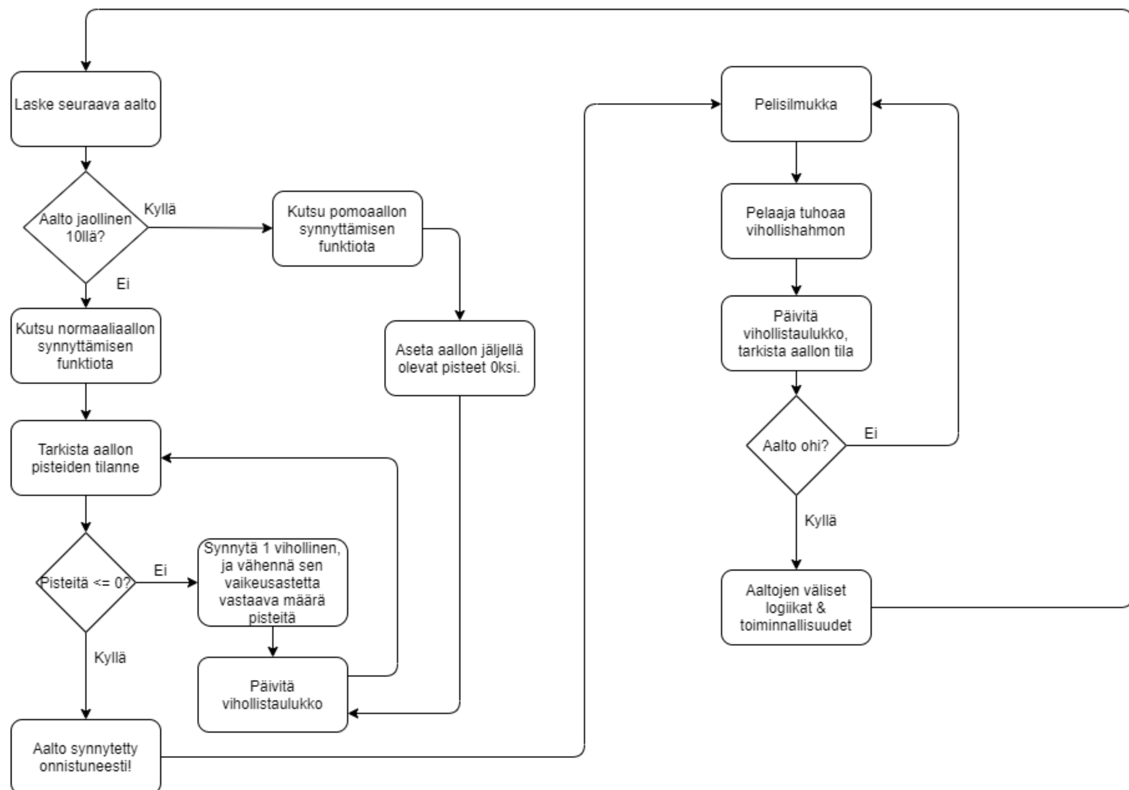


Kuva 6. Pelimaailman näkymä vihollisia synnyttävästä blueprintista, jonka päällä spawnpointteja.

7.4.3 Vihollisaaltojen hallinta

Vihollisaaltojen hallintaan käytetään actor-tyypin taulukkomuuttujaa. Vihollishahmon syntyessä lisätään taulukkoon referenssi agentista. Vihollishahmon kuollessa poistetaan hahmon referenssi taulukosta. Taulukon hallinnan ollessa toteutettu oikein, voidaan taulukkoa käyttää aaltojen tilan tarkasteluun. Taulukkoa

tarkastelemalla voidaan identifioida aaltojen päättymishetki ja rakentaa logiikka seuraan aallon alkamiselle. Ohjelman optimisoimiseksi taulukkoa tarkastellaan aina vihollisen kuollessa. Täten ei tarvitse suorittaa jatkuvia turhia tarkasteluja, joka parantaa ohjelmiston suorituskykyä. Kuvio 2 kuvaa vihollisaaltojen synnyttämisen ja seurannan logiikan.



Kuvio 2. Agenttien synnyttämisen ja aaltopohjaisuuden rakentamisen vuokaavio.

7.5 Tekoälyagenttien tehtävät

Opinnäytetyössä tekoälyagenttien päätoiminen tehtävä on olla vihollishahmoja pelaajalle. Pelaajan tehtävänä on eliminoida tekoälyagentit, joten tekoälyagenttien tehtävänä on tehdä se mahdollisimman mielenkiintoiseksi pelaajalle. Pelissä ei toistaiseksi ole tarinaa, joten tarinavetoisuutta ei synny. Täten pelin on oltava mielenkiintoinen pelimekaanikoiltaan, jotta pelaajan mielenkiinto säilyy. Tekoälyagenttien tehtävä on keskeinen, sillä pelin pääasiallinen sisältö on vihollishahmojen tuhoaminen. Vihollisagenttien vastaava tehtävä on tuhota pelaaja, ja tehdä heidän itsensä tuhoamisensa hankalaksi. Tavoitteeseensa tekoälyagentti pääsee erilaisin käytöspuun sisältämin tehtävin.

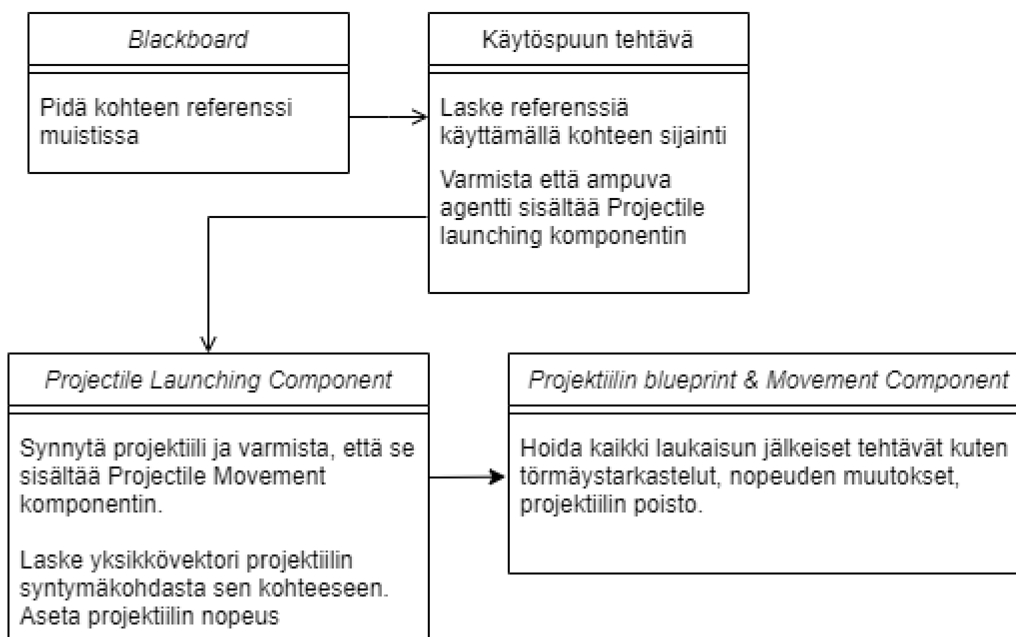
7.5.1 Hyökkäys pelaajaa kohtaan

Tekoälyagenttien kollektiivinen tehtävä on tuhota pelaaja. Toteuttaaksensa tehtävänsä tekoälyagenteille rakennettiin hyökkäys, jolla tekoälyagentit voivat tuhota pelaajan. Tekoälyagentin hyökkäys on projektiili, joka ammutaan pelaajaa kohti. Projektiilille luotiin oma blueprinttinsä, jossa sille lisättiin Projectile Movement -komponentti, jolla kontrolloidaan projektiilin liikkumista. Opinnäytetyössä komponentilla kontrolloidaan projektiilin aloitusnopeutta, maksiminopeutta, kimmoisuutta, sekä siihen kohdistuvan gravitaation voimaa. Muuttamalla näitä asetuksia saatiin projektiilit liikkumaan pelaajaa kohti siten, että niitä on mahdollista väistää.

Projektiilin laukaisuun toteutettiin erillinen blueprint, koska projektiili ei ole olemassa ennen kuin vihollinen ampuu sen. Jos projektiilia ei ole olemassa se ei myöskään voi synnyttää itseään. Blueprint luodaan siten, että se periytyy SceneComponent-nimistä parent blueprintistä, koska silloin voidaan käyttää tätä blueprinttiä toisen blueprintin sisällä komponenttina. Komponentti lisätään ampuvalle vihollisagentille. Toistaiseksi komponentti sisällytetään kaikille vihollisagenteille, mutta jatkossa uudenlaisia hyökkäyksiä lisätessä vaihdetaan toiminnallisuus siten, että komponentti lisätään synnyttämisen yhteydessä, mikäli vihollisen hyökkäys on projektiilityyppinen.

Yhden ammuksen ampuminen on siis monen eri järjestelmän ja blueprintin yhteistyötä (kuvio 3). Hyökkäys käynnistetään käytöspuuhun luodulla tehtävällä. Tehtävä varmistaa, että ampuva tekoälyagentti sisältää ampumiseen käytettävän komponentin, sekä vastaanottaa tekoälyagentin referenssin blackboardilta. Tehtävä laskee ampuvan tekoälyagentin sijainnin pelimaailmassa referenssin avulla. Tehtävästä kutsutaan projektiilin laukausemisen komponentin sisältämää funktiota. Funktiossa synnytetään itse projektiili, sekä varmistetaan että projektiili sisältää projektiilin liikkumisen komponentin. Syntymisen jälkeen projektiilista otetaan yksikkövektori kohti kohdetta ja kerrotaan vektori projektiilin halutulla vauhdilla. Tulokseksi tuleva vektori asetetaan projektiilin nopeudeksi. Projektiili ja projektiilin sisältämä projektiilin liikkumisen komponentti hoitavat lopun toiminnallisuuden. Osumistarkastelu toteutetaan projektiilin blueprinttiin käyttäen Event Hit -eventtiä, jossa voidaan tarkastaa mihin projektiili osui. Projektiilin osuessa pelaajan aiheutetaan pelaajalle vahinkoa samalla tavalla kuin

vihollishahmoihinkin, eli käyttäen Apply Damage -solmua ja vastaanottamalla eventillä AnyDamage.

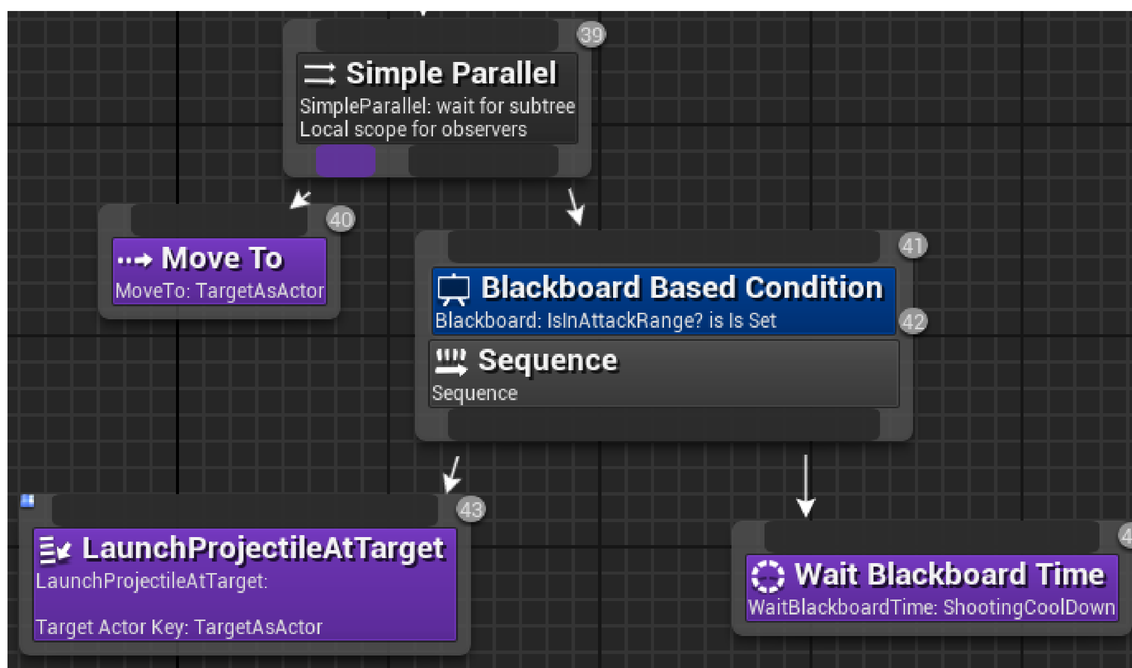


Kuvio 3. Projektiilin ampumisen kronologinen järjestys kuvattuna eri järjestelmien tehtävien kautta.

Tekoälyagenttien tulee liikkua samalla kun ne ampuvat kohti pelaajaa, joten käytöspuussa käytetään Simple Parallel -komposiittia, jossa prioriteetilla suurempi haara liikuttaa tekoälyagenttia kohti pelaajaa. Pienemmän prioriteetin haaran läpikäynti asetettiin ehdolliseksi käyttämällä decoraattoria. Tätä decoraattoria tarkastellaan käytöspuun aivan ylimpänä asiana itse luodulla servicellä puolen sekunnin välein. Servicessä tarkastellaan tekoälyagentin ja tekoälyagentin kohteen etäisyyttä ja asetetaan decoraattori tuloksen mukaan. Decoraattori tarkastelee siis tekoälyagentin ja pelaajan hahmon etäisyyttä, jotta agentti tietää onko se riittävän lähellä hyökätäkseen. Toteuttamalla tarkastelu servicellä estetään agenttia ampumasta attack rangensa ulkopuolelta tilanteessa, jossa tekoälyagentti joutuu syystä tai toisesta attack rangensa ulkopuolelle jo kerran siellä käytyään.

Sequence-komposiittia käytetään hyökkäysten Cooldownin aikaansaamiseksi. Suoritettuaan laukaisutehtävän etenee käytöspuu sekvenssin toiseen solmuun, joka on Wait Blackboard Time. Tällä solmulla odotetaan Blackboardin sisältämän float-tyyppisen muuttujan osoittama aika. Tekoälyagentin alustamisvaiheen Attack Cooldown muuttuja välitettiin blackboardille, jota hyödynnetään tässä

Wait Blackboard Time -solmun parametrinä. Täten saadaan aikaan hyökkäysten välille aikaa, ilman että tekoälyagentti keskeyttää liikkumista. Kuvassa 7 nähdään miltä logiikka näyttää käytöspuussa.



Kuva 7. Yhtäaikaisen liikkeen ja ampumisen logiikka.

7.5.2 Pelaajan hyökkäysten väistäminen

Tekoälyagentin väistämisliikkeiden toteuttamiseen projektissa oli kaksi vaihtoehtoa. Toinen vaihtoehto on piirtää suoraa viivaa pelaajan aseen piipusta ja tarkastella osuuko viiva vihollishahmoon ja sen seurauksena toteuttaa logiikka, jolla väistäminen tapahtuu. Tämän tyyppisellä toteutuksella tekoälyagentti väistää, kun pelaaja osoittaa aseellansa kohti tekoälyagenttia. Väistäminen tällaisessa tapauksessa ei siis ole sidonnainen pelaajan ampumiseen, vaan tähtäämiseen. Toinen vaihtoehto on käyttää funktiota Predict Projectile Path by Trace Channel projektiilin synnytyksen hetkellä. Kyseisellä funktiolla lasketaan projektiilin todennäköinen lentorata, jonka kohdatessa tekoälyagentin kanssa ilmoitetaan siitä tekoälyagentille ja suoritetaan jatkologiikka väistämiselle. Käytännössä nämä toteutustavat eroavat siten, että toisessa väistäminen tapahtuu pelaajan osoittaessa aseella vihollisagenttia, kun taas toisessa pelaajan ampuessa agenttia kohden. Näistä vaihtoehdoista päädyttiin toteuttamaan jälkimmäinen, jossa siis väistäminen tapahtuu pelaajan ampuessa. Väistämistä varten

hyödynnetään tekoälyagentille arvottua muuttujaa Dodge Chance, jonka arvo kertoo prosentuaalisen todennäköisyyden sille, että tekoälyagentti suorittaa väistöliikkeen pelaajan ampuessa. Maassa liikkuvien agenttien väistämissuunta tapahtuu aina pelaajasta katsottuna suoraan vasemmalle tai oikealle, kun taas lentävällä agentilla mahdollisuus on myös väistää korkeussuunnassa ylös tai alas. Itse väistöliike on käytännössä tekoälyagentin sijainnin muuttaminen toiseen sijaintiin, johon lisätään visuaalisuutta käyttäen Unreal Enginen partikkelijärjestelmää. Sijainnin muutos tapahtuu välittömästi, joten seurauksena väistöliike on teleportin kaltainen.

7.6 Agentin liikkuminen

Tekoälyagentin liikkumiselle toteutettiin kolme eri vaihtoehtoa, maassa liikkuminen, maassa liikkuminen jossa lisänä teleportin käyttö, sekä lentävä liikkuminen. Tekoälyagentin taidon kohde määrittää mitä kohti tekoälyagentti liikkuu.

7.6.1 Maassa liikkuminen

Maassa liikkuvien agenttien liikkuminen tapahtuu käyttäen Unrealin NavMeshiä. NavMeshiä käyttävä agentti saa liikkumisen tarvittavat asetukset Character Movement -komponentilta, joka sijaitsee tekoälyagentin blueprintissä. Komponentti vastaa tekoälyagentin liikkumisen nopeudesta johon tekoälyagentin muuttujalla Speed haluttiin vaikuttaa. Alustamisen yhteydessä kyseinen muuttuja välitetään Character Movement -komponentin attribuutiksi.

Maassa liikkuvat agentit käyttävät liikkumiseen Move To -tehtävää, joka tarvitsee toimiakseen kohteen, jota kohti liikkua. Tehtävän kohde voi olla sijainti vektorimuodossa, tai actor-muuttuja. Move To -tehtävä NavMeshin kanssa yhteistyössä hoitaa reitinlaskennan kokonaisuudessaan. Tehtävä liikuttaa agenttia suorinta mahdollista reittiä kohti kohdettaan. Kun suuri määrä agentteja liikkuu samalla tavoin kohdin pelaajaa, saattaa silloin agenttien liikkuminen tuntua epäluonnolliselta.

Epäluonnollisuuteen vaikutetaan eri liikkumistavoilla, liikkumisnopeuksilla sekä tekoälyagenttien eri kohteilla. Epäluonnollisuuteen voidaan vaikuttaa myös käyttämällä parametrinä vektorina annettavaa sijaintia ja jakamalla liike kohti lopullista kohdetta useampaan toimintaan (kuva 8). Vektori parametrinä voi periaatteessa olla mikä vain, joten liikkeen jakaminen useaan eri iteraatioon on helpohkoa. Vektoriparametri täytyy laskea etukäteen, joten sille tuli toteuttaa oma tehtävänsä. Tehtävässä tarkastellaan agentin ja kohteen välistä vektoria, jaetaan se lyhyemmäksi ja asetetaan vektorin kahdelle muulle arvolle mahdollisuus sivuttaiseen liikkeeseen. Lisää sattumanvaraisuutta liikkeeseen saadaan hyödyntämällä GetRandomPointInNavigableRadius-solmua, jolla pisteen ympärille voidaan laskea alue, jolta valitaan sattumanvaraisesti uusi piste.

Kuva 8. Pilkkomalla agentin Move To -tehtävä saadaan rikottua tekoälyagentin epäluonnolliselta tuntuva täysin suora liike.

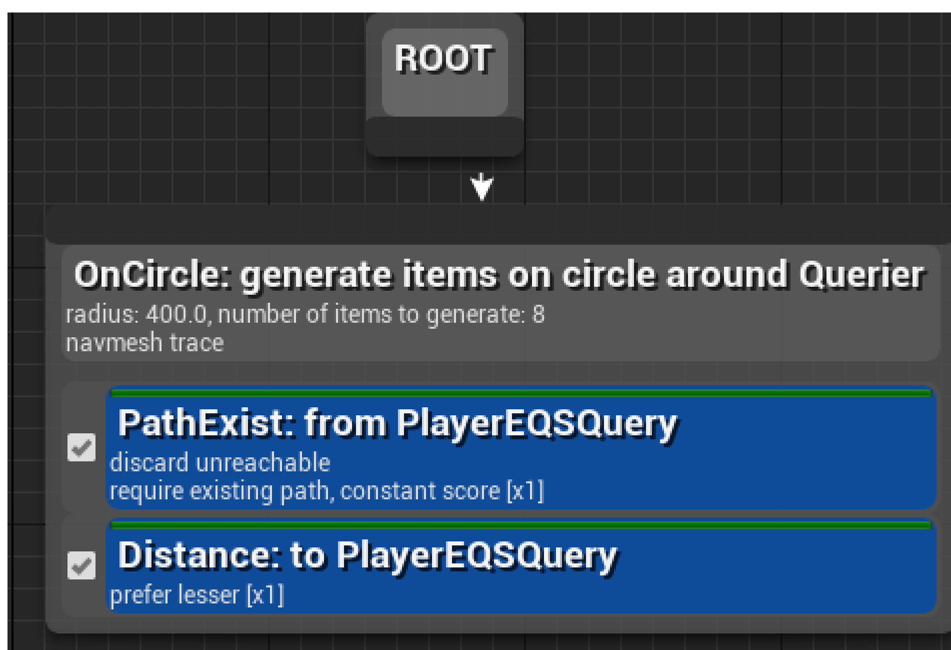
7.6.2 Teleportin käyttäminen

Teleporttia hyödyntävä agentti käyttää liikkumiseen samoja järjestelmiä ja solmuja kuin normaali maassa liikkuva agenttikin. Niiden lisäksi teleporttiin käytetään EQS-järjestelmää ja kahta lisätehtävää käytöspuussa.

Ympäristöntutkinnan keskipistettä kutsutaan kontekstiksi ja tässä opinnäytetyössä tarvitaan kontekstiksi myös pelaajan hahmo, joka tapahtuu luomalla EnvQueryContext_BlueprintBase-nimisen blueprintin pohjalta oma blueprint. Tämän blueprintin tehtävä on toimittaa konteksti ympäristötutkinnan tiedusteluille tai testeille. Se sisältääkin funktiot tehtävänsä toteuttamiseen. Blueprintissä ylikirjoitetaan ProvideSingleActor-funktio, ja asetetaan Get Player Character -solmulla funktion palauttamaksi kontekstiksi pelaajan hahmo.

Tiedustelulle rakennetaan testattavat pisteet parametrien avulla. Toteutettuun tiedusteluun asetettiin parametreiksi ympyrän muoto, 8 testattavaa pistettä sekä ympyrän kooksi 400 unreal enginen mittayksikköä. Testattavien pisteiden kontekstina toimii testiä suorittava tekoälyagentti itse. Teleportin haluttu toiminnallisuus määrittää tiedustelun parametrit, koska testattavat pisteet toimivat teleportin mahdollisina kohdesijainteina.

Yksi tiedustelu sisältää kaksi testiä, joiden perusteella mahdolliset teleportin sijainnit pisteytetään (kuva 9). Molemmat näistä testeistä käyttävät kontekstinaan pelaajahahmoa. Ensimmäinen testi tarkastaa, että mahdollisesta sijainnista on olemassa reitti pelaajan hahmon luokse. Tämän tarkoituksena on varmistaa se, ettei vihollisagentti jää jumiin sijaintiin mistä se ei voi edetä kohti pelaajaa. Kaikki pisteet, joista ei ole mahdollista navigoida kohti pelaajaa hylätään. Mikäli kaikki pisteet olisivat sellaisia mistä navigointi pelaajan hahmoon ei onnistuisi, hylättäisiin koko tiedustelu ja jätettäisiin teleportti toteuttamatta. Jälkimmäinen testi pisteyttää sijainnit niiden sijainnin perusteella. Mitä lähempänä pelaajaa piste on sen parempi.



Kuva 9. EQS järjestelmällä toteutettavat tiedustelut ovat ulkonäöltään ja käyttölogiikaltaan hyvin samankaltaisia käytöspuun kanssa

Käytöspuussa tiedustelua kutsutaan Run EQS Query -tehtävällä. Tehtävän parametrinä sille annetaan suoritettava tiedustelu sekä blackboardin muuttuja, johon tiedustelun tulos palautetaan. Potentiaalisia sijainteja tutkiessa palautuksen kohteeksi valitaan vektorimuuttuja. Käytöspuulle tarvitaan myös itse teleportin toteuttava tehtävä, jolle ympäristötutkinnan tulos välitetään. Teleportti toteutettiin samalla tavalla kuin väistöliike sillä erolla, että tekoälyagentin rotaatio korjataan aina osoittamaan kohti pelaajaa teleportin yhteydessä käyttämällä Set Actor Rotation -solmua. Oikea suunta rotaatiolle löydetään käyttämällä Find Look At Rotation -solmua. Teleportin aikana tekoälyagenttiin on lähes mahdotonta osua ammuksella, joten teleportille asetetaan aikarajoite sen käytön välille käyttämällä Wait-solmua käytöspuussa.

7.6.3 Lentäminen

Lentävän agentin liikkumiseen ei voida hyödyntää NavMeshiä, koska se ei suoraan tue ilmassa liikkumista. Ainoa tapa kiertää NavMeshin toimimattomuus lentävillä agenteilla olisi luoda lentäville agenteilla maata pitkin kulkeva näkymätön komponentti, jota käytettäisiin kaikkeen liikkumisen laskentaan. Tässä törmätään ongelmaan, jossa tämä näkymätön komponentti ei havaitse pelaajalle nä-

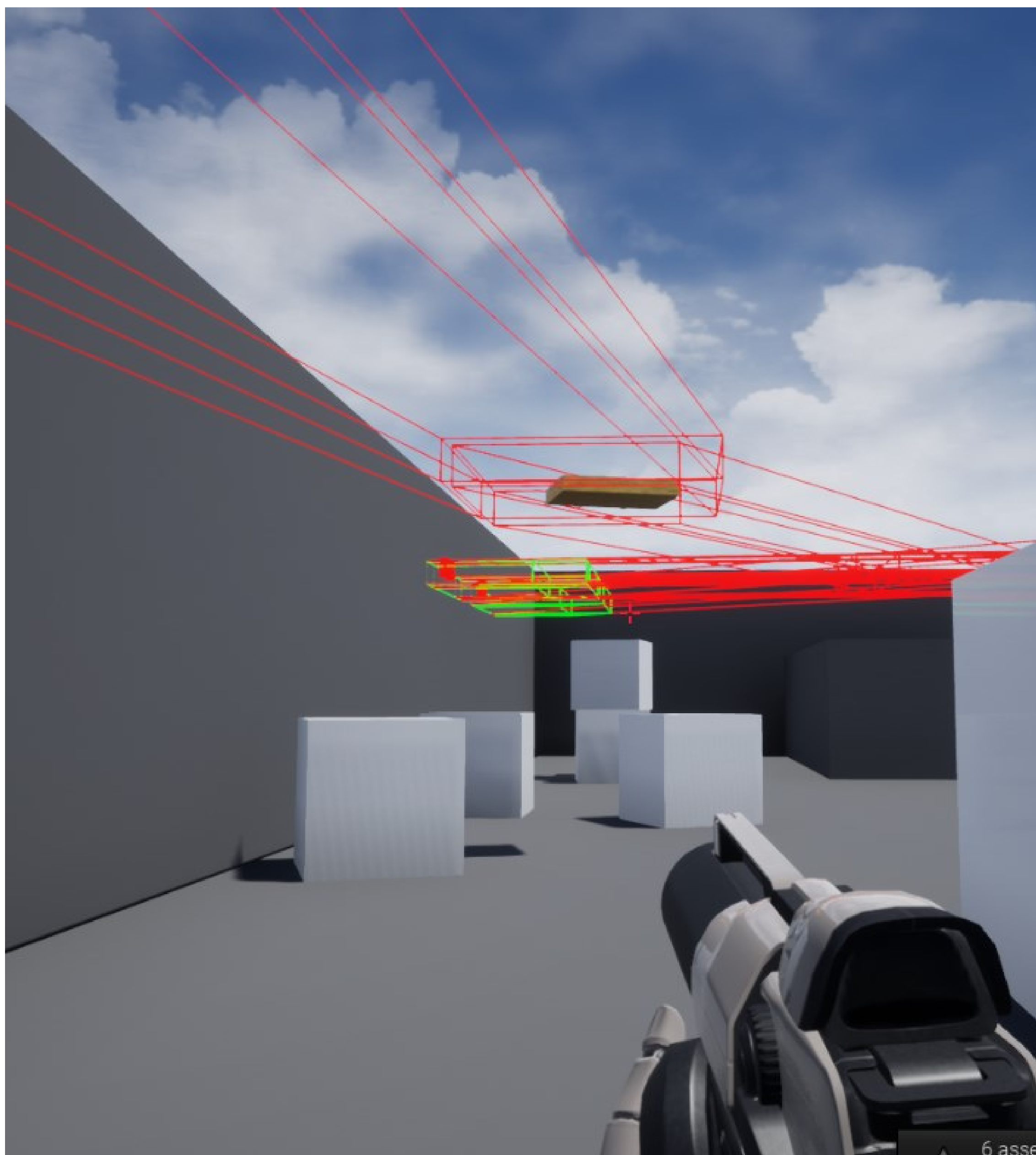
kyvän kehon edessä olevia esteitä, jolloin pelaajalle näkyvä osa tekoälyagentista kulkisi seinien lävitse tai jäisi niiden taakse jumiin riippuen agentin asetuksista. Täten lentävän agentin liikkuminen täytyi toteuttaa itse tai käyttämällä Unrealin marketplacesta löytyvää komponenttia. Päädyttiin toteuttamaan itse, koska toteutushetkellä komponentissa oli yhteensopivuusongelmia uuden Unreal Enginen version kanssa.

Lentävälle tekoälyagentille liikkumiseen tarvittavat tehtävät täytyy toteuttaa itse. Move To –tehtävää ei voida käyttää agenteilla, jotka eivät navigoi NavMeshiä pitkin. Lentävälle tekoälyagentille täytyy siis toteuttaa tehtävä kohteen laskemiseen, sekä sitä kohti liikkumiseen. Kohteen laskeminen lentävälle agentille on huomattavasti monimutkaisempaa kuin muiden liikkumistyyppien kohteiden laskenta, koska lentävän agentin tapauksessa kohteen laskemisen yhteydessä täytyy varmistaa se, että agentin ja kohteen välinen lentorata on esteetön. Lentoradan mahdollisuus pitää varmistaa, koska fysiikkapohjaiselle lentävälle tekoälyagentille esteiden väistämisen toteuttaminen on hyvin hankalaa ja toteutuksen skaala kasvaisi liian suureksi opinnäytetyössä toteutettavaksi.

Käytännössä lentämisen kohteen valitsemisen tehtävä toteutetaan siten, että se vastaanottaa parametreinä lentokorkeuden, agentin kohteen, sekä etäisyyden pienimmän ja suurin sallitun arvon kohteesta, johon päästessä tekoälyagentin liike on onnistunut. Vektorilaskentaa hyödyntämällä lasketaan potentiaalinen liikkeen loppusijainti tekoälyagentin kohteen läheltä. Annetut parametrit määrittävät potentiaalisen sijainnin. Potentiaalinen sijainti tarkastetaan BoxTraceByChannel-solmulla, joka tarkastaa tekoälyagentin nykyisen sijainnin ja potentiaalisen tulevan sijainnin välisen lentoradan esteettömyyden. Solmu tarkastaa lentoradan laatikon muodossa, jossa laatikon koko annetaan parametrinä solmulle. Tekoälyagentin koko saadaan parametriksi tähän käyttämällä GetActorBounds-solmua, joka palauttaa yhden actorin ulottuvuudet laatikon muodossa.

Käyttämällä kyseisiä solmuja saadaan lentoradan tarkastamisesta mahdollisimman tarkka. Mikäli tekoälyagentin lentoradalla on este, hylätään lentorata ja lasketaan uusi potentiaalinen lentokohde. Uudelleenlaskennalle rakennettiin logiikka, joka katkaisee uudelleenlaskemisen, mikäli se suoritetaan yli 20 kertaa siinä onnistumatta. Tällaisessa tapauksessa tehtävä palautetaan epäonnistu-

neena ja asetetaan uudeksi liikkumisen kohteeksi piste, josta agentti oli nykyiseen pisteeseen matkannut. Kaiken tämän tarkoituksena on välttää tekoälyagentin jumiin jääminen sijaintiin, josta sillä ei ole valideja lentoratoja kohti pelaajaa. Kuvasta 10 voidaan huomata, miltä lentoradan laskeminen näyttää konepellin alla.

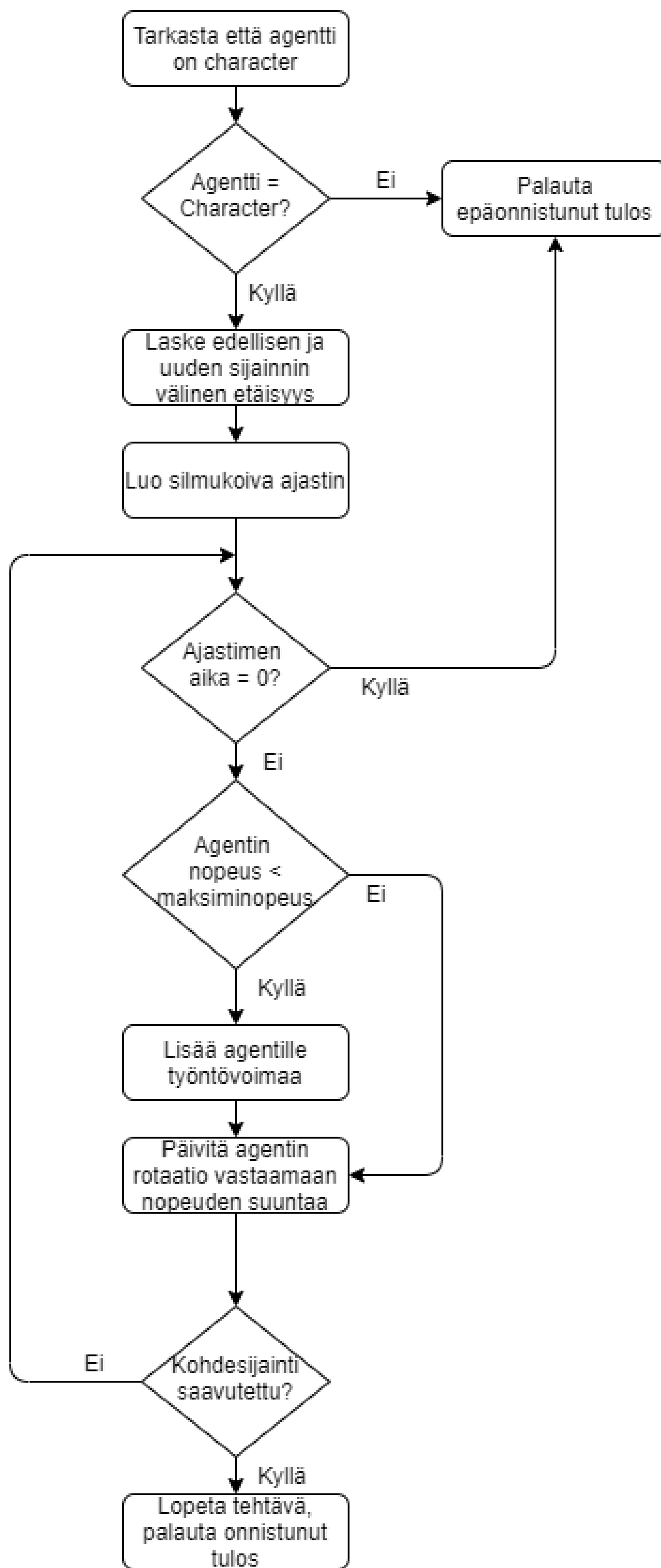


Kuva 10. BoxTraceByChannel-solmun Debug-optio mahdollistaa lentävän agentin liikkumisen kohteen laskemisen visualisoinnin.

Lentämisen toteuttamiselle tarvitaan erillinen tehtävä, johon aikaisemmin laskettu uusi sijainti annetaan parametrinä. Muita tarvittavia parametrejä ovat kaksi työntövoimaa rajoittavaa numeerista muuttujaa, joilla vaikutetaan maksiminopeuteen ja -kiihtyvyyteen. Myös liikkeen suoritusaika, sekä lopullisen sijainnin toleranssi tarvitaan muuttujiksi. Suoritusaika sekä lopullisen sijainnin toleranssi

toteutetaan, ettei tapahdu tilannetta, jossa tekoälyagentti jää jumiin, jos se ei syystä tai toisesta pääse kulkemaan sille annettuun sijaintiin. Tällaisessa tilanteessa tekoälyagentti odottaisi paikallaan, kunnes liikkeen suoritusaika loppuu. Vasta ajan loputtua laskisi agentti uuden sijainnin ja sitä kautta jatkaisi liikettä. Mikäli tekoälyagentti poikkeaa lentoradalta, jatkaa se silti matkaansa kohti sille annettua sijaintia, joten agentti voi jumiutua paikalleen vain tilanteissa, joissa jo validiksi tarkistettu sijainti muuttuu epävalidiksi liikkeen aikana.

Ensimmäisenä tehtävä tarkistaa, että sitä suorittava agentti on tyypiltään character. Käytännössä tämä tarkastus on helpoin tapa varmistaa, että tekoälyagentti sisältää komponentin, johon voidaan altistaa työntövoimaa. Characterin tapauksessa työntövoimaa voidaan altistaa joko kollisiokomponenttiin, tai mesh-komponenttiin. Tämän projektin hierarkiassa kollisiokomponentti on vihollisen juurikomponentti, joten voimaa kohdistetaan siihen. Seuraavana tehtävä mittaa lähtösijainnin ja päätesijainnin välisen etäisyyden, jonka se tallettaa myöhempää käyttöä varten. Tehtävässä luodaan seuraavaksi silmukoiva Event, jota suoritetaan parametrinä annetun suoritusajan verran. Event tarkistaa ensimmäiseksi agentin tämän hetken lentonopeuden ja applikoi työntövoimaa agenttiin, mikäli lentonopeus on alle maksiminopeuden. Tarkastus tehdään Get Physics Linear velocity -solmulla, jota verrataan maksiminopeuden vektoriin. Mikäli työntövoimaa applikoidaan, lisätään se Add Force -solmulla. Applikoitavan voiman määrä lasketaan alkuperäisen etäisyyden, nykyisen etäisyyden, nykyisen sijainnin, sekä maksimikiihtyvyyden avulla. Tehtävässä päivitetään seuraavana agentin rotaatio vastaamaan sen liikkeen suuntaa, jotta liike myös näyttäsi mahdollisimman realistiselta. Eventin ja tehtävän suoritus katkaistaan agentin saavuttaessa päämääränsä tai parametrinä annetun suoritusajan kuluessa loppuun (kuvio 4).



Kuvio 4. Työntövoimaa lisäävän tehtävän vuokaavio.

7.7 Tekoälyagenttien elinkaaren päättyminen

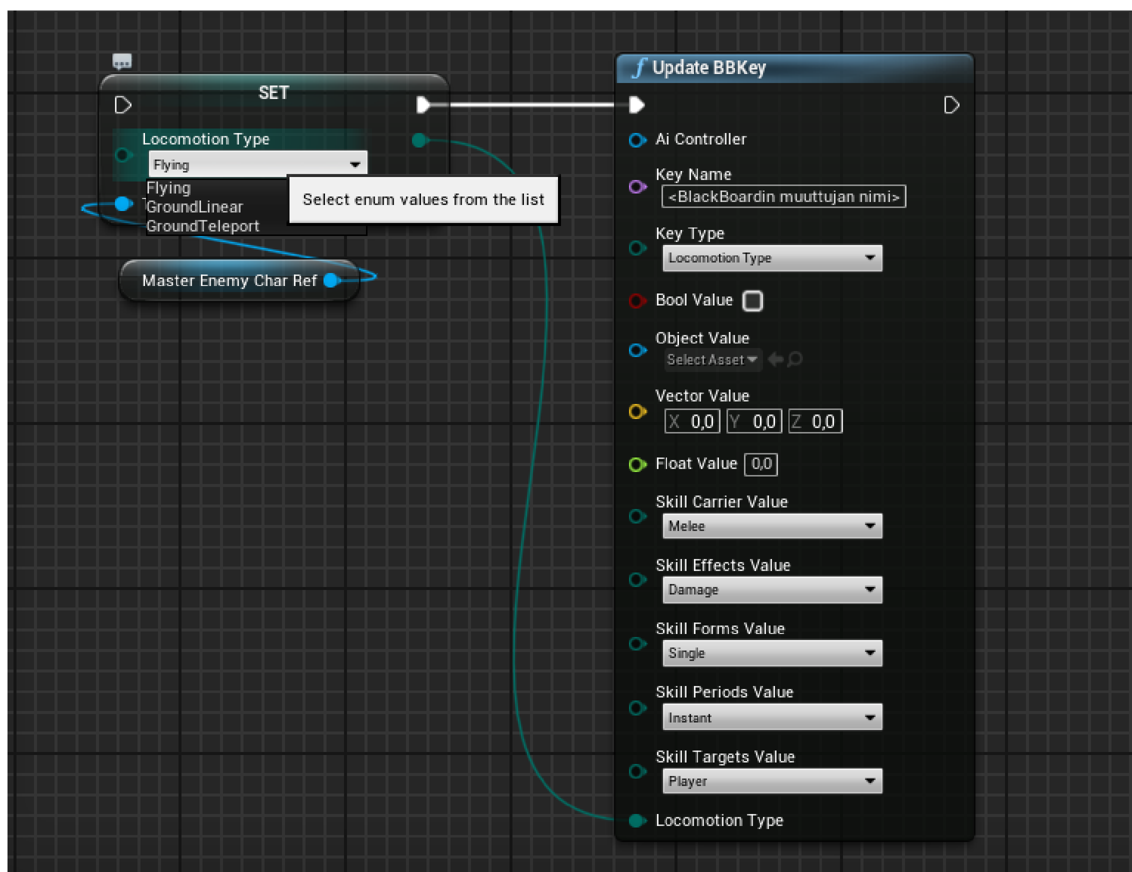
Tekoälyagenttien elinkaari päättyy, kun niiden HP arvo tippuu arvoon nolla. HP laskee joka kerta, kun pelaaja osuu tekoälyhahmoon ammuksella. Vähentämisen määrä riippuu pelaajan projektiilin sisältämästä damage-muuttujasta. Vahingon aiheuttamiseen hyödynnetään Unreal Enginen sisältämää funktiota Apply Damage, ja vahingon laskemiseen eventtiä AnyDamage. Tekoälyagentin elinkaaren päättyessä agentti tuhotaan solmulla Destroy Actor ja poistetaan sen referenssit aktiivisten vihollisten taulukosta.

7.8 Tekoälyagenttia tukevat muut toiminnot peliprojektissa

Opinnäytetyössä toteutettiin myös asioita, jotka edesauttavat agentin toimintaa tai parantavat pelaajan kokemusta, mutta jotka eivät suoraan liity tekoälyagentin elinkaareen. Tässä kappaleessa käymme lävitse kyseiset ohjelman osat.

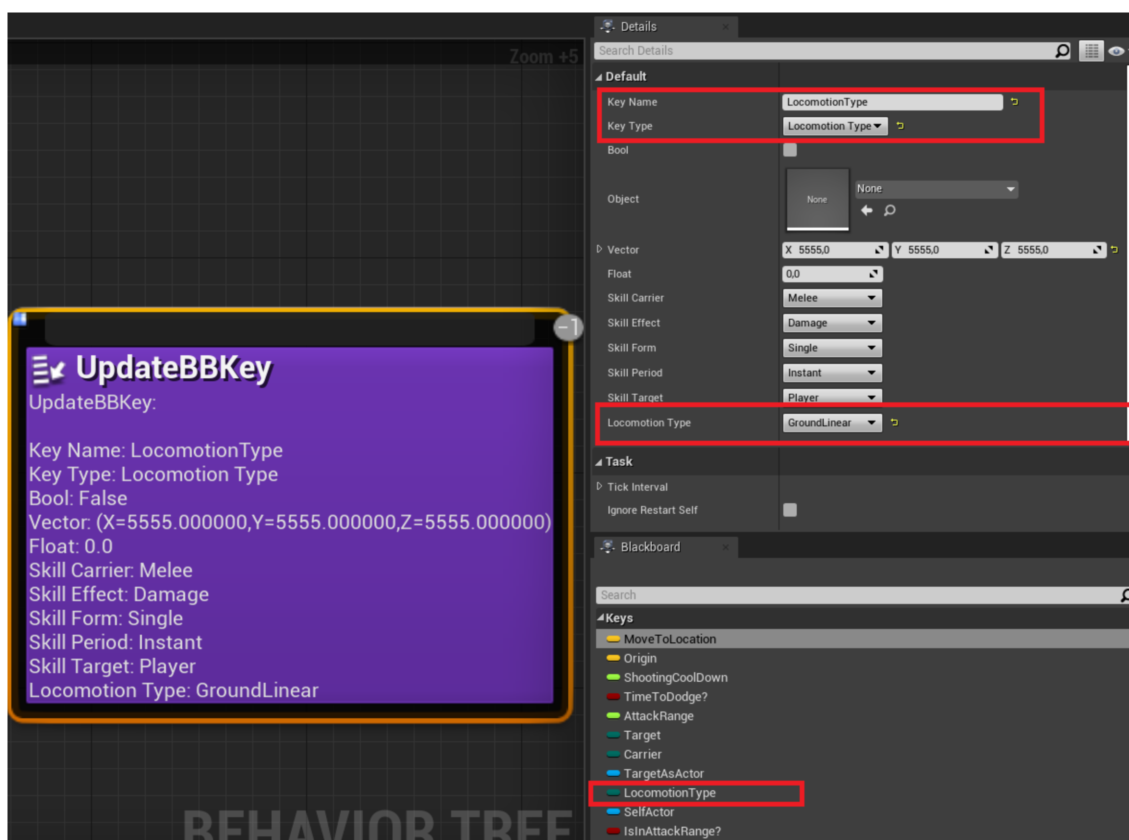
7.8.1 Tekoälyagentin muistin muuttujien päivittäminen

Normaali lähestymistapa blackboardin muuttujien arvojen päivittämiseen pelaamisen aikana on muuttaa niitä käytöspuulle tehdyissä tehtävissä. Tässä projektissa kuitenkin kaikilla vihollisilla käytetään samaa käytöspuuta, jonka seurauksena käytöspuu on hyvinkin suurikokoinen ja muuttujien asettamisen sijainnit helposti katoavat. Samoin joissakin tilanteissa halutaan päivittää blackboardin sisältämää muuttujaa muualtakin kuin käytöspuusta tai blueprintistä, jossa on jo suora referenssi käytöspuuhun. Kappaleessa 7.3.7 kerroin alustamiseen käytettävästä funktiokirjastosta. Funktiokirjaston funktiota voidaan käyttää myös blackboardin muuttujien päivittämiseen sovelluksen ollessa käynnissä. Esimerkiksi vihollishahmon liikkumistapa voidaan päivittää (kuva 11) minkä tahansa blueprintin sisältä, josta saadaan referenssi AIControlleriin.



Kuva 11. Esimerkki vihollishahmon liikkumismuodon päivittämisestä Blackboardille blueprinttien Event Graphissa käyttäen funktiokirjaston funktiota.

Funktio myös valjastettiin käytöspuun käyttöön tekemällä sille käytöspuun tehtävä (kuva 12). Tehtävälle annetaan muuttujiksi samat muuttujat mitkä funktiokin sisältää. Tehtävässä välitetään sen sisältämät muuttujat funktiolle, tarkoittaen sitä, että käytöspuusta voidaan suoraan kutsua funktiota ja sitä kautta muuttaa haluttua muuttujaa tehtävän details-paneelista. Hyötynä käytöspuulle valjastamisesta on se, että agentin käytöstä ohjaavan muuttujan muutoksista ollaan tietoisempia niiden ollessa näkyvissä suoraan käytöspuussa.



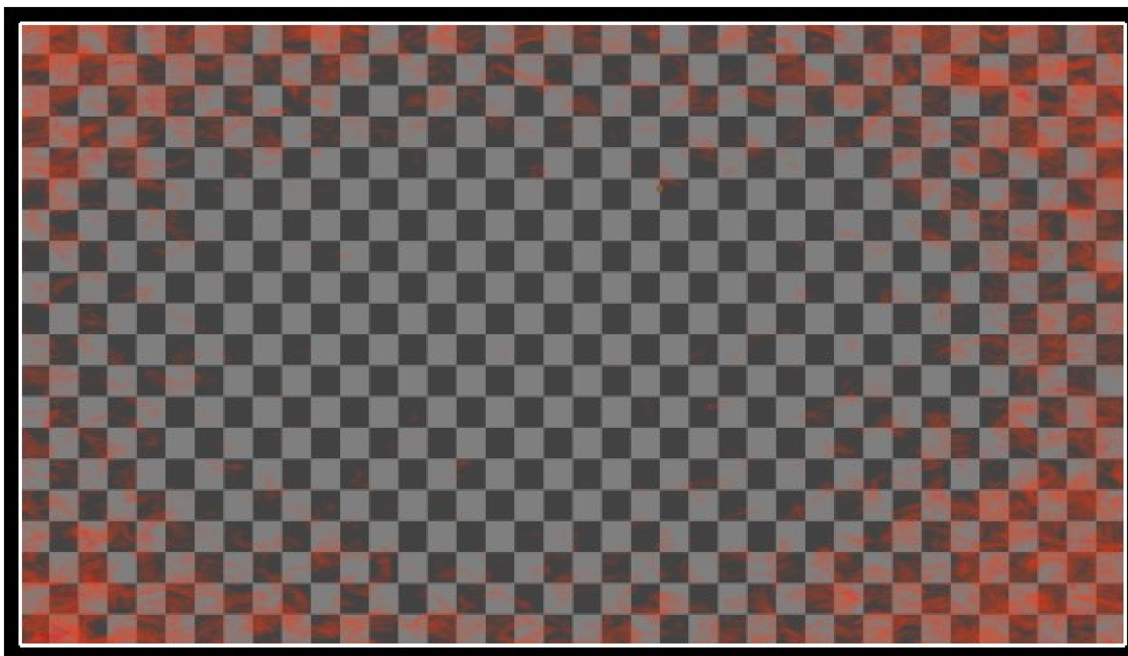
Kuva 12. Esimerkki vihollishahmon liikkumismuodon päivittämisestä Blackboardille käyttäen funktiokirjaston funktiota käytöspuun tehtävästä käsin.

7.8.2 Vahingon esittäminen pelaajalle

Pelaajalle halutaan indikoida vahingon toteutuminen silloin, kun tekoälyagentti vahingoittaa pelaajaa. Vahinko esitetään pelaajalle käyttämällä Widgettejä. Widgetit ovat Unreal Enginen käyttöliittymien luontiin tarkoitettu järjestelmä. Vahingon esittämiseen käytetään kahta widgettiä, joista toinen on jatkuvasti näkyvissä ja siinä esitetään pelaajalle tärkeää dataa, kuten elossa olevien vihollisten määrä, ammusten tilanne ja pelaajan jäljellä olevat HP:t. Tällaisesta käyttöliittymästä puhutaan yleensä HUDina tai Heads-up Displayna. HUDin tarkoitus on mahdollistaa datan näyttäminen pelaajalle virtuaalisessa muodossa siten, että se on mahdollisimman helposti saatavilla pelaajalle ilman ylimääräisiä interaktioita.

Toinen käyttöliittymä on yksikertaisesti kuva, joka lisätään pelaajan näkökenttään, kun pelaajaan aiheutetaan vahinkoa. Kuva häivytetään pois näkyvistä graduaalisesti muutaman sekunnin kuluessa sen lisäämisestä. Ammuntapeleissä

tämän kaltainen vahingon esittäminen pelaajalle on hyvin yleistä. Kuva on pääosin läpinäkyvä (kuva 13), jotta pelaajan ottaessa vahinkoa näkee hän kuitenkin vielä vihollisagentit ja voi jatkaa taistelua.



Kuva 13. Pelaajan näyttöikkunan päälle lisättävä käyttöliittymäkuva, joka indikoi pelaajan vastaanottaneen vahinkoa.

Kuvan häivyttämiseen ei ole suoraa toiminnallisuutta Unreal Enginessä, vaan se täytyy rakentaa itse käyttäen sisäkkäisiä ajastimia. Sisäkkäisistä ajastimista ensimmäinen asetetaan silmukoimaan itseään parametrin mukaisen ajan välein. Ulkoisemman ajastimen parametriksi annetaan kuvan näkyvyyden kokonaisaika, eikä sitä aseteta silmukoimaan. Ulkoisemman ajastimen tehtävänä on katkaista sisemmän silmukoivan ajastimen pyörintä. Sisemmän ajastimen silmukassa kuvan läpinäkyvyttä korotetaan jokaisessa iteraatiossa. Kun iteraatiot on suoritettu, poistetaan widget, sekä ajastimet. Widgetit ja ajastimet ovat entititeettejä Unreal Enginessä, jotka kannattaa poistaa kokonaan ja luoda tarvittaessa aina uudelleen, koska niissä ei ole tärkeää saada yksittäistä instanssia talteen. Poistamalla inaktiivisia asioita optimoidaan ohjelman muistin kulutusta pienemmäksi.

8 Integrointi

Integrointi suoritettiin henkilön kanssa, joka oli toteuttanut suuren osan muusta projektista. Täten pyrittiin välttämään pahimmat konfliktit projektissa. Integroinnissa hyödynnettiin Unreal Editorista löytyvää migrate-työkalua, jolla voidaan siirtää yhden projektin sisältämiä tiedostoja toiseen projektiin. Integrointi sujui pääosin ongelmitta, mutta migraten tuodessa projektin kaikki tiedostot toiseen projektiin, tuli joitakin päällekkäisiä toiminnallisuuksia. Päällekkäiset toiminnallisuudet tuli korjata poistamalla toinen, tai rakentamalla koodiin molemmat toiminnallisuudet huomioonottava ratkaisu. Is Head Mounted Display Connected ja Is Head Mounted Display Enabled -solmuilla voidaan tarkastaa, että onko VR-laitteet kiinni ja valmiina käyttöön, jonka pohjalta voidaan viedä koodia eri toteutushaaroihin.

9 Jatkokehitysideat

Projektia tullaan jatkokehittämään toimeksiantajan toimesta. Henkilökohtaisesti en projektin jatkokehitykseen osallistu, joten tämän kappaleen jatkokehitysideat ovat teoreettisia. Jatkokehitysideat ovat myös henkilökohtaisella tasolla pähkäiltyjä ja vastaavat kysymykseen ”Mitä haluaisin nähdä valmiissa peliprojektissa tekoälyn kannalta katsottuna?”.

9.1 Taidot

Tekoälyagentin taito on ensimmäinen asia, jonka kehittämistä jatkaisin. Taidon kohteen arvontaan toisin lisää logiikkaa. Lisäisin tekoälyagenteille mahdollisuuden vaihtaa taidon kohdetta, mikäli sillä olisi parempi kohde. Esimerkiksi agentti voisi priorisoida agentin HP:n palauttamista, jolla ne ovat vähissä. Käytännössä se onnistuisi lisäämällä logiikkaa osumatarkastaluun. Osumatarkastelussa tarkastetaan vihollisagentin HP:n arvo ja lisätään sille tagi, tai tehdään erillinen taulukko suuren prioriteetin vihollisille. Tekoälyagentin taidon kohdetta täytyy tällaisessa tilanteessa päivittää useasti.

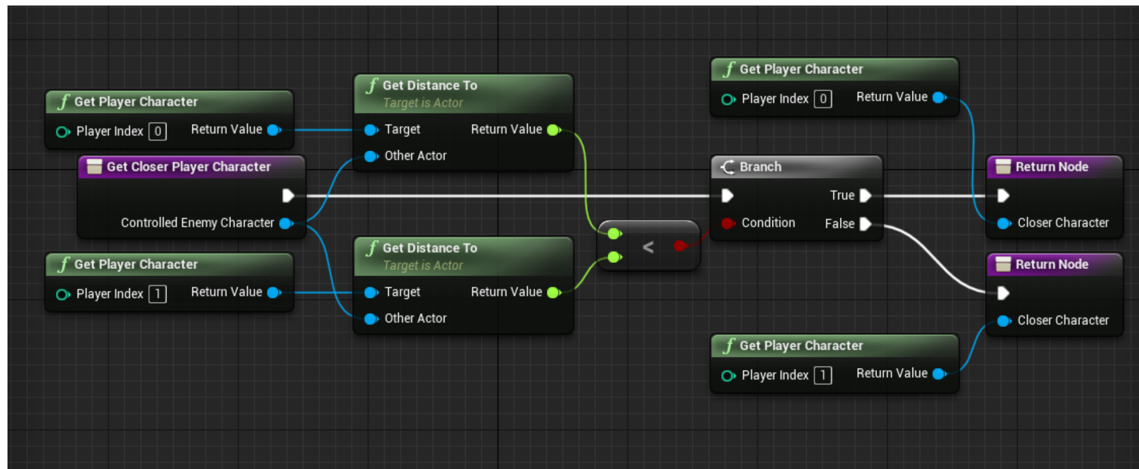
Hyödyntämällä Event Dispatchereita taas voitaisiin toteuttaa efekti, jossa heikko tekoälyagentti ikään kuin kutsuu apua, ja muut tekoälyagentit vastaavat siihen. Mielestäni Event Dispatchereilla toteuttamalla saadaan tekoälyagenteista realistisemman oloiset, jossa muutokset voivat tapahtua sen perusteella mitä toiselle tekoälyagentille kävi tai käy.

Taitoja alkaisin kehittää myös käytännössä. Peliin lisää ilmettä toisi toisiaan parantavat tekoälyagentit ja tekoälyagenttien erilaiset hyökkäykset pelaajaa kohtaan.

9.2 Pelaajan puolella taistelevat hahmot

Pelaajaan puolella taistelevia agentteja haluttiin alun perin toteuttaa jo opinnäytetyön puitteissa. Skaalautuvasti toteutettuna pelaajan puolella taistelevat hahmot kuitenkin ovat niin suuri paketti, että se jätettiin jo suunnitteluvaiheessa pois. Yksinkertaisesti tekoälyagentteja voitaisiin käännyttää taitonsa puolesta käyttämään niitä vihollisagentteihin ja sitä kautta saada efektiä pelaajan puolella taistelevista agenteista. Kuitenkin jatkokehityksessä halutaan pitää auki moninpelin mahdollisuus, jolloin tekoälyagenteille halutaan parempi järjestelmä niiden liittolaisten määrittämiseen ja ohjaamiseen.

Moninpelin toteuttaminen ei tekoälyn kannalta teoriassa muuta paljoa, mutta käytännössä jokaiseen Get Player Pawn, tai Get Player Character -solmuun täytyy rakentaa logiikkaa pelaajahahmon valintaan. Molemmat mainituista solmuista saavat parametrinä Player Index -nimisen numeerisen arvon. Sovelluksen sisältäessä vain yhden pelaajahahmon saadaan oikea referenssi pelaajaan aina indeksillä 0. Pelihahmoja ollessa useampi tarvitaan logiikka oikeassa indeksissä sijaitsevat pelaajahahmon valintaan (kuva 15). Tekoälyn toteutuksessa käytetään Get Player Character -solmua tekoälyagentin kohteen konvertoinnin yhteydessä, sekä ympäristötutkinnan testeissä, joissa kontekstina on pelaajahahmo. Projektiin sopivasti voidaan pelaajahahmo valita sattumanvaraisesti, mutta mielestäni jonkinlaisen järkevän logiikan rakentaminen siihen, että pelaajat kokevat jotakuinkin samanlaista haastetta olisi järkevä rakentaa.



Kuva 15. Esimerkki kahden pelaajahahmon etäisyyden tutkinnasta tekoälyagentista katsottuna.

9.3 Aallot

Projektin lopullisessa versiossa ei ole tarkoitus olla demoversion kaltaista selkeää loppua, vaan pelin on tarkoitus loppua vasta kun pelaaja kuolee. Aaltojen loppumattomaksi saattamiseen tarvitsee vain rakentaa pomoaaltojen logiikka, ja poistaa käytöstä loppumattoman aaltojen kasvun rajoittavan loppumisen logiikan.

9.4 Vahingon aiheuttaminen

Peliin lisää taitopohjaisuutta voitaisiin tuoda asettamalla tekoälyagenttien kehoon kohtia, joihin osuminen tuottaa enemmän vahinkoa. Käytännössä se tapahtuisi yksinkertaisesti toteuttamalla tekoälyagentille useita kollisiokomponentteja ja toteuttamalla eventit jokaiseen kollisiokomponenttiin osumiseen kertomalla aiheutettavaa vahinkoa jollain kertoimella.

9.5 Pisteytys ja pelaajalle osoitettava Data

Pelaajan pelaamisen onnistumista voitaisiin pisteyttää jatkossa paremmin. Tällä hetkellä pelaajan onnistumista ei mitata kuin aallon numerossa, johon asti pelaaja selvisi hengissä. Demoversion selkeä loppu myös rajoittaa pelaajien pa-

remmuuden mittausta hankalammaksi, sillä jos kaikki pelaajat selviävät hengissä viimeisen aallon niin parhaasta suorituksesta ei ole tietoa. Jatkokehitys taklaa osittain ongelmaa, kun pomoaallot toteutetaan ja demoversion selkeä loppu otetaan pois käytöstä. Silloin onnistumista määrittää aallon numero, johon asti pelaaja pääsee.

Pelin aikana vihollishahmoihin osuessa jonkunnäköinen indikaattori niihin osumisesta, kuten vaikkapa aiheutetun vahingon määrä pompahtaa esiin viholliseen osuessa. Pelin loppuessa pelaajalle voitaisiin osoittaa dataa esimerkiksi omasta osumatarkkuudesta, pääosumien määrästä ja tarkkuudesta.

9.6 Tekoälyagentin aistit

Pelin tyyppi aiheutti sen, ettei tekoälyagentille koettu tärkeäksi toteuttaa aistien perusteella toimimista. Jatkokehityksessä kuitenkin tekoälyagentin aistien simuloinnilla voitaisiin tuoda lisää syvyyttä agenttien toimintaan. Samu Peltola (2020) toteuttaa opinnäytetyössään tekoälyagentin aistien simuloimista käyttäen AIPerception-komponenttia. Peltola simuloi opinnäytetyössään agentin näköaistia ja hyödyntää sitä vartioinnin kaltaiseen tehtävään agentilla. Esimerkiksi näkö- tai kuuloaistia simuloimalla saataisiin tekoälyagentin toiminnoista ihmismäisempiä, jossa tekoälyagentti vastaa ärsykkeeseen. (Peltola 2020.) Kyseisen kaltaista aistien simulointia voitaisiin myös tämän opinnäytetyön jatkokehityksessä toteuttaa.

10 Pohdinta

Tässä luvussa reflektoidaan toteutettua työtä, pohditaan onnistumisia sekä haasteita. Luvussa myös mietitään vaihtoehtoisia toteuttamistapoja, jotka olisi mahdollisesti ollut parempia valintoja kuin käytetyt tavat.

10.1 Onnistumiset ja haasteet

Opinnäytetyötä voidaan pitää kokonaisuudessaan onnistuneena. Toimeksiantajan asettavat vaatimukset saavutettiin. Järjestelmän kannalta mietittynä eritoten lentämisen tehtävät onnistuivat erityisen hyvin. Sen toteuttaminen oli projektin ehdottomasti hankalin toteutettava asia, mutta lopputuloksesta sitä ei välttämättä huomaisi. Samoin enumeraattoreiden käyttö alustamisen yhteydessä osoittautui viisaaksi valinnaksi sen helpottaessa sattumanvaraisuuden luonnissa. Tekoälyagenteista saatiin persoonallisia hahmoja, jotka eroavat toisistaan selkeästi ilmankin visuaalisuutta.

Opinnäytetyön suurimmat haasteet liittyivät arkkitehtuuriin. Tekoälyagentit halutaan lähtökohtaisesti alustaa ennen niiden syntymistä maailmaan, eli tekoälyagentin vaikeuspisteet täytyy asettaa ennen kuin agentti synnytetään. Tämä aiheuttaa sen, että synnyttävästä entiteetistä ei voida kontrolloida syntyvää agenttia millään lailla. Tämä aiheuttaa efektin, jossa tekoälyagentteja saatetaan synnyttää yksi ylimääräinen jokaiseen aaltoon. Kyseinen haaste jätettiin huomioimatta projektin luonteen takia, jossa aallot ovat jo valmiiksi sattumanvaraisuuden alaisia.

Haastetta tuotti myös taitojen arvonnan arkkitehtuurin luominen opinnäytetyön osana. Itse taitojen arvonnan toteuttaminen oli onnistunut, mutta koska tekoälyagentin liikkeen kohteena on taidon kohde, johti se tilanteeseen, jossa osa tekoälyagenteista ei liiku mihinkään synnyttyään pelimaailmaan. Tämä johtuu siitä, että niiden taidon kohde olivat ne itse. Toiminnallisuuksien testaamista varten tekoälyagenteilta otettiin pois mahdollisuus saada kohteeksi oma itsensä.

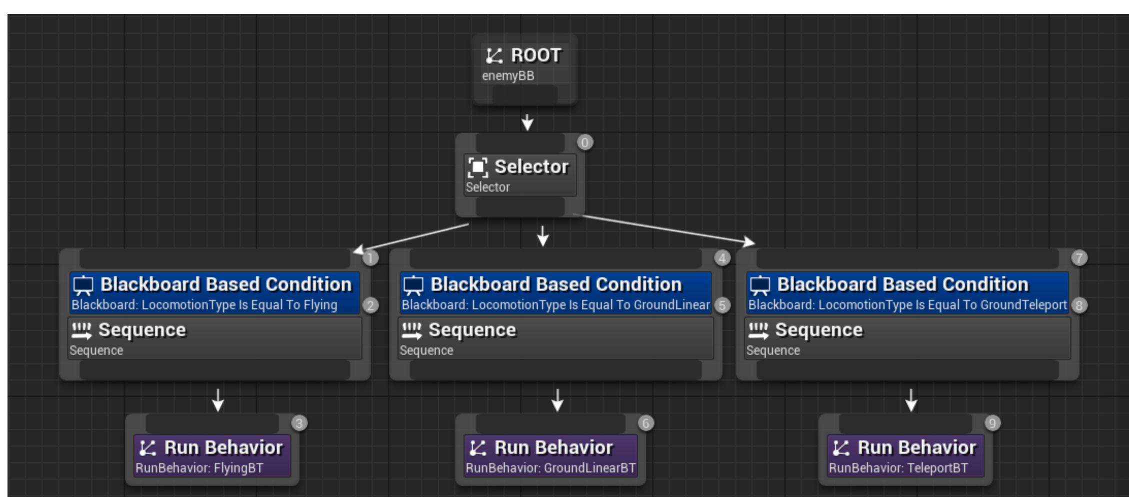
10.2 Vaihtoehtoiset toteuttamistavat

10.2.1 Käytöspuun optimointi

Käytöspuun toteutuksessa käytettiin paljon enumeraattoreita, joilla ohjataan käytöspuun haarojen läpikäyntiä. Enumeraattoreiden sisältäessä monta vaihtoehtoa, kasvaa käytöspuun koko myös suuremmaksi ja suuremmaksi. Yhden

käytöspuun toteutus on jatkokehityksen kannalta huono koodin seurattavuuden kannalta. Täten jälkikäteen korjaisin toteutuksen sisältämään useita eri käytöspuita. Kaikki erilliset käytöspuut voivat käyttää samaa blackboardia, sekä hyödyntää samoja itse toteutettuja tehtäviä.

Käytöspuusta voitaisiin tehdä helpommin seurattava käyttämällä dynaamisia sisäkkäisiä käytöspuita Run Dynamic Behaviour -tehtävällä. Tällöin pääkäytöspuun tehtävänä on vain asettaa tekoälyagentille suoritettava sisäkkäinen käytöspuu (kuva 14). Vastaavanlaisen toteutuksen voisi myös toteuttaa AIControllerin sisälle, jonne rakennettaisiin logiikka käytöspuun valinnalle. AIControlleriin toteuttaessa olisivat kaikki käytöspuut silloin rinnakkaisia toteutuksia, joista valitaan joidenkin parametrien perusteella suoritettava käytöspuu. AIControllerista viedään nykyisessä toteutuksessa muuttujat käytöspuulle, joten se olisi looginen paikka myös suorittaa käytöspuun valintaa, koska se sisältää kaikki nykyiset käytöspuun ohjaukseen tarvittavat muuttujat jo valmiiksi.



Kuva 14. Pilkkomalla käytöspuu useaan eri käytöspuuhun käyttämällä run behaviour, tai run behaviour dynamic -tehtäviä saadaan pääkäytöspuusta hyvin helposti seurattava.

10.2.2 Taidon Arkkitehtuuri

Taidon arkkitehtuurin korjaisin mahdollisuuksien mukaisesti sellaiseksi, että taidon kohde ei olisi enumeraattori. Nykyisen enumeraattorin sijaan muuttaisin taidon kohteen actor-tyyppiseksi taulukoksi. Actor-taulukosta arpominen olisi hieman hankalampaa, koska sitä ei voida täyttää etukäteen, mutta tekemällä

kaikkien mahdollisten kohteiden taulukko, sekä pitämällä sitä ajan tasalla hahmojen kuollessa ja syntyessä saataisiin se todennäköisesti toteutettua. Korjatulla arkkitehtuurilla olisi se etu, ettei taidon kohdetta tarvitsisi konvertoida enumeraattorista actoriksi. Mikäli jokin muukin käytöspuulle vietävä enumeraattori tarvitsisi konvertoida toiseksi muuttujatyypiksi, tulisi koko konvertointi toteuttaa täysin uudelleen, koska se on mahdotonta toteuttaa modulaarisesti.

10.2.3 Widgetit VR-maailmassa

Widgetit ovat opinnäytetyön ainoa asia, joka vaatii korjauksia VR-maailmaan siirryttäessä. Widgettien interaktio ei toimi oikein VR-maailmassa, kun käytetään perinteisiä näyttöikkunan päälle lisättäviä widgettejä. Widgeteistä täytyy tehdä 3D widgettejä ja ne tulee toteuttaa pelimaailman sisälle näyttöikkunan päälle lisäämisen sijasta. Pelimaailmassakin ne voidaan lisätä esimerkiksi pelaajan kameraobjektiin kiinni siten, että widget kulkee pelaajalle kuvaa tuottavan kameran kanssa kiinteässä pisteessä. Toinen vaihtoehto on lisätä widget esimerkiksi pelaajan käyttämiin VR-kontrollereihin siten tuoden efektin, jossa kontrollereita katsomalla saadaan näkyviin tarvittava data. Widget voidaan myös lisätä pelimaailmassa sijaitsevaan staattiseen objektiin, jolloin data on näkyvissä aina samassa paikassa pelimaailman sisällä.

Lähteet

Cordone, R. 2019. Unreal Engine 4 Game Development Quick Start Guide. Packt Publishing. https://www.packtpub.com/product/unreal-engine-4-game-development-quick-start-guide/9781789950687 15.3.2021.

GamesFromScratch. 2016. Alternative Programming Languages with Unreal Engine 4. https://gamefromscratch.com/alternative-programming-languages-with-unreal-engine-4/ 17.3.2021.

Lahtinen, S. 2020. Hyötypohjaisen tekoälyn toteuttaminen Unity-pelimoottorissa. Kajaanin ammattikorkeakoulu. Tietojenkäsittely. Opinnäytetyö. https://www.theseus.fi/bitstream/handle/10024/346326/Lahtinen_Sami.pdf 7.5.2021.

Lappalainen, P. 2020. Itsenäiset älykkäät agentit. Jyväskylän yliopisto. Informaatioteknologian tiedekunta. Kandidaatintutkielma. https://jyx.jyu.fi/bitstream/handle/123456789/68899/1/URN%3ANBN%3Afi%3Ajyu-202005083109.pdf 12.3.2021.

Lee, J. 2015. An Overview of Unreal Engine. Packt Publishing. https://hub.packtpub.com/overview-unreal-engine/ 12.3.2021.

Lowing, B. 2017. Writing an EQS test. Mercuna Developments. https://mercuna.com/writing-eqs-test/ 26.3.2021.

Romero, M & Sewell, B 2019. Blueprints Visual Scripting for Unreal Engine - Second Edition. Packt Publishing. https://subscription.packtpub.com/book/game_development/9781789347067/1/ch01lvl1sec04/the-blueprint-editor-interface 24.3.2021.

Souza, P. 2020. Unreal Engine AI with Behaviour Trees. Epic Games. https://www.youtube.com/watch?v=iY1jnFvHgbE 17.3.2021.

Peltola, S. 2020. Tekoäly Unreal Engine -ympäristössä. Tampereen ammattikorkeakoulu. Tieto- ja viestintätekniikka. Opinnäytetyö https://www.theseus.fi/bitstream/handle/10024/338527/Peltola_Samu.pdf 6.5.2021.

Unreal Engine. 2021a. Blueprint Overview. Epic Games. https://docs.unrealengine.com/en-US/ProgrammingAndScripting/Blueprints/Overview/index.html 8.2.2021.

Unreal Engine. 2021b. AAIController. Epic Games. https://docs.unrealengine.com/en-US/API/Runtime/AIModule/AAIController/index.html 8.2.2021.

Unreal Engine. 2021c. Character. Epic Games. https://docs.unrealengine.com/en-US/InteractiveExperiences/Framework/Pawn/Character/index.html 9.2.2021.

Unreal Engine. 2021d. AI Perception. Epic Games. https://docs.unrealengine.com/en-US/InteractiveExperiences/ArtificialIntelligence/AIPerception/index.html 9.2.2021.

Unreal Engine. 2021e. Arteficial Intelligence. Epic Games. https://docs.unrealengine.com/en-US/InteractiveExperiences/ArtificialIntelligence/index.html 15.3.2021.

Unreal Engine 2021f. Behaviour Tree Node Reference: Tasks. Epic Games. https://docs.unrealengine.com/en-US/InteractiveExperiences/ArtificialIntelligence/BehaviorTrees/Behav

iorTreeNodeReference/BehaviorTreeNodeReferenceTasks/in-
dex.html 15.3.2021.
Unreal Engine 2021g. In-Editor Testing (Play & Simulate). Epic Games.
https://docs.unrealengine.com/en-
US/BuildingWorlds/LevelEditor/InEditorTesting/index.html 26.3.2021.
Unreal Engine 2021h. Behavior Tree Overview. Epic Games. https://docs.unre-
alengine.com/en-
US/InteractiveExperiences/ArtificialIntelligence/BehaviorTrees/Behav
iorTreesOverview/index.html 22.3.2021.