Unityllä tehty tekoäly-oppimateriaali

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Ammattikorkeakoulututkinnon opinnäytetyö

Hämeenlinnan korkeakoulukeskus, Tietojenkäsittelyn koulutusohjelma

Kevät, 2021

Lauri Koho

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Tietojenkäsittelyn koulutusohjelma
Hämeenlinnan korkeakoulukeskus

Tiivistelmä

Tekijä Lauri Koho

Vuosi 2021

Työn nimi Unityllä tehty tekoäly-oppimateriaali

Ohjaajat Tommi Lahti

TIIVISTELMÄ

Opinnäytetyön tavoitteena oli etsiä pelinkehittämisestä kiinnostuneille henkilöille helposti lähestyttävä ohjelmointityökalu tekoälyn tekemiseen Unity-pelimoottorille. Työssä tutustutaan Bolt Visual Scripting -ohjelmointityökaluun, jota käyttäen luodaan helppo ja opettavainen oppimateriaali tekoälyn käyttöönotosta visuaalisella ohjelmoinnilla.

Opinnäytetyön teoriaosuudessa verrataan visuaalista ohjelmointia kirjoitettuun ohjelmointiin. Teoriaosuudessa käydään myös läpi yleisiä tekoälytyyppejä ja näitä hyödyntäviä Unity-työkaluja. Käytännön osuudessa luotiin esimerkkiprojekti, jossa visuaalista ohjelmointia hyödyntämällä pystytään ottaa käyttöön tekoälylogiikka nopeasti ja helposti seuraamalla oppimateriaalia.

Visuaalisella ohjelmoinnilla muidenkin kuin ohjelmoinnin osaajien on mahdollista osallistua monipuolisesti pelimaailman ja ohjelmointilogiikan luomiseen. Johtopäätöksenä voidaan todeta, että Bolt Visual Scripting on helposti lähestyttävä ohjelmointityökalu visuaaliseen ohjelmointiin, joka luo hyvät edellytykset tekoälyn oppimiselle ja oman pelin luomiselle.

Avainsanat Visuaalinen ohjelmointi, tekoäly, Unity

Sivut 32 sivua ja liitteitä 1 sivu

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Degree programme in Business Information Technology
Hämeenlinna University Center

Abstract

Author Lauri Koho
Year 2021
Subject Artificial intelligence learning material made with Unity
Supervisors Tommi Lahti

ABSTRACT

The aim of the thesis was to find an easily accessible programming tool for people interested in game development to create artificial intelligence for the Unity game engine. The thesis introduces the Bolt Visual Scripting tool, which is used to create comprehensive and instructive learning material of artificial intelligence through visual programming.

The theoretical part of the thesis compares visual programming with written programming. The theoretical part also reviews common types of artificial intelligence and Unity tools that utilize these. In the practical part, an example project was created by using visual programming. In this way, artificial intelligence could be implemented quickly and easily by following the learning material.

With visual programming, it is possible for non-programming professionals to participate in the creation of the game world and programming logic. In conclusion, Bolt Visual Scripting is an approachable programming tool that creates good opportunities for learning artificial intelligence and creating your own game.

Keywords Visual programming, Artificial intelligence, Unity

Pages 32 pages and appendices 1 page

Sisällys

Kuvat

Liitteet

1 JOHDANTO

Jos olet joskus pelannut videopelejä, olet ollut tekemisissä tekoälyn kanssa. Olipa se mikä tahansa videopeligenre, löydät niistä jonkinlaisia tekoälyä hyödyntäviä elementtejä. Videopelien tekoälyn tarkoituksena on luoda illuusio elävästä pelimaailmasta, jolloin pelaajat saavat paremman pelikokemuksen. Yleisin rooli tekoälyllä pelimaailmassa on ei-pelattavan hahmon käyttäytymislogiikan hallinta, jonka tekemisessä hyödynnetään erilaisia ohjelmointikieliä ja tekoälyhaaroja. (Lou, 2017)

Aloitteleva pelinkehittäjä, jolla ei ole aikaisempaa ohjelmointikokemusta, voi kohdata turhautumisen tunteita etsiessään ensimmäiselle peliprojektilleen ohjelmointityökalua Unity-pelimoottorista. Tämän opinnäytetyön ensisijainen tehtävä on auttaa etsimään peliohjelmoinnista kiinnostuneelle helposti lähestyttävä ohjelmointityökalu, joka innostaa ohjelmoinnin oppimista ja luo pelimaailmaa syventävän tekoälyn tekemiselle parhaan mahdollisen alustan.

Opinnäytetyössäni tutkin visuaalisen ohjelmoinnin hyötyjä kirjoitettuun ohjelmointiin verrattuna ja tutustun Unity-pelimoottorille hankittuun Bolt Visual Script -ohjelmointieditoriin. Teen luokittelun Unity-pelimoottorilla hyödynnettävistä tekoälyistä ja niiden luontityökaluista. Tutkimukseni lopuksi hyödynnän Bolt Visual Script -työkalua tehdessäni Unity-pelimoottorille pelikentän, jossa ohjelmoin visuaalista ohjelmointieditoria käyttäen kahdelle peliobjektille tekoälyä käyttävät elementit.

Tutkimuskysymykseni olivat:
1. Mitä Unity tuo Visual Scriptillä pelin kehittämiseen?
2. Kuinka Bolt auttaa aloittelevaa pelinkehittäjää tekoälyn luonnissa?
3. Miten otetaan käyttöön tehokkaasti tekoäly kulkukaaviolla ja tilakaaviolla?

2 Visuaalinen ohjelmointi

Tässä luvussa tutkitaan visuaalista ohjelmointia ja sen hyötyjä kirjoitettuun ohjelmointiin verrattuna. Lopuksi tutustutaan Unityn omaan visuaaliseen ohjelmointijärjestelmään Boltiin.

2.1 Visuaalisen ohjelmoinnin vertailu kirjoitettuun

Visuaalisella ohjelmoinnilla tarkoitetaan ohjelmointijärjestelmää, jossa ohjelmointikielen elementit löytyvät käyttäjälle graafisina rakennuspaloina. Rakennuspalojen informaatio ja ulkonäkö kertovat käyttäjälle rakennuspalan funktion, jolloin käyttäjän on helppo ymmärtää ohjelman tietovirran liikkumista erilaisten palojen välillä. Rakennuspalat sisältävät monenlaisia toimintoja, kuten muuttujan tietoja, matemaattisia laskelmia, objekteja tai koordinaatteja. (Ionos, 2020)

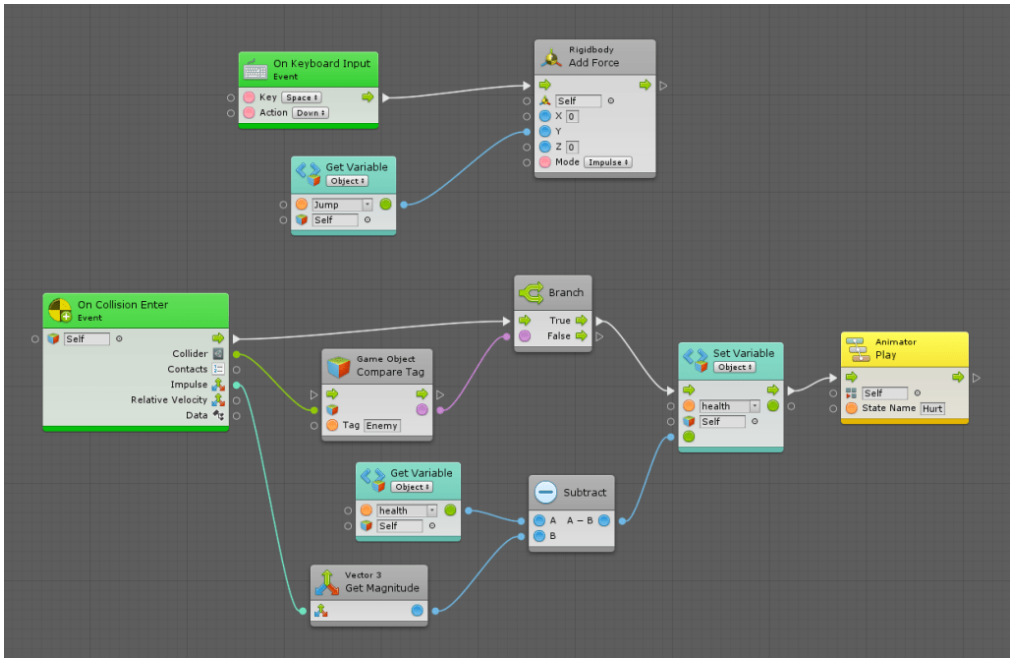
Visuaalisen ohjelmoinnin suurin hyöty kirjoitettuun ohjelmointiin verrattuna on se, ettei käyttäjän tarvitse heti osata valtavaa määrää komentosyntakseja ja niiden työnkulun logiikkaa tai ohjelmistokielen sanastoja. Kirjoitusvirheet ovat myös minimoitu visuaalisessa ohjelmoinnissa, koska funktiot löytyvät valmiiksi palikoista, jotka yhdistetään loogisella ajattelulla toimivaksi kokonaisuudeksi. Kirjoitetussa ohjelmoinnissa kirjoitusvirheet ovat arkipäivää, koska käyttäjä keskittyy suureen määrään informaatiota pitkäksi aikaa, mikä väsyttää nopeasti keskittymiskykyä. Vaikka ohjelmointieditori tarjoaisikin automaattisesti täydennystä väärin kirjoitetulle syntaksille, ei ohjelma välttämättä toimi mahdollisen funktioristiriidan takia. Tämä tarkoittaa, että käyttäjä joutuu tekemään ylimääräistä työtä ja käyttämään aikaa tarkistaessaan mitä virheitä koodista löytyy. (Ionos, 2020)

2.2 Bolt Visual Scripting

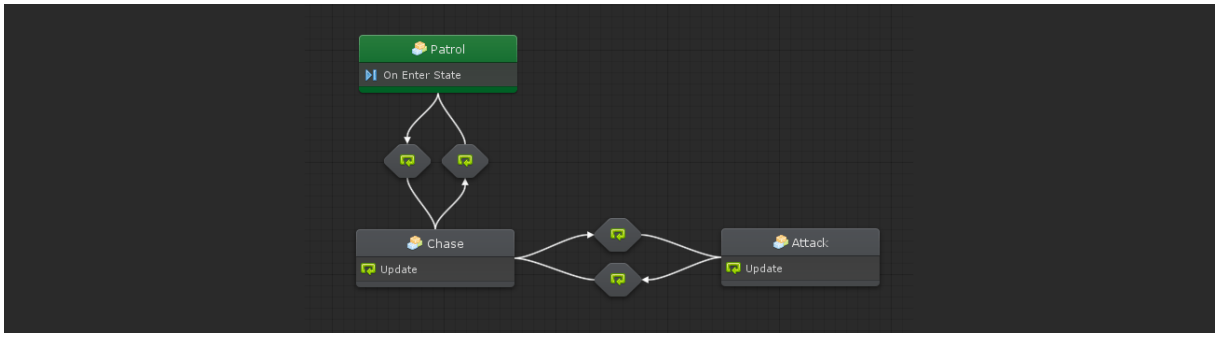
Bolt on visuaalinen C#-kieltä käyttävä ohjelmointieditori, jonka on kehittänyt yritys Ludiq. Ohjelmointieditorin laajennuksen tavoitteena on, että muut kuin ohjelmoijat saavat mahdollisuuden kirjoittaa ja ymmärtää pelilogiikkaa. Tämä mahdollistaa esimerkiksi taiteilijoiden ja suunnittelijoiden monipuolisemman tekoprosessiin osallistumisen. (Low-scope, 2019; ks. myös GameDaily.biz., 2020; VionixStudio, 2020)

Boltissa on kaksi pääominaisuutta: kulkukaavio (Kuva 1) ja tilakaavio (Kuva 2). Kulkukaaviossa yhdistetään toiminnot ja arvot loogisessa järjestyksessä. Tilakaaviossa taas luodaan erilaisia tiloja ja niiden välisiä siirtymiä. (Unity, 2020) Unity Technologies osti Boltin toukokuussa 2020, minkä jälkeen Unity ilmoitti Bolt Visual Scriptin ilmestyvän heinäkuun lopussa ilmaiseksi Unity Asset Storessa (GameDaily.biz., 2020; ks. myös VionixStudio, 2020). Vuonna 2021 Unity Technologies aikoo tarjota Boltin pelimoottorin sisäänrakennettuna ydinominaisuutena, jolloin luodaan samalla johdonmukaisuus ja integrointi muihinkin pelimoottorin visuaalisiin kehitystyökaluihin, mikä parantaa käyttäjäkokemusta (Unity, 2020).

Kuva 1 Esimierkkikuva Kulkukaaviosta ”Flow Graphs”. (VionixStudio, 2020)



Kuva 2 Esimerkkikuva tilakaaviosta "State Graphs". (Low-scope, 2019)



2.2.1 Kulkukaavio (Flow Graph)

Kulkukaavio on kuin normaali C#-tiedosto, joka sisältää tarvittavat muuttujat ja funktiot yksinkertaisen pelilogiikan tekemiseen. Boltissa C#-tiedoston sijaan luodaan ominaisuustiedosto, jota kutsutaan kulkumakroksi (Flow macro). Makro on uudelleenkäytettävä kaavio, jolla voidaan viitata useaan peliobjektiin. Kulkumakro tarvitsee toimiakseen kulkukoneen, joka on komponentti, joka asetetaan peliobjektiin suorittamaan kulkukaavion logiikkaa pelitilassa. (Low-scope, 2019; Unity, 2020; ks. myös VionixStudio, 2020)

2.2.2 Tilakaavio (State Graph)

Tilakaavio luodaan ominaisuustiedostoon nimeltä tilamakro (State macro), jota sitten pelitilassa suoritetaan sille tarkoitetulla tilakoneella. Tilakaavio sisältää itsenäisiä kulkutiloja, joihin luodaan kulkukaaviolla peliobjektin tilaa kuvaava logiikka esim. jahtaa, hyökkää, vaella, pakene. Kulkutilojen välistä liikkumista hallitaan siirtymillä, jotka ovat kulkukaaviota, joihin asetetaan erilaisia siirtymäehtoja, kuten onko pelaaja tietyn etäisyyden päässä. (Low-scope, 2019; Unity, 2020)

3 Unityn tekoäly ja niiden käyttökohteet

Tässä luvussa käsitellään kolmea erilaista tekoälytyyppiä Unity-pelimoottorille ja niiden Unity-työkaluja.

3.1 Reaktiivinen tekoäly (Reactive AI)

Reaktiivinen tekoäly tarkoittaa tekoälyjärjestelmää, jolla ei ole muistoja tai aikaisempaa kokemusta nykyisestä päätöksestään. Tähän tekoälyyn kuuluu, että tietokone havaitsee maailman suoraan ja toimii se mukaan, mitä näkee. Tästä täydellinen esimerkki on Deep Blue, IBM:n shakkipelin supertietokone, joka voitti shakin suurmestarin Garry Kasparovin 1990-luvun lopussa. Deep Blue pystyy tehdä ennusteita, mitä liikkeitä sille tai vastustajalle voisi seuraavaksi tapahtua ja valita optimaalisimman siirron mahdollisista vaihtoehdoista. Tästä huolimatta tekoälyllä ei ole mitään käsitystä menneisyydestä tai aikaisemmin tapahtuneesta. (The conversation, 2016)

3.1.1 C# -ohjelmointikieli

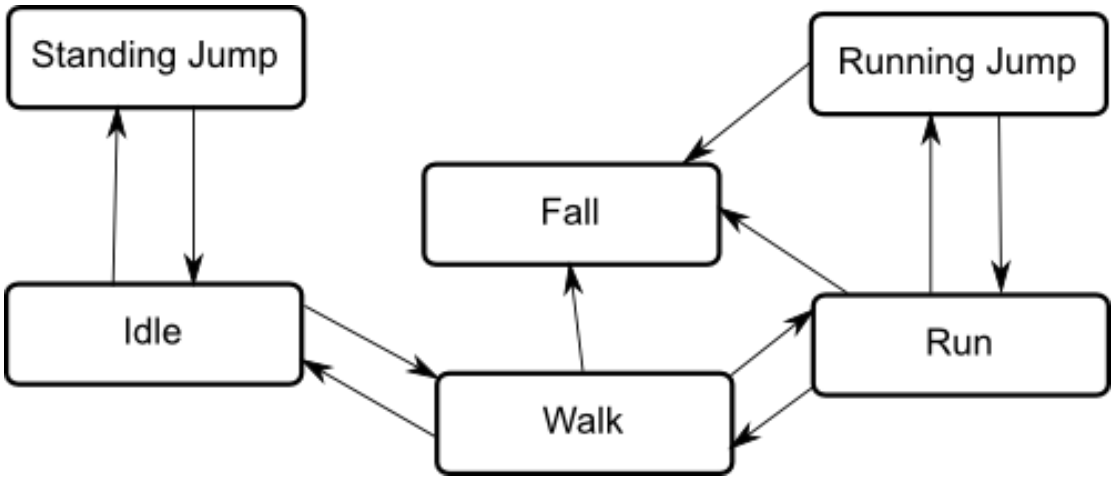
Kaikki Unityn käyttämät kielet ovat olio-ohjelmointikieliä ja näistä yleisin on C#-kieli (Gamedesigning, 2020; Unity, 2020). C# on monipuolinen ja aloittelijaystävällinen ohjelmointikieli, jossa on paljon yhteistä muiden ohjelmointikielien, kuten C:n ja Javan kanssa. (Android Authority, 2020) Tekoälyn tekeminen C#-kielellä on täysin käyttäjän omien taitojen ja tietämyksen varassa, jolloin turhautuminen ohjelman toimimattomuuteen voi tulla nopeasti vastaan. Opetusmateriaalit ja videot toki auttavat Unitylle tekoälyn tekemisessä, mutta niiden opetuksen tasot saattavat vaihdella niin paljon, ettei koodin ymmärtämisestä ja seuraamisesta tule mitään. Sen takia aloittelevalle pelinkehittäjälle on suositeltavaa hyödyntää Unity-pelimoottorin tarjoamia tekoälyjä pelin tekemiseen. (Gamedesigning, 2020)

3.1.2 Äärellinen tilakone (Finite State Machine)

Äärellinen tilakone (kuva 3) on malli, jota käytetään kuvaamaan ja säätämään suorituksen kulkua. Tyypillisesti sitä käytetään jaksologiikkapiireissä ja tietokoneohjelmissa. Äärellisen tilakoneen tekeminen aloitetaan luomalla tiloja, jotka kuvastavat peliobjektin toimintoja pelimaailmassa. Näiden tilojen välille asetetaan pelitilannetta vastaava logiikka käyttämällä siirtymiä, joita ohjelma

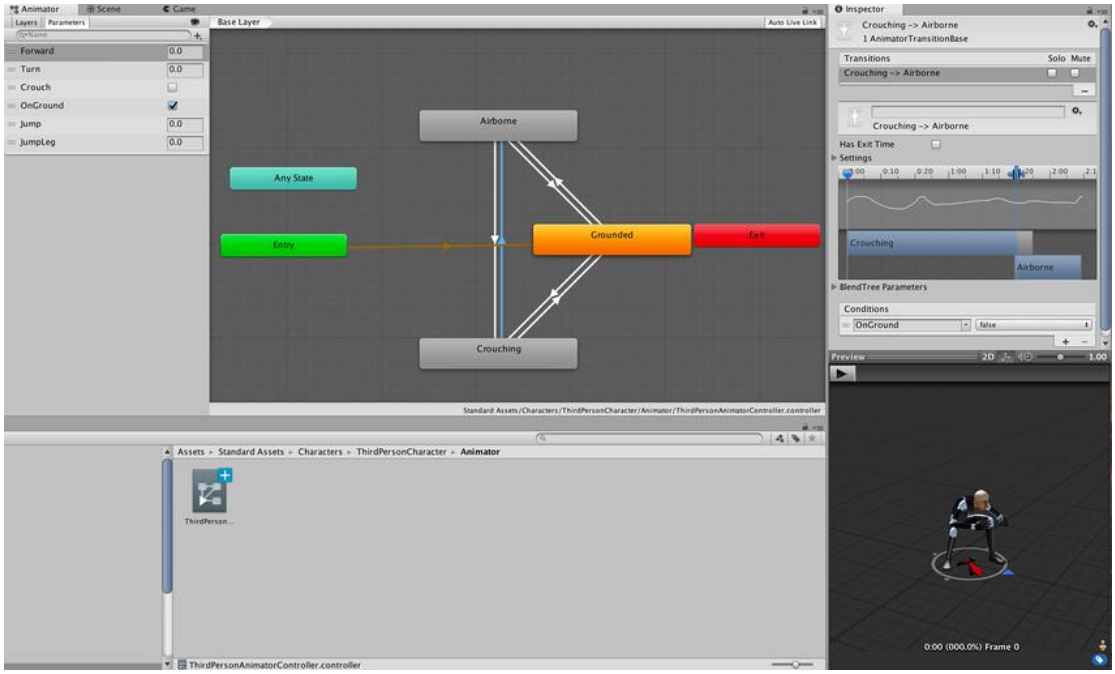
suorittaa niin kauan, kunnes haluttu lopputulos saavutetaan. (Syahputra, M. Arippa, A. Rahmat, R, Andayani, U, 2019, s. 3)

Kuva 3 Äärellistä tilakoneesta esimerkkikuva. (Unity documentation, 2020)



Unitylla on kolme tapaa tehdä tekoäly tilakoneella: kirjoittaa se kokonaan itse C#:lla, ostaa lisäosa kolmannen osapuolen valmistajilta Unity Assets Storesta tai käyttää Unityn sisäänrakennettua animaatiojärjestelmää (Kuva 4). Itsekirjoitettu toteutus toimii sille suunnitellulle tehtävälle, mutta jos mallia tarvitsee uudelleen pelin logiikan luonnissa, voi huomata kirjoittavansa samaa koodia uudelleen ja uudelleen. Assets Storesta löytyvien kolmannen osapuolen tilakoneominaisuuksien etuja ovat tehokkuus ja selkeä käyttöliittymä, mutta kolmannen osapuolen ominaisuuden investointihinta ja mahdolliset integrointiongelmat Unityn kanssa voivat estää aloittelevaa pelinkehittäjän tarttumasta näihin ladattaviin työkaluihin. Unityn animaatiojärjestelmä on pelimoottoriin sisäänrakennettu tilakone, joka on suunniteltu pelianimaatioiden luomiseen. Se sisältää tarvittavat työkalut tilakoneen tekoälyn muokkaamiseen ja visualisointiin pelitilan aikana. Käyttämällä animaatiojärjestelmää on mahdollista kopioida sama tilakone muille peliobjekteille. (Tsung, 2016; ks. myös Unity documentation, 2020)

Kuva 4 Unityn animaatiojärjestelmä ”Animator”. (Unity documentation, 2020)



3.1.3 Käyttäytymispuu (Behavior Tree)

Käyttäytymispuu on tapa luoda monimutkaisia tekoälyjärjestelmiä, jotka ovat modulaarisia ja reaktiivisia. Se sai nimensä hierarkkisesta, haarautuvasta solmujärjestelmästä, joka muistuttaa puuta. Rakenteen alkua kutsutaan juureksi (root), sen sisäisiä solmuja kutsutaan ohjausvirtaussolmuiksi (control flow nodes) ja niistä jakautuvia lehtisolmuja kutsutaan suoritussolmuiksi (execution nodes). Solmut voivat edustaa testejä tai käyttäytymistä. Juuren, ohjausvirtasolmun ja lehtisolmun liittämistä toisiinsa kuvaillaan terminologialla vanhempi ja lapsi. Suorittaminen aloitetaan puun juuresta, josta tieto kulkee sen lapsiin ohjausvirtasolmuihin, mistä se jatkaa jokaisen lehtisolmulapsensa läpi, kunnes ehto täyttyy. (Colledanchise & Ögren, 2018, s. 3–6)

Unityn omaa versiota käyttäytymispuusta, kuten animaatiojärjestelmän tilakoneesta, ei pelimoottorista löydy, mutta sen saa käyttöön ohjelmoimalla sen itse C#-kielellä tai lataamalla kolmannen osapuolen maksullisen visuaalisen ohjelmointieditorin Unity Assets Storesta.

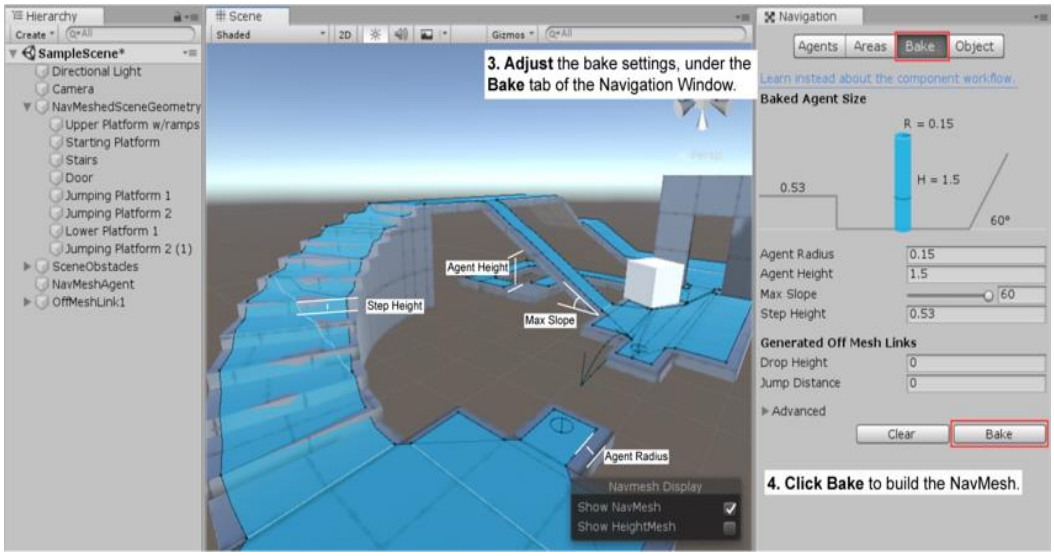
3.2 Harkitseva tekoäly (Deliberative AI)

Harkitseva tekoäly toimii ratkaisemalla ongelmia, mikä vastaa optimaalisten ratkaisujen etsimistä isosta ratkaisutilasta. Tekoälyn käyttäytyminen luodaan ratkaisemalla hyvin määriteltyjä ongelmia, jotka on kehystetty ympäristömalliin. Malleihin kehittäjä luo halutun käyttäytymisen ja näitä malleja kutsutaan suunnittelualueeksi. Harkitsevan tekoälyn huonona puolena on, että sen jokaisella hakualgoritmilla on rajoituksia käsiteltävien ongelmien kokoon nähden, jolloin pelin sujuvuus kärsii tekoälyn käsitellessä asetettuja käyttäytymismalleja. (Pangilinan, Lukas & Mohan, 2019)

3.2.1 Navigointiverkko (NavMesh)

Navigointiverkko (Kuva 5) on Unityn sisäänrakennettu navigointiverkkogeneraattori, jolla pystytään määrittämään pelimaailman kuljettavia pintoja. Navigointiverkko etsii peliobjektiin asetettavan tekoälyagentti-komponentin avulla lyhimmän mahdollisen reitin pisteiden välillä. Luomisprosessia tasogeometriasta kutsutaan navigointiverkkoleivonnaksi. Prosessi kerää kaikki pelikentän määritetyt kuljettavat pinnat ja esteet, jotka on merkitty staattisella navigoinnilla, ja luo sen pohjalta määritetyn navigointiverkon. (Barrara, Kyaw, Naing, 2018; Unity documentation, 2020) Bolt Visual Scripting hyödyntää kulku- ja tilakaavioissa navigointiverkkoa liikkumistekoälyn luomisessa navigointiagentti nimisellä solmukomponentilla.

Kuva 5 Navigointiverkko. (Unity documentation, 2020)



3.2.2 Tavoitteellinen toiminnan suunnittelu (Goal Oriented Action Planner: GOAP)

Tavoitteellinen toiminnan suunnittelu on suunnitteluarkkitehtuuri, joka toimii pelien autonomisen hahmokäyttäytymisen reaaliaikaiseen hallintaan. Tavoitteellisella toiminnan suunnittelulla irrotetaan koodilla suunnitelmat toisistaan, jolloin kutakin toimintoa voidaan muokata ilman, että muut suunnitelmat eivät häiriinny. Suunnitelmien irrottamisella toisistaan myös koodista tulee modulaarisempi ja helpommin ylläpidettävä, jolloin toimintojen lisääminen ja poistaminen vaivattomasti mahdollistuu. (Chaudhari, 2017)

Tavoitteellisen toiminnan suunnittelun tekoälyjärjestelmä tarjoaa itsenäisille agenteille kyvyn suunnitella toimintasarjan dynaamisesti tavoitteen saavuttamiseksi annetuilla ehdoilla. Agentin valitsema toimintasarja määräytyy agentin sekä pelimaailman nykyisestä suunnitelmasta, jolloin agentit pystyvät valitsemaan älykkäämmin intuitiivisimman toimintasarjansa, vaikka niiden ehdoksi olisi asetettu sama tavoite. Agentti toimittaa kohdetavoitteen yhdessä pelimaailmantilan ja luettelon kelvollisista toiminnoista suunnittelijalle. Tätä prosessia kutsutaan suunnitelman laatimiseksi. Tavoitteellisen toiminnan suunnittelu-suunnittelija (GOAP planner) tarkastelee suunnitelman laatimia edellytyksiä ja vaikutuksia määritelläkseen toimintojonon tavoitteen saavuttamiseksi. Jos se onnistuu, palauttaa se agentille suunnitelman, jota seurata. Tavoitteellisen toiminnan suunnittelun luodaan C#-kielellä. (Chaudhari, 2017)

3.3 Koneoppiminen (Machine Learning)

Koneoppiminen on tekoälyn haara, joka luo menetelmiä ja algoritmimalleja automaattisesti datasta. Toisin kuin muut tekoälyhaarat, jotka on rajattu selkeillä säännöillä, koneoppiminen oppii kokemuksista ja tallentaa sen myöhempää käyttöä varten. Sääntöihin perustuva tekoäly suorittaa tehtävänsä samalla tavalla kuin koneoppimisjärjestelmän suorituskykyä voidaan kouluttaa altistamalla algoritmia uusilla tiedoilla. (Heller, 2019) Unityssa hyödynnetään jäljitelmäoppimista (Imitation Learning) ja vahvistusoppimista (Reinforcement Learning).

3.3.1 Jäljitelmäoppiminen (Imitation Learning)

Jäljitelmäoppimisen tarkoituksena on, että agentille syötetään tietoa ympäristöstä ja tekoäly jäljittelee sitä parhaansa mukaan. Asiantuntija (tyypillisesti ihminen) antaa tekoälyagentille sarjan näyttöjä, joita agentti kartoittaa havaintojen ja toimintojen avulla tehtävän optimaalisen suorituksen saamiseksi. Jäljitelmäoppiminen on pelinkehittämisessä saavuttanut suosiota, koska sillä on helppo opettaa agenteille monimutkaisia tehtäviä ilman tarvetta suunnitella palkitsemisfunktiota tietyille tehtäville. (Perhanidis, 2020)

3.3.2 Vahvistusoppiminen (Reinforcement Learning)

Vahvistusoppimisen tarkoituksena on kouluttaa agenttia vastaamaan ympäristöä kokeilemalla ja erehtymällä. Hyvä esimerkki tästä on DeepMind’s AlphaGo, joka oppiakseen Go-pelin, matki ihmispelaajia suuresta historiallisten pelien joukosta. Sen jälkeen DeepMind’s AlphaGo vahvisti oppimistaan suurella määrällä itseään vastaan pelaamia Go-pelejä kokeilemalla ja erehtymällä. (Heller, 2019)

Unity julkaisi hiljattain avoimen lähdekoodin koneoppimiseen suunniteltuun laajennukseen, jonka avulla peliympäristöjä voidaan käyttää agenttien koulutukseen. Agenttien koulutus toteutetaan erillisen Python-sovellusohjelmointirajapinnan kautta. (Choudhury, 2020; Steiger, 2020; Perhanidis, 2020)

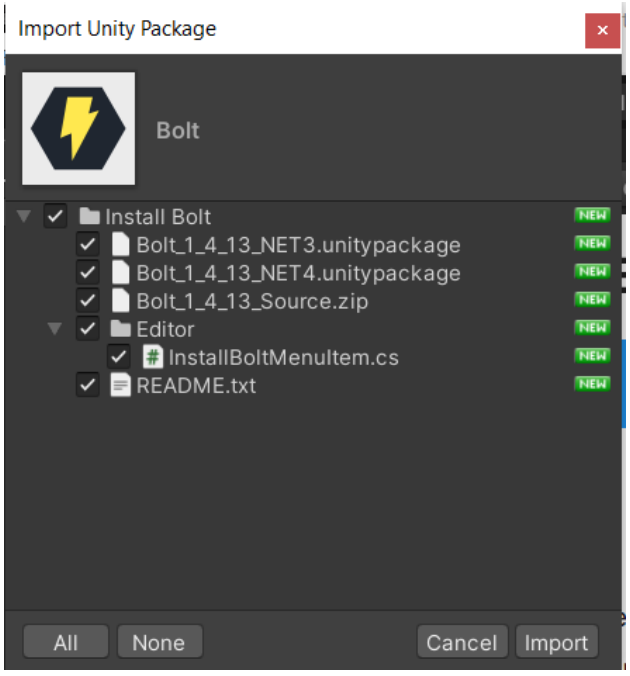
4 Bolt tekoälyn tekeminen Unitylle

Luvussa käydään läpi Bolt Visual Scriptin asentaminen Unity-pelimoottorille, jota hyödyntäen toteutetaan vaiheittain Player-peliobjektin liikkuminen pelikentällä ja Monster-peliobjektin tekoälylogiikka.

4.1 Boltin asennus Unitylle

Boltin asennus Unity-pelimoottorille on yksinkertaista. Unityn ylävalikosta löytyy Window-painike, jonka alavalikosta valitaan Assets Store. Avautuvan Assets Store nettisivun hakukenttään kirjoitetaan Bolt, jolloin hakutulokset tulevat esiin. Näistä vaihtoehdoista valitaan Unity Technology Bolt. Boltin lataussivulta löytyy asennuspaketin kaikki tekniset tiedot ja käyttäjien arvostelut Boltista. Lataaminen Unity-pelimoottorille aloitetaan latauspainikkeella, jonka jälkeen ladattu asennuspaketti (Kuva 6) tuodaan Unity projektin ominaisuuskansioon.

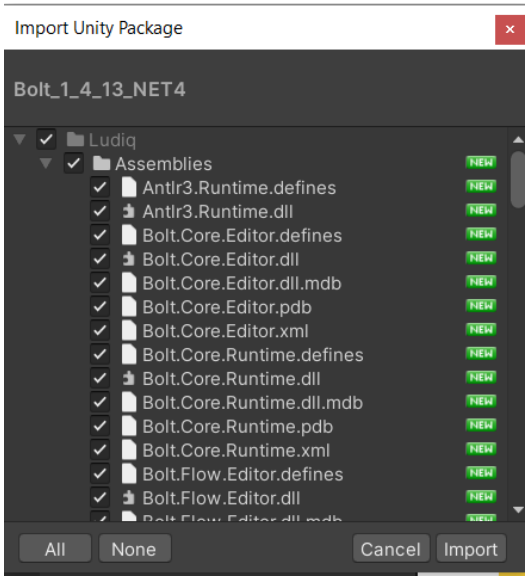
Kuva 6 Assets Storesta ladattu Bolt-asennuspaketti.



Seuraavaksi avataan projektin ominaisuuskansiosta löytyvä Install Bolt -kansio, josta löytyy kaksi asennusversiota (scripting runtime version) .NET 3 ja .NET 4 (Kuva 7). Näistä .NET 3 on suunniteltu Unity 2019.x versiota vanhemmille Unity-versioille, jolloin projektissa käytettävään 2019.x versiolle

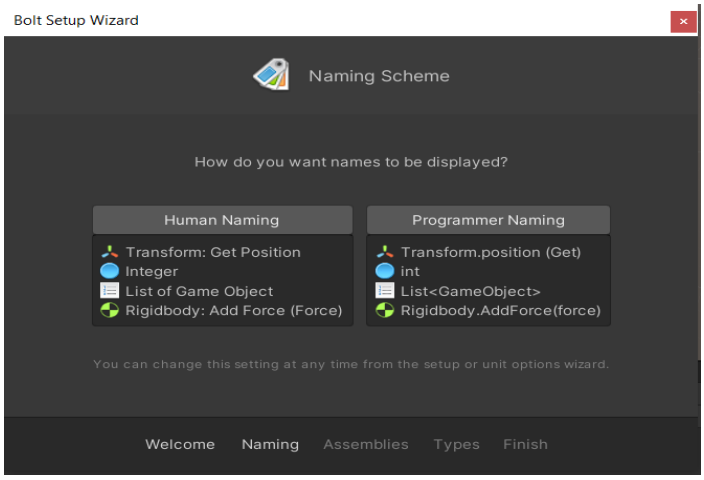
asennetaan .NET 4-asennuspaketti. Asennusversion voi varmistaa valitsemalla Unityn ylävalikon Tools-painike ja sieltä Install Bolt, jolloin Unity avaa oikean .NET version asennettavaksi.

Kuva 7 .NET 4-Asennuspaketti.

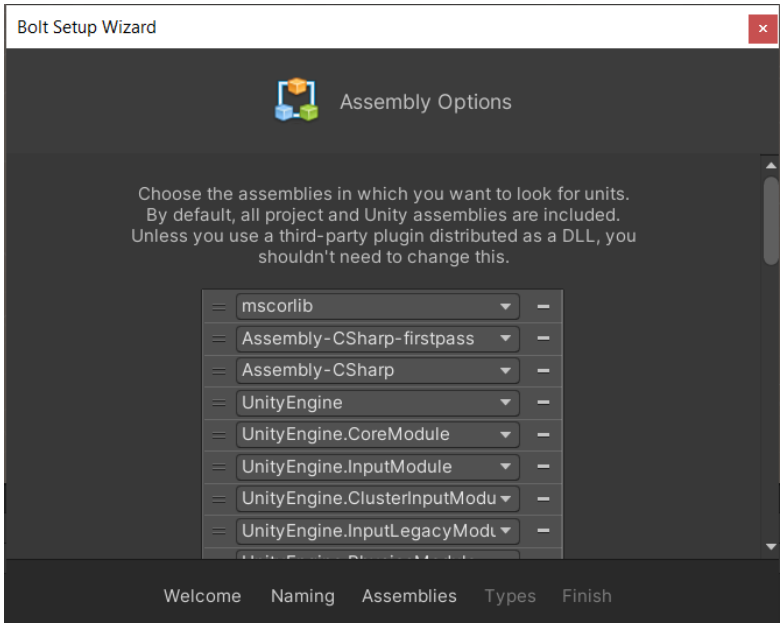


Valittu .NET-asennuspaketti tuodaan, jolloin avautuu ohjatun asennustoiminnon ikkuna. Ensimmäisenä muuttujat nimetään joko ohjelmointi- tai käyttäjäystävällisellä nimeämisellä (Kuva 8). Seuraavaksi avautuu Assembly Options (kuva 9) ja Type Options (kuva 10) -ikkunat, näihin ei tarvitse tehdä tässä asennusesimerkissä muutoksia. Type Options -ikkunan lopussa löytyy generate-painike, josta asennus suoritetaan. Asennuksen jälkeen projektin ominaisuuskansioon on luotu Ludig-kansio, joka sisältää Bolt Visual Scripting ominaisuudet. Tämän jälkeen Install Bolt -kansiolla ei ole enää käyttöä ja sen voi poistaa.

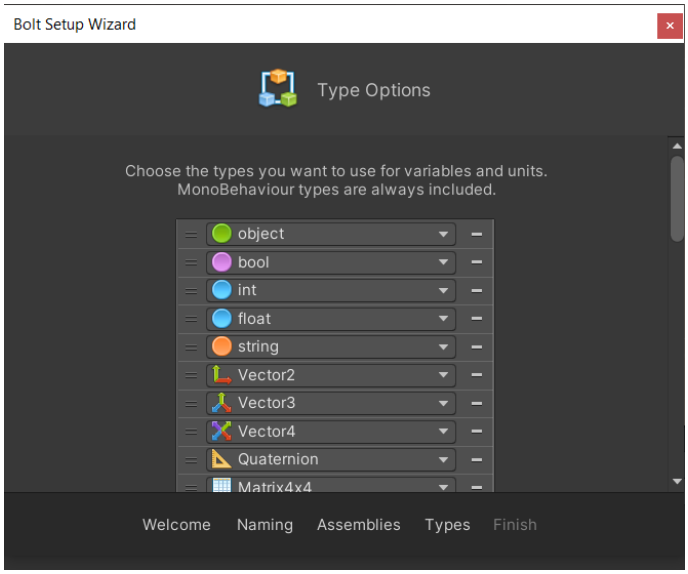
Kuva 8 Naming Scheme -ikkuna.



Kuva 9 Assembly Options -ikkuna.

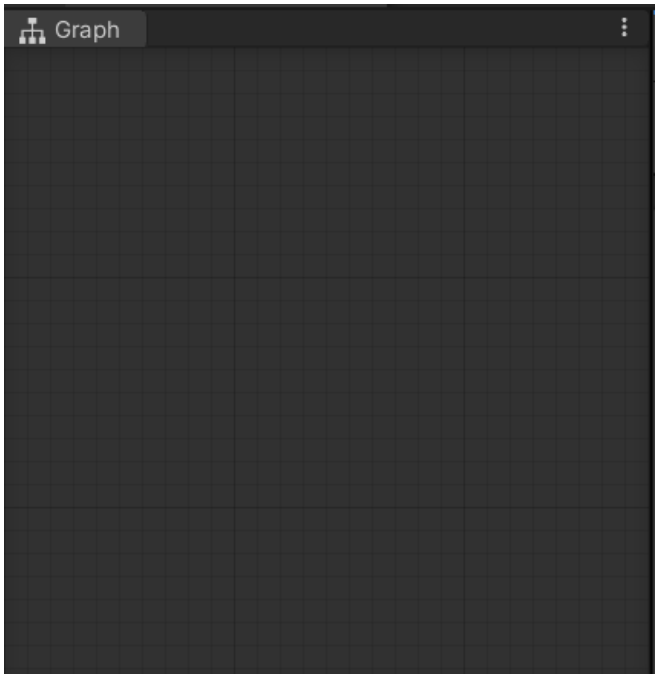


Kuva 10 Type Options -ikkuna.

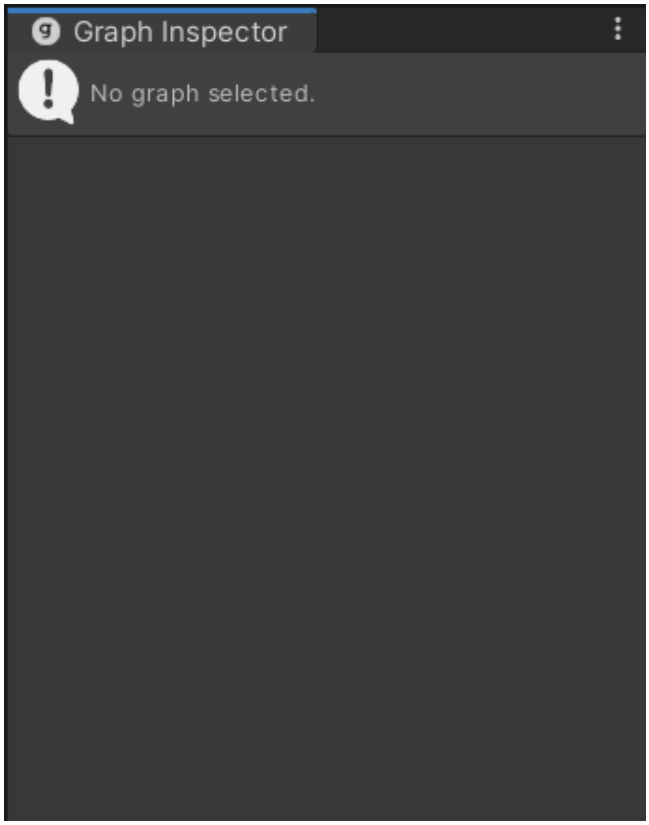


Boltin asennuksen jälkeen Unity-pelimoottorin Window ylävalikkoon on tullut kolme uutta ikkunaa: Graph (Kuva 11), Graph Inspector (Kuva 12) ja Variables (Kuva 13). Graph-ikkunalla visuaalista ohjelmointia suoritetaan ja seurataan reaaliaikaisesti. Graph Inspector -ikkuna näyttää valitun elementin sisältävän informaation. Variables-ikkunalla pystyy lisäämään muuttujia peliobjekteille julkisesti tai yksityisesti.

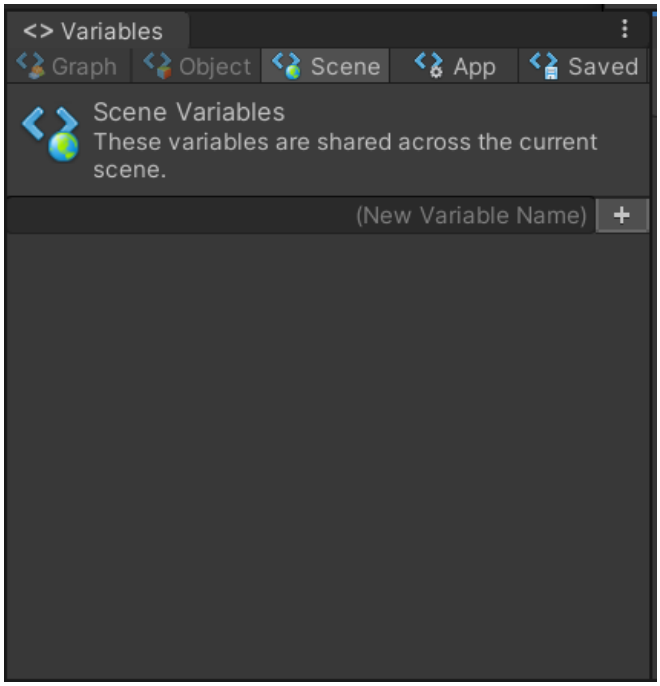
Kuva 11 Graph-ikkuna.



Kuva 12 Grap Inspector -ikkuna.



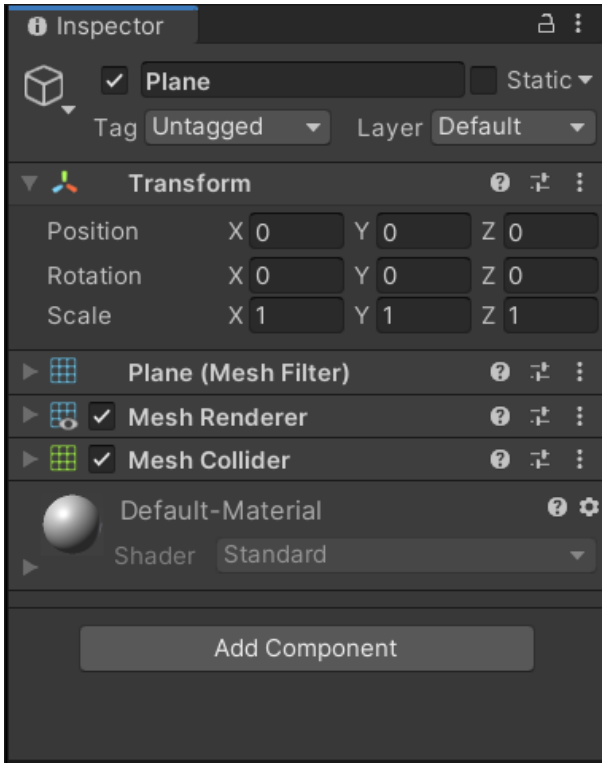
Kuva 13 Variables-ikkuna.



4.2 Pelikentän ja hahmojen luonti

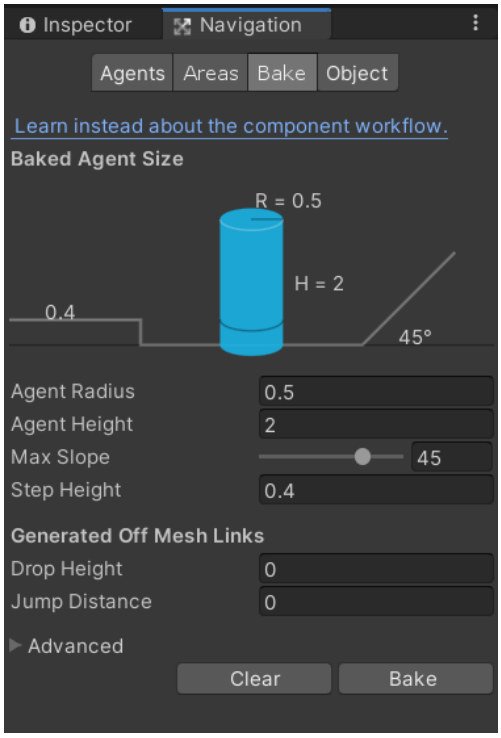
Boltin asentamisen jälkeen aloitetaan pelikentän luominen. Unityn ylävalikosta valitaan Gameobject - 3D object - Plane. Inspector-ikkunassa (Kuva 14) kirjoitetaan Plane ja sen Layer nimetään Ground:iksi ja asetetaan Scale x ja y-akselien arvoksi kaksi, jolloin pelikentän sivut kasvavat 2 kertaa 10 X 10 metriä. Groundille luodaan materiaali, joka muuttaa värin kuvastamaan maata, ja se tehdään painamalla projektin ominaisuuskansiossa hiiren oikeaa nappia ja valitsemalla Create - Material. Seuraavaksi luodaan pelikentälle seinät ylävalikosta Gameobject - 3D object - Cube. Cube ja sen Layer nimetään Wall, nämä peliobjektit muokataan Unityn työkaluilla halutun muotoisiksi ja sijoitetaan pelikentällä labyrinttimäiseksi sokkeloksi. Ground ja Wall peliobjektien Static-valintaruutu aktivoidaan, koska ne halutaan sisällyttää pelikentän tulevaan NavMesh-leivontaan.

Kuva 14 Plane-peliobjektin Inspector-ikkuna.



NavMash-toiminnot löytyvät siirtymällä ylävalikossa Window - AI - Navigation. Navigation-ikkunan (Kuva 15) ominaisuuksissa löytyy: Agents, Areas, Bake ja Object. Agents-ominaisuudessa muokataan agentin korkeutta ja kulkukaltevuuksia pelikentällä. Areas-ominaisuudessa määritetään agentin kulkukustannuksin alueittain. Bake-ominaisuudella leivotaan määritelty navigointiverkko pelikentälle. Object-ominaisuudella määritellään peliobjektin staattisuus ja mille Areas-alueelle se kuuluu. Aktivoimalla Bake-ominaisuuden luodaan pelikentälle sinisen värinen alue, jolla NavMeshAgentti-komponentilla varustettu peliobjekti pystyy navigoimaan.

Kuva 15 Navigation-ikkuna.



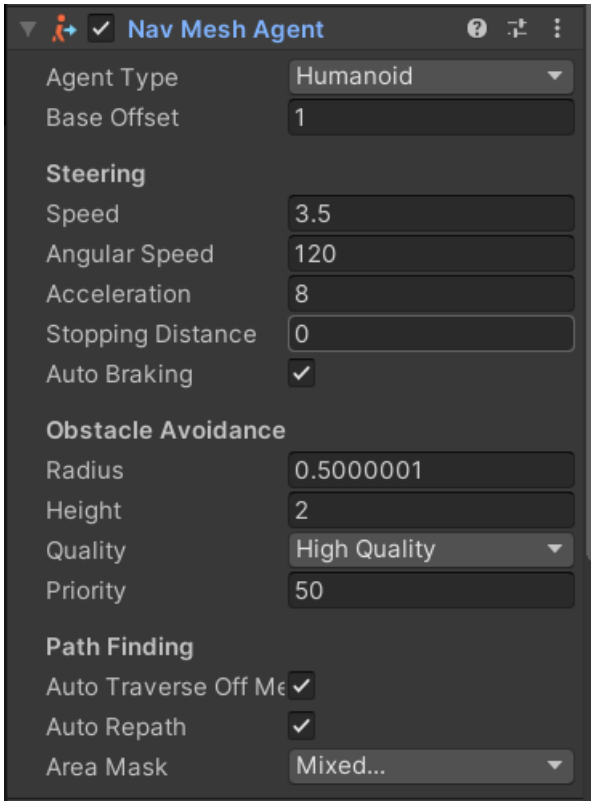
Pelihahmojen luomiseen käytetään ylävalikon Gameobject - 3D object - Cylinder. Lieriötä luodaan kolme ja niistä muokataan erilaisia, jotta ne erottuvat helpommin toisistaan, nämä saavat nimeksi Player, Monster ja Ally. Kaikille lieriölle luodaan eriväriset materiaalit: Player saa sinisen, Monster punaisen ja Ally keltaisen. Player ja Monster -peliobjekteihin liitetään laatikot silmiksi osoittamaan z-akselin suuntaa, joka on Unityssa peliobjektin etupuoli ja kulkusuunta. Lopuksi lisätään Tagi Playerille, joka löytyy valmiina Tag-valikosta ja Allylle lisätään Add Tag -toiminnolla Ally niminen tag. Tageja käytetään ohjelmoinnissa löytämään peliobjektin pelikentän sijainti.

4.3 Pelaajan liikkuminen Boltin avulla

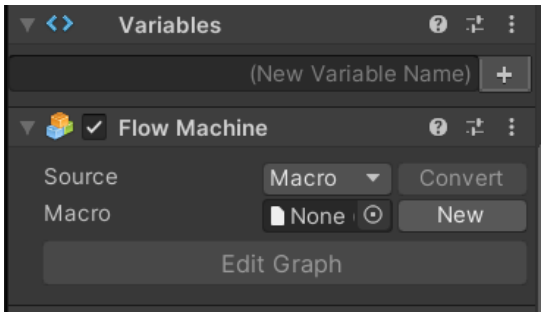
Player-peliobjektin Inspector-ikkunasta löytyvällä Add Component -toiminnolla haetaan Nav Mesh Agent (Kuva 16) ja Flow Machine -komponentit (Kuva 17). Haettu Flow Machine luo samalla Variables-komponentin, johon Player-peliobjektin yksityiset muuttujat voidaan luoda. Seuraavaksi luodaan Movement-makro, joka sisältää kulkukaavion, johon Bolt Visual Scriptillä luodaan pelilogiikkaa. Movement-makroa varten on hyvä luoda Macro niminen tiedosto projektin ominaisuuskansioon, jolloin ominaisuuskansio ja makrot pysyvät hyvässä järjestyksessä. Luotu

Movement-makro avautuu Player-objektin Graph-ikkunassa Flow Graphina (Kuva 18), jolloin liikkumislogiikan visuaalinen ohjelmointi voi alkaa.

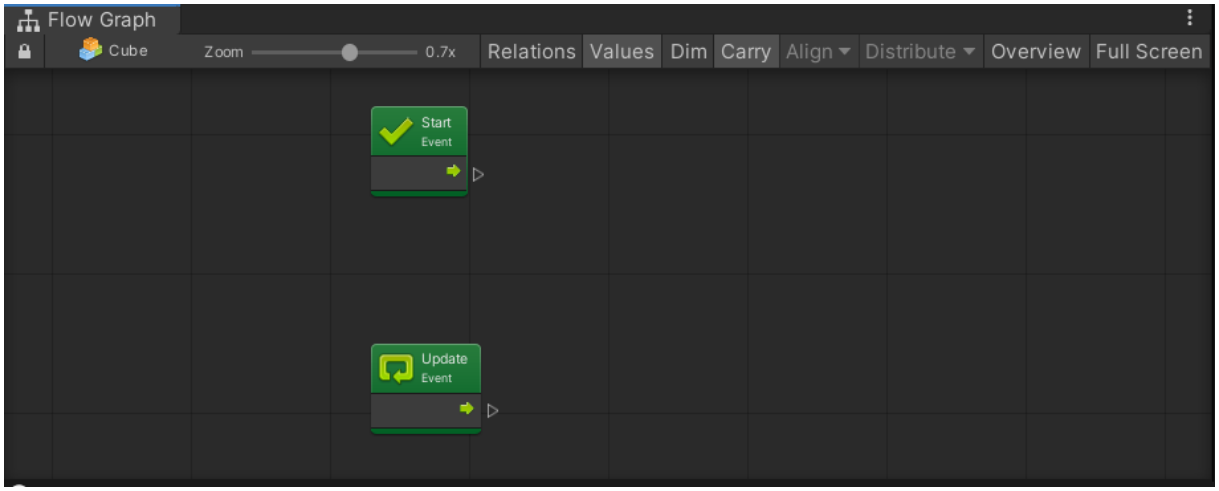
Kuva 16 Nav Mesh Agent -komponentti.



Kuva 17 Flow Machine ja Variables -komponentit.

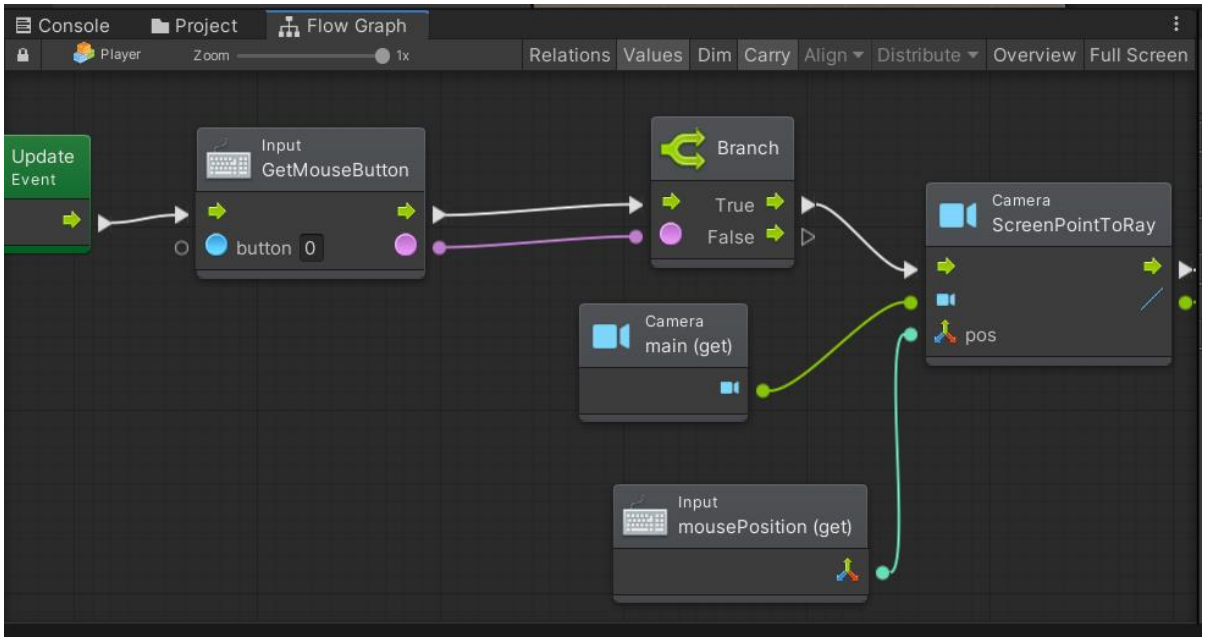


Kuva 18 Luotu Player-peliobjektin Movement-makro.



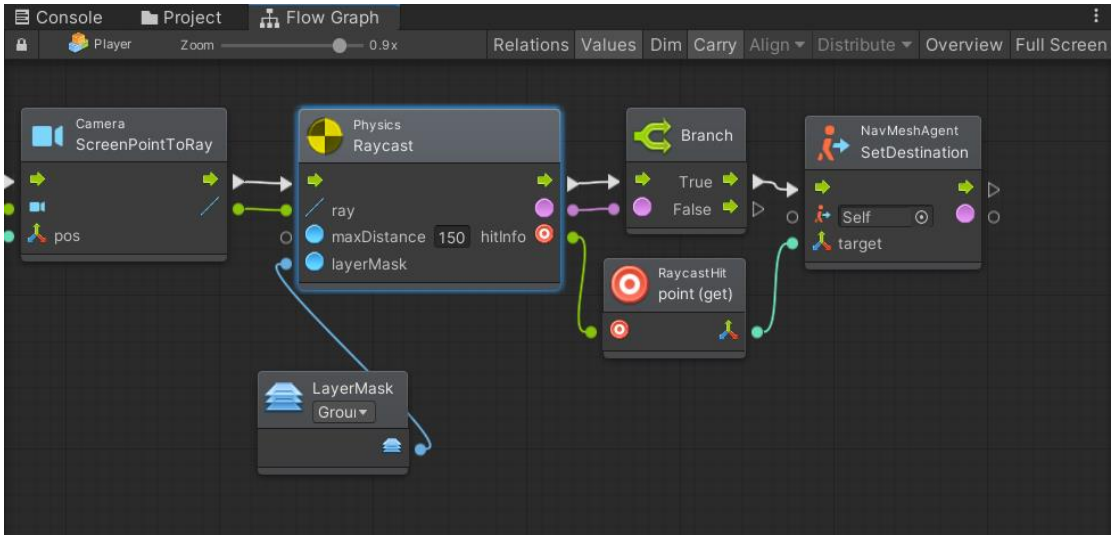
Makron kulkukaavio sisältää Start- ja Update Eventin, nämä toimivat ohjelmoinnin virtauksen lähtökohtina ja näkyvät vihreinä yksikköinä. Start Eventia kutsutaan vain kerran kaavion luonnissa ja Update Eventiä päivitetään joka sekunti. Start Event voidaan poistaa kulkukaaviosta, koska sitä ei tarvita esimerkin liikkumislogiikan luomisessa. Update-yksikön Output-lähdöstä vedetään nuoli haluttuun kohtaan, jolloin ilmestyy hakutoiminto nimeltään Fuzzy Finder. Hakutoiminnon saa esiin myös painamalla hiiren oikeaa nappia ja painamalla Add Unit. Hakutoimintoa käyttäen haetaan Input GetMouseButton -yksikön, jolla tarkistetaan, painetaanko hiiren painiketta. Yksikölle pystytään määrittämään haluttu hiiren painike: 0 tarkoittaa vasenta painiketta, 1 oikeaa ja 2 keskimmäistä painiketta. Seuraavaksi haetaan Branch-yksikkö, jota käytetään määrittämään ohjelmointivirran suunta riippuen, onko jokin looginen (boolean) tosi (true) tai epätosi (false). Asetamme Branch-lähdön todeksi, jolloin ohjelmointivirta liikkuu vain, kun se on tosi. Seuraava askel on hakea Camera Screen Point to Ray -yksikkö, joka luo säteen hiiren osumiskohdasta pelimaailmaan pelinäytöllä. Yksikkö tarvitsee kaksi tietoa toimiakseen. Target-tuloon haettu Camera Main (get) -yksiköllä kerrotaan, että käytetään pääkameraa pelimaailman katsomiseen. Position-tuloon haetaan Input mousePosition (get) -yksikkö, joka antaa hiiren pelimaailman sijainnin (Kuva 19).

Kuva 19 Movement makron visuaalisen ohjelmoinnin vaiheet Updateista Camera ScreenPointToRayhin asti.



Seuraavaksi haetaan Physics Raycast -yksikkö, joka luo säteen pelimaailmaan ja havaitsee sen osumakohdan. Yksikkö sisältää ray, hitInfo, maxDistance ja layerMask. Ray-tulo ottaa vastaan tiedot Camera Screen Point to Ray -säteestä, maxDistance asetetaan 150, joka määrittää suurimman etäisyyden, jonka säteen tulisi tarkistaa törmäyksiltä. LayerMask-tuloon haetaan LayerMask Literal, jonka Layer-valikosta löytyy Ground, jolla määritetään haluttu osumakerros. Seuraavaksi haetaan Branch, johon Physics Raycastin ohjelmointivirta ja tosi-lähtö asetetaan. Lopuksi haetaan NavMeshAgent SetDestination -yksikkö, jota käytetään peliobjektin navigointiin pelikentällä. Yksikkö sisältää kaksi target tuloa: ensimmäiseen määritetään mihin peliobjektiin navigointi tehdään, toiseen asetettaan kohde, johon navigoidaan, tämä tieto saadaan RaycastHit Point (get) -yksiköllä, joka yhdistetään Physics Raycastin hitInfoon ja NavMeshAgentin targettiin (Kuva 20).

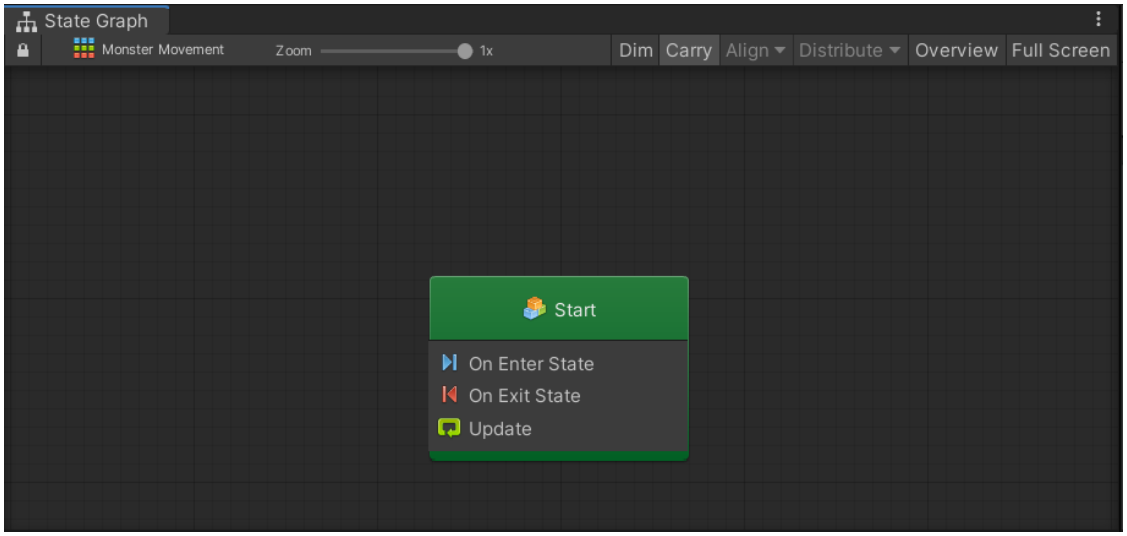
Kuva 20 Movement-makron visuaalisen ohjelmoinnin vaiheet Physics Raycastistä NavMeshAgenttiin.



4.4 Monster (Non-Player Character NPC)

Monster-peliobjektiin haetaan samalla tavalla komponentit, kuin Player-objektiin. Eroavaa on, että Flow Machinen sijaan haetaan State Machine, jolla pystytään jakamaan ohjelma pienempiin kulkutilaosiin. State Machinella luodaan Monster Movement niminen State makro (Kuva 21), joka voidaan sijoittaa projektin makrokansioon. Luotu makro sisältää Start-kulkutilan, jossa on valmiina kolme yksikköä: On Enter State Event, Update Event ja On Exit State Event. On Enter ja On Exit State pystytään määrittämään haluttu toiminto kulkutilaan ohjelmointivirran saapuessa ja poistuessa kulkutilasta. Tässä opinnäytetyöesimerkissä emme kuitenkaan tarvitse näitä Eventejä, joten ne voidaan poistaa kulkutilasta. Seuraavaksi Start-kulkutila nimetään uudelleen Idle-nimiseksi Graph Inspector -ikkunasta löytyvään Title-kohtaan, tällöin pystytään paremmin kuvaamaan kulkutilojen funktioita.

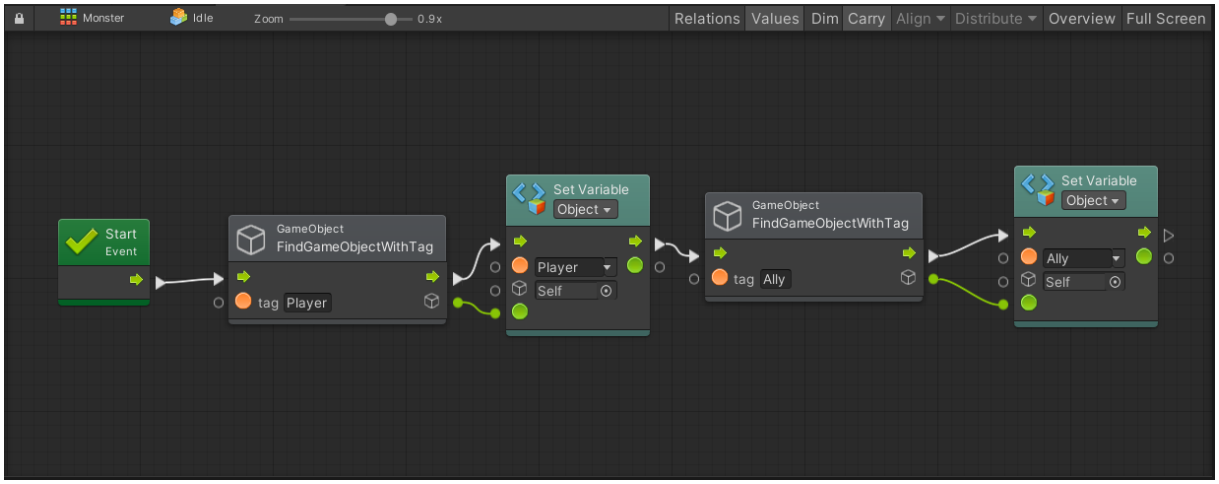
Kuva 21 Luotu Monster Movement -makro.



4.4.1 Idle-kulkutila

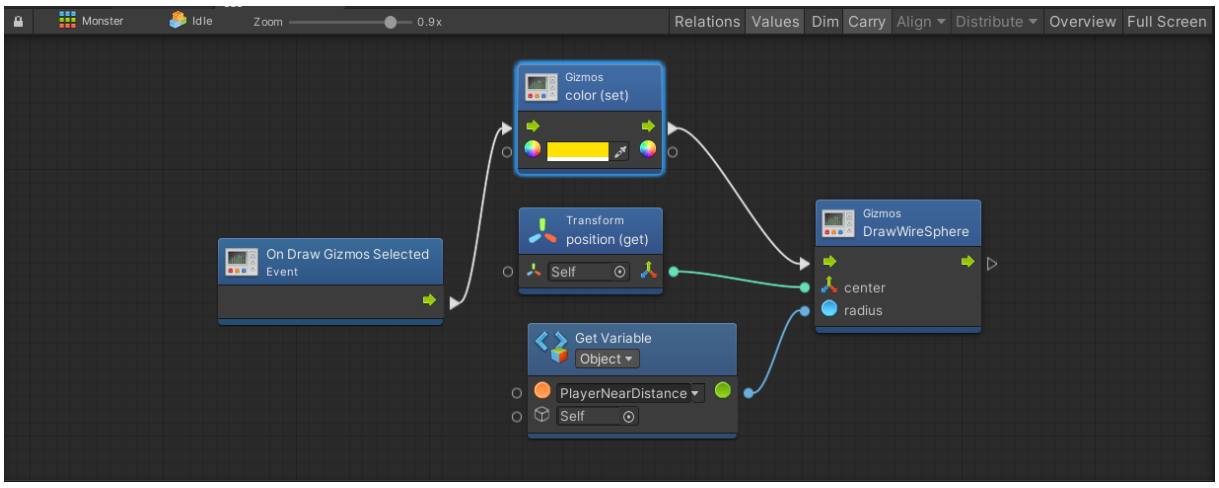
Idle-kulkutilan tekoälyn visuaalinen ohjelmointi aloitetaan hakemalla Start Event -yksikkö, jolloin pystytään löytämään Player- ja Ally-peliobjektien sijainnit pelimaailmalla pelin käynnistyessä. Monster-peliobjektin Object Variables -ikkunaan luodaan kaksi muuttujaa nimeltään Ally ja Player. Tyypiltä tyhjiksi jätettyihin muuttujiin varastoidaan visuaalisessa ohjelmoinnissa haetut peliobjektit. Start-yksikön Output-lähdöstä vedetään nuoli haluttuun kohtaan, jolloin ilmestyy hakutoiminto. Hakutoimintoa käyttäen haetaan Gameobject FindGameObjectWithTag -yksikkö, jonka tag-kohtaan kirjoitetaan tarkasti Player, jotta Bolt löytää haetun peliobjektin. Tämän jälkeen haetaan Set Object Variable -yksikkö, jonka Name-listasta löytyy aikaisemmin luodut Player-muuttuja. Haettu peliobjekti sijoitetaan vihreällä nuolella New Value kohtaan. Ohjelmointivirtaa jatketaan Set Variable Object yksiköstä ja samat vaiheet tehdään Ally-peliobjektin hakemiselle (Kuva 22).

Kuva 22 Player ja Ally-peliobjektien asettaminen muuttujiin pelin alussa.



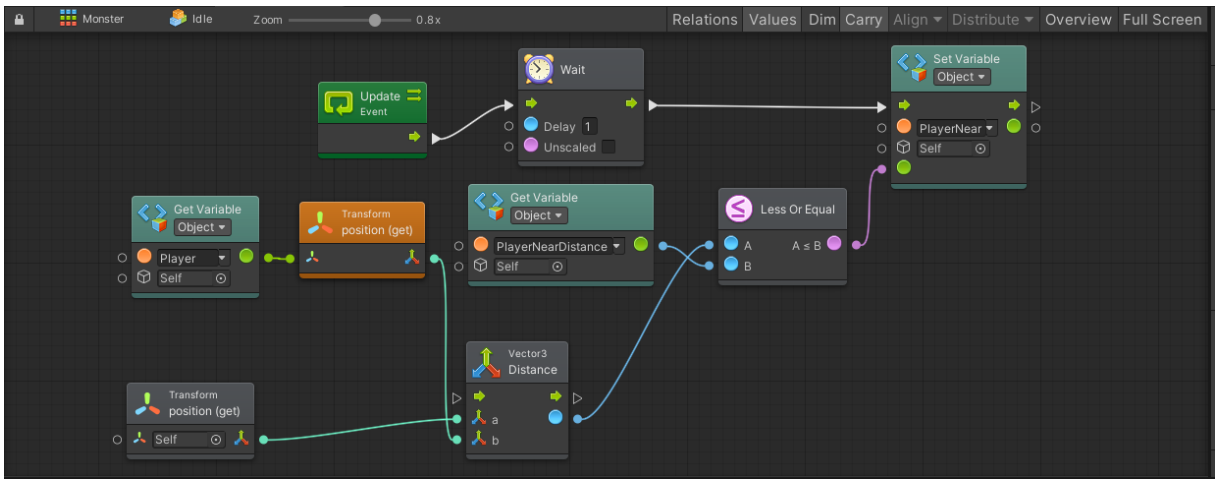
Seuraavaksi luodaan muuttuja PlayerNearDistance, jonka tyypiksi asetetaan float ja arvoksi kolme. Tätä muuttujaa käytetään Monster ja Player peliobjektien etäisyyden tarkkailuun. Jotta PlayerNearDistance-muuttujan kokoa olisi helpompi seurata Scene-ikkunaeditorissa, luodaan lankapallo Monster-peliobjektin ympärille On Draw Gizmos Selected Event -yksikön avulla. Yksiköstä vedetään nuoli ja etsitään hakutoiminnolla Gizmos Color (set) -yksikkö, jolla luodaan haluttu väri langalle. Seuraava askel on hakea Gizmos DrawWireSphere -yksikkö, joka sisältää center ja radius -muuttujat. Centerillä asetetaan lankapallon keskus haluttuun paikkaan, tämä saadaan hakemalla peliobjektin sijainti Transform Position (get) -yksiköllä. Radius määrittää pallon säteen, joka tiedot saadaan hakemalla Get Object Variable -yksikön PlayerNearDistance muuttujan tiedot (Kuva 23).

Kuva 23 Gizmos-yksiköt.



Seuraavana luodaan Object Variables -ikkunassa PlayerNear-muuttuja, jonka tyyppi on Boolean, tämä ottaa vastaan loogisen tiedon onko Player-objekti PlayerNearDisctance sisällä vai ei. Käytetään Update Event -yksikköä, jossa aktivoidaan sen Coroutine-toiminnon, tämä mahdollistaa ohjelmointivirran päivityksen pysäyttämisen Wait For Seconds -yksikön avulla. Wait For Seconds -yksikön Delay-muuttuja asetetaan ykköseen, jolloin luodaan sekunnin viive Monster-peliobjekti luontevampaan reagoimiseen pelimaailmassa. Seuraavaksi ohjelmointivirrassa haetaan Set Object Variable -yksikkö, johon asetetaan PlayerNear. Jotta voidaan vertailla Player ja Monster -objektien etäisyyttä, haetaan Player Get Object Variable -yksikkö ja asettamalla siihen Transform position (get) -yksikön, saadaan peliobjektin sijainti pelimaailmassa. Seuraavaksi haetaan Transform position (get), jossa on oletuksena Self-muuttuja. Nämä Transforms positionit liitetään Vector3 Distance -yksikköön, joka palauttaa peliobjektien etäisyyden toisistaan. Distance yhdistetään Less Or Equal -yksiköön, jolla tutkitaan, onko asetettu arvo pienempi tai yhtä suuri toiseen arvoon verrattuna. Tämän jälkeen haemme PlayerNearDistance-muuttuja, joka asetetaan Less Or Equal -yksikkön toiseksi vertailtavaksi arvoksi. Jos peliobjektien etäisyys on pienempi kuin PlayerNearDistance-muuttujan, asetetaan arvo tosi PlayerNear-muuttujalle, muussa tapauksessa looginen arvo on asetettu oletuksena epätodeksi (Kuva 24).

Kuva 24 Player ja Monster-peliobjektin etäisyys tarkkailu visuaalisella ohjelmoinnilla.

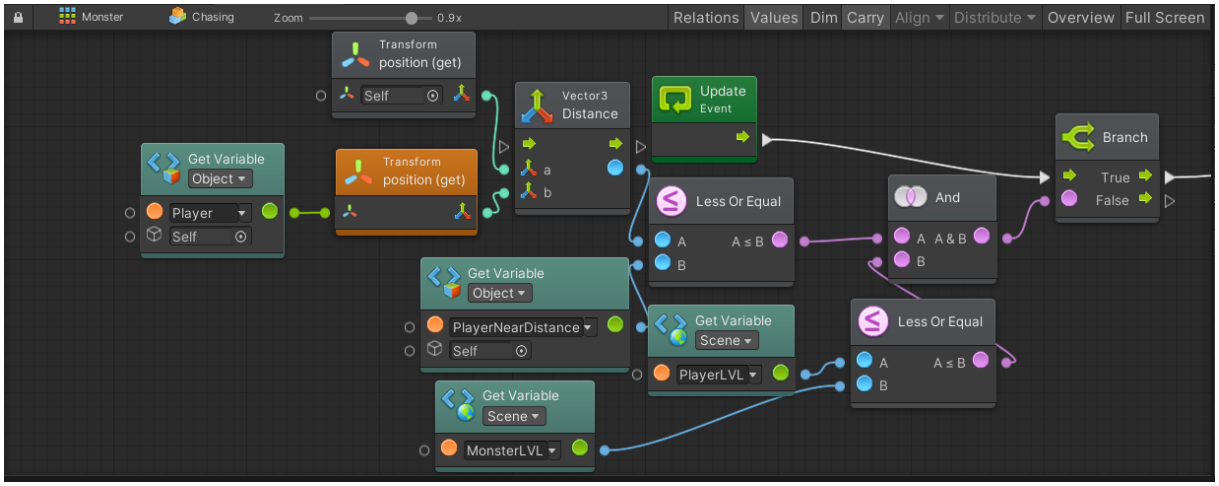


4.4.2 Chasing-kulkutila

Idle-kulkutilan valmistuttua, luodaan Chasing-kulkutila, johon luodaan Monster-peliobjektin jahtaamistekoäly. Kulkutila luodaan painamalla hiiren oikeaa nappia State Graph -ikkunassa ja valitsemalla valikosta Create Flow State. Painetaan kerran luotua kulkutilaa, jolloin pystytään Graph Inspector-ikkunassa nimeämään sen nimellä Chasing. Poistamme ennen visuaalista ohjelmointia kulkutilasta kaiken muun paitsi Update Eventin. Chasing-tekoälylogiikka rakennetaan kolmesta ohjelmointiosasta, joista kahdessa tarkkaillaan etäisyyksien täyttymistä ja peliobjektien tasoja toisiinsa. Viimeisessä ohjelmointiosassa määritetään, jahtaako Monster Playeriä vai pakenee.

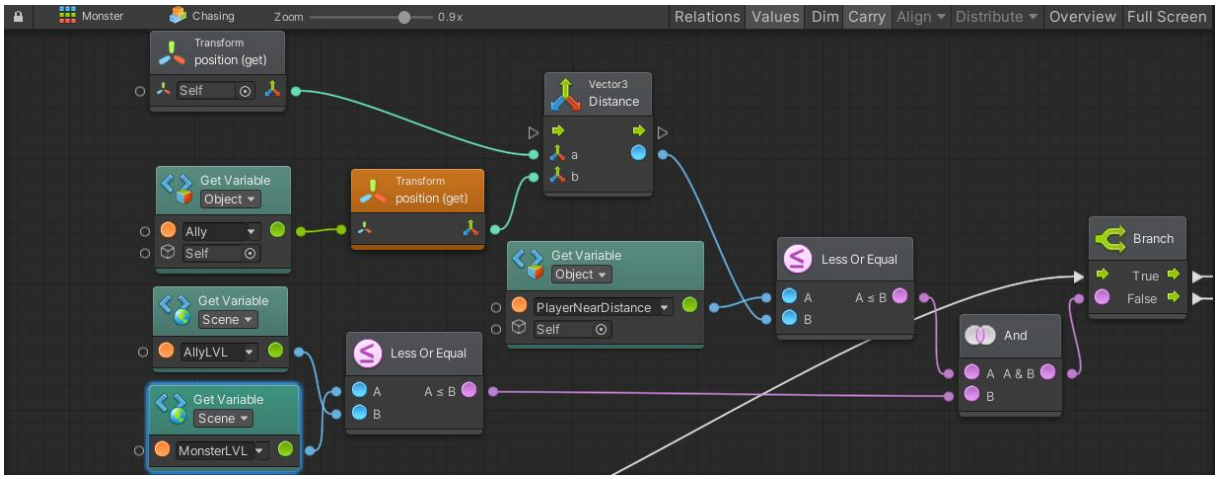
Ensimmäisessä ohjelmointiosassa vedetään Update-yksiköstä nuoli Branch-yksikölle, johon liitetään kahden Less Or Equal -yksiköstä saatu lopputulos. Ensimmäisessä tutkitaan samalla tavalla kuin Idle-kulkutilassa Player ja Monster -peliobjektien etäisyyttä toisiinsa. Jos peliobjektien välinen etäisyys on pienempi kuin PlayerNearDistance, palautetaan yksiköstä tosi. Seuraavaksi luodaan Variables-ikkunasta löytyvällä Scene Variablesin avulla julkisia muuttujia, joihin kaikilla projektin objekteilla on pääsy. Luodaan muuttujat PlayerLVL, jonka arvoksi asetetaan yksi, MonsterLVL arvoksi asetetaan kaksi ja AllyLVL arvoksi kolme, näiden tyypiksi asetetaan Integer. Toisessa Less Or Equal -yksiköllä verrataan PlayerLVL lukua MonsterLVLin lukuun: jos se luku on pienempi, palautuu yksiköstä tosi. Kahdesta Less Or Equal -yksiköstä vedetään nuolet And-yksikköön, joka palauttaa Branch-yksikölle tosi vain, jos molemmat Less Or Equal -yksiköt palauttavat saman tosi-tuloksen (Kuva 25). Branch-yksikkö jatkaa ohjelmointivirtaa uuteen Branch-yksikköön.

Kuva 25 Chasing-kulkutilan visuaalisen ohjelmoinnin ensimmäinen vaihe.



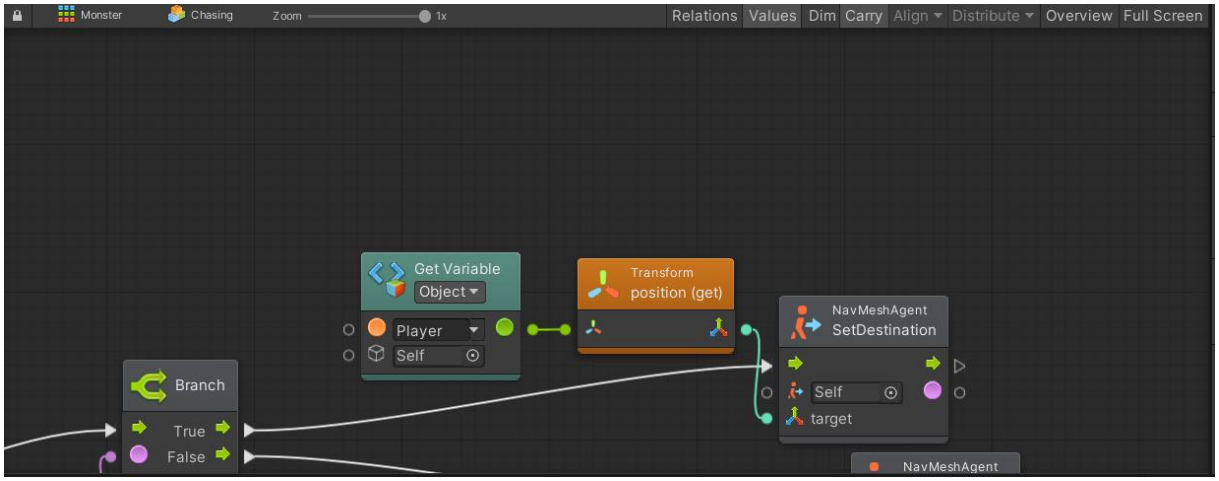
Toinen ohjelmointivaihe toimii samalla tavalla kuin ensimmäinen vaihe, mutta siinä tutkitaan Ally-peliobjektin etäisyyttä Monster-objektiin, kuin etäisyys on enemmän kuin PlayerNearDistance-muuttuja palautetaan tosi And-yksikköön. Peliobjektien etäisyyden mennessä alle PlayerNearDistanceen-arvon, muuttuu Less Or Equal -yksikön looginen tulos epätodeksi, jolloin ohjelmointivirta muuttuu kolmannessa ohjelmointivaiheen Branch-yksikössä. Siinä myös tutkitaan MonsterLVL tasoa, jonka ollessa pienempi AllyLVL-tasoon verrattuna, palauttaa And-yksikköön arvon tosi. (Kuva 26).

Kuva 26 Chasing-kulkutilan toinen vaihe.



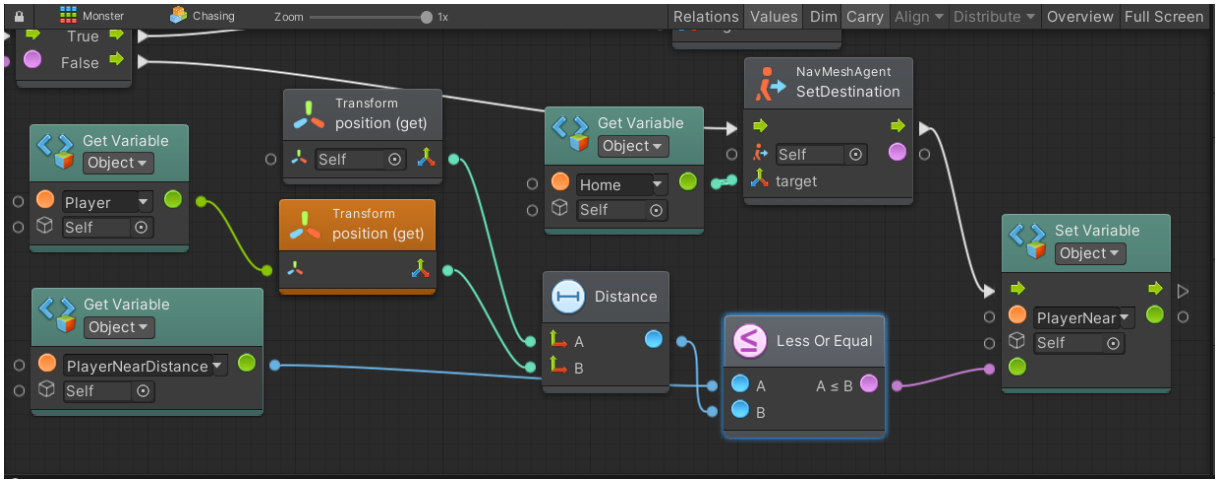
Kolmas ohjelmointivaihe Alkaa Branch-yksiköllä, johon toinen ohjelmointivaihe antaa loogisen arvonsa. Tämä Branch haarautuu kahteen eri ohjelmointivirtaosioon: jahtaa tai pakene. Näitä tekoälylogiikoita ohjataan aikaisemmin annetutilla tosi tai epätosi -arvojen avulla. Jos pelimaailman peliobjektien etäisyydet ja tasojen arvot on tosi, ohjelmointivirta suunnataan NavMeshAgent SetDestination -yksikölle, jonka target-tuloon haetaan Get Object Variable ja Transform position (get) -yksiköillä Player-peliobjektin pelimaailman sijainti. Tämän tiedon saadessaan NavMeshAgent SetDestination -yksikkö pystyy laskemaan lyhimmän reitin navigointiverkkoa hyödyntäen ja aloittamaan jahtaamisen (Kuva 27).

Kuva 27 Ohjelmointivirta jahtaa-toiminnolle.



Ennen kuin aloitetaan pakenemisen visuaalinen ohjelmointi, luodaan Object Variablesiin Home-muuttuja tyypiltään Vector3. Tähän Vector3 pystytään asettamaan pelimaailman sijainti X, Y ja Z-koordinaateilla, tähän muuttujaan luodaan Monster-peliobjektille pakenemiskohde. Branch-yksikön lähdöstä epätosi vedetään ohjelmointivirta NavMeshAgent SetDestination -yksikköön, jonka target-tuloon haetaan Get object Variable-yksiköllä Home-muuttujan tiedot, jolloin Monster ottaa kohteeksi pakopaikan koordinaatit. Lopuksi yhdistetään NavMeshAgent SetDestination -yksiköstä virtauksen Set Object Variable-yksikköön, johon muuttamalla PlayerNear-muuttujan arvon epätodeksi, pystytään hallitsemaan siirtymiä Idle ja Chasing -kulkutilojen välillä. PlayerNear-muuttujan logiikan muuttaminen epätodeksi alkaa hakemalla Get Object Variables -yksiköllä Player. Objektin Transform position (get) verrataan Monsterin omaan Transform positioniin hakemalla Distance-yksikkö, joka palauttaa kahden peliobjektin etäisyyden toisistaan. Vertaamalla Less Or Equal -yksikössä onko PlayerNearDistance vähemmän tai yhtä paljon kuin etäisyys Player ja Monster-peliobjektin saadaan tulokseksi epätosi (Kuva 28).

Kuva 28 Ohjelmointivirta pakene-toiminnolle.

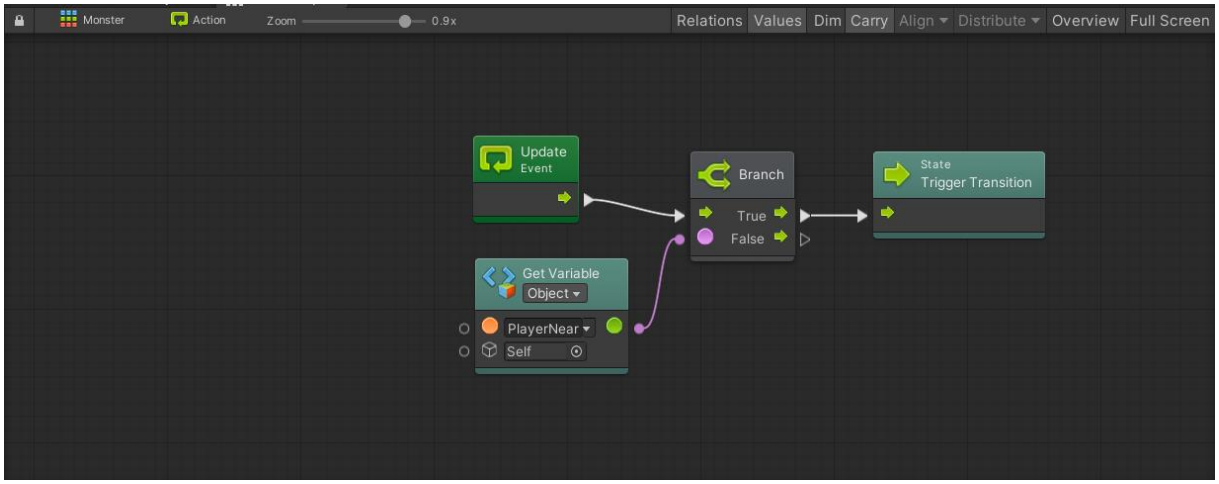


4.4.3 Idle ja Chasing -kulkutilojen väliset siirtymät

Siirtymillä hallitaan kulkutilojen ohjelmointivirran kulkua. Siirtymä luodaan painamalla hiiren oikeaa nappia halutun kulkutilan päällä ja valitsemalla valikosta Make Transition. Siirtymän kulkusuunta määritetään ensimmäisestä kulkutilasta toiseen. Luotu siirtymä voidaan nimetä Graph Inspector -ikkunassa ja tuplaklikkaamalla siitä pystymme avaamaan sen kulkukaavion. Kulkukaavio sisältää valmiiksi State Trigger Transition, jolla siirrytään ohjelmointivirtauksella haluttuun kulkutilaan.

Idle-kulkutilasta tehdään siirtymä Chasing-kulkutilaan. Avataan kulkukaavio, johon luodaan logiikka, jossa Player-peliobjektin aktivoidessa PlayerNear-muuttujan tehdään siirtymä Idle-tilasta Chasing-tilaan. Haemme Update Event -yksikön, siirtymää tarvitsee päivittää. Seuraavaksi vedetään Updatesta ohjelmointivirran Branch-yksikköön, johon haetaan Get Object Varialbe -yksiköllä PlayerNear-muuttuja. Branch yhdistetään lähdöstä tosi Trigger Transition -yksikköön, jolloin PlayerNear-muuttujan arvon muuttuessa todeksi tehdään siirtymä Chasing-kulkutilaan (Kuva 29). Seuraavaksi luodaan Chasing-kulkutilasta Idle-kulkutilaan siirtymä. Siirtymän luonti toteutetaan samalla tavalla kuin aikaisempi siirtymä, mutta Branch-yksikön epätosi lähdöstä tehdään ohjelmointivirta Trigger Transition- yksikköön.

Kuva 29 Siirtymän visuaalinen ohjelmointi.



5 Yhteenveto

Ensimmäisen tutkimuskysymykseni tarkoituksena oli selvittää, mitä Unity tuo Bolt Visual Scriptillä pelin kehittämiseen. Käyttäessä Boltia pelinkehittämisessä ei tarvitse aluksi tietää paljoa ohjelmointisyntakseja, koska Bolt tarjoaa asentamisvaiheessa mahdollisuuden käyttäjäystävällisellä nimeämisellä, joka kääntää monimutkaiset ohjelmointinimet helpommin ymmärrettävään muotoon. Visuaalisessa editorissa löytyy ennakoiva virheenkorjaus, joka ilmoittaa ohjelman kaatumisesta ennen kuin peli laitetaan päälle. Sillä pystyy myös seuraamaan ohjelman tietovirtaa ja muokkaamaan sitä reaaliajassa, vaikka pelitila olisi päällä.

Toinen tutkimuskysymykseni oli, kuinka Bolt auttaa aloittelevaa pelinkehittäjää tekoälyn luonnissa. Boltilla aloitteleva pelinkehittäjä pystyy helposti ottamaan käyttöön Unityn navigointiverkon peliobjektien älykkääseen liikkumiseen pelikentällä, jolloin säästyy aikaa muiden pelin osa-alueiden tekemiseen. Tekoälyn luominen ei-pelattaville hahmoille on Boltin tilakaaviolla yksinkertaista siinä hyödynnettävien siirtymätiloja takia ja näihin siirtymiin pystytään luomaan monipuolisesti pelitilannetta vastaavia logiikoita. Visuaalisesta editorista on helppo löytää tilannetta vaativan yksikkö etsimen (Fuzzy Finder) avulla, johon kirjoittamalla löytää sekunneissa kaikki tarvittavat metodit ja muuttujat.

Kolmas tutkimuskysymykseni oli, miten otetaan käyttöön tehokkaasti tekoäly kulkukaaviolla ja tilakaaviolla. Kulkukaavion logiikan oppiminen ja käyttöönotto on nopeaa, pelihahmo liikkuu tekoälyn avulla pelikentällä hiiren osoittamaan paikkaan, ja se pystytään luomaan jo muutamassa tunnissa netistä löytyvillä video- tai opetusmateriaaleilla ilman aikaisempaa kokemusta. Tilakaavion käyttöönotto on hitaampaa, sillä aloittelevan pelinkehittäjän on aluksi vaikea ymmärtää siirtymäsolmulogiikoiden mahdollisia ristiriitoja, joita syntyy peliobjektin hitaasta liikkumisesta tilakaavion nopeaan tiedonsiirtoon verrattuna. Näitä ristiriitoja on helppo muokata tutkimalla reaaliaikaisesti tilakaavion tiedonsiirtoa pelitilassa ja tekemällä ajastin-solmuelementillä tietoliikenteeseen taukoja, jolloin pelihahmo ehtii reagoida pelimaailman tilanteisiin oikealla tavalla. Tilakaaviolle löytyy Internetistä laaja skaala opetusmateriaalia erilaisten tekoälyjen luontiin, joita aloitteleva pelinkehittäjä voi hyödyntää omassa projektissaan.

Tehdessäni opinnäytetyötä opin monta erilaista toteutustapaa tekoälyn tekemiseen Unity-pelimoottorille. Näitä toteutustapoja testaamalla huomasin, että visuaalinen ohjelmointi soveltuu parhaiten aloittelevalle pelinkehittäjälle tekoälylogiikan oppimiseen. Visuaalisen ohjelmointieditorin avulla tiedon seuraaminen ja muokkaaminen ohjelmassa on luontevampaa loogisen lähestymistapansa ansiosta, jolloin ohjelmoinnin oppiminen lisääntyy. Opittuaan visuaalisen ohjelmoinnin aloitteleva pelikehittäjä pystyy hyödyntämään näitä taitoja työelämässä ja henkilökohtaisissa projekteissaan.

Kirjallisen raportin lisäksi tehtiin videosarja opinnäytetyön käytännön vaiheista mahdolliseen opetuskäyttöön.

Lähteet

Android Authority. (2020) What is Unity? Everything you need to know. Haettu 10.12.2020 osoitteesta https://www.androidauthority.com/what-is-unity-1131558/

Barrara, R. Kyaw, A. Naing, T. (2018) Implementing Pathfinding for AI agents with NavMesh in Unity, opetusohjelma. GameDev.net. Haettu 16.12.2020 osoitteesta https://www.gamedev.net/tutorials/programming/artificial-intelligence/implementing-pathfinding-for-ai-agents-with-navmesh-in-unity-r4903/

Chaudhari, V. (2017). Goal Oriented Action Planning. Blogijulkaisu 12.12.2017. Haettu 17.12.2020 osoitteesta https://medium.com/@vedantchaudhari/goal-oriented-action-planning-34035ed40d0b

Choudhary, A. (2020). Everything You Need To Know About Machine Learning In Unity 3D. *Analytics India Magazine.* Haettu 21.12.2020 osoitteesta https://analyticsindiamag.com/everything-you-need-to-know-about-machine-learning-in-unity-3d/

Colledanchise, M. & Ögren, P. (2018) *Behavior Trees in Robotics and AI: An Introduction*. Boca Raton: CRC Press. Haettu 14.12.2020 osoitteesta https://arxiv.org/pdf/1709.00084.pdf

Gamedesigning. (2020) AI in Unity: The Future of Gaming and Beyond. Haettu 10.12.2020 osoitteesta https://www.gamedesigning.org/learn/unity-ai/

GameDaily.biz. (2020) Unity Technologies acquires Bolt from Ludiq. Haettu 30.11.2020 osoitteesta https://gamedaily.biz/article/1729/unity-technologies-acquires-bolt-from-ludiq

Heller, M. (2019) What is machine learning? Intelligence derived from data. *InfoWorld*. Haettu 18.12.2020 osoitteesta https://www.infoworld.com/article/3214424/what-is-machine-learning-intelligence-derived-from-data.html

Ionos. (2020) Visual programming – the easy path to the digital world. Haettu osoitteesta https://www.ionos.com/digitalguide/websites/web-development/visual-programming/

Lou, H. (2017) AI in Video Games: Toward a More Intelligent Game. Blogijulkaisu 28.8.2017. Haettu 22.12.2020 osoitteesta http://sitn.hms.harvard.edu/flash/2017/ai-video-games-toward-intelligent-game/

Low-scope. (2019) Review: Bolt Visual Scripting. Haettu 30.11.2020 osoitteesta https://low-scope.com/unity-asset-review-bolt-visual-scripting-plugin/

Pangilinan, E. Lukas, S. Mohan, M. (2019). *Creating Augmented and Virtual Realities.* Sebastobol: O'Reilly Media. Haettu 15.12.2020 osoitteesta https://books.google.fi/books?id=XGmNDwAAQBAJ&pg=PT265&lpg=PT265&dq=deliberative+artificial+intelligence+to+unity&source=bl&ots=UTGzMsCehg&sig=ACfU3U1eXBkVD1eIr7Y3Xaih6qCFnbACFw&hl=fi&sa=X&ved=2ahUKEwik7uXk48_tAhVH2SoKHbMMCCw4ChDoATACegQIAhAC#v=onepage&q=deliberative%20artificial%20intelligence%20to%20unity&f=false

Perhanidis, C. (2020) Imitation Learning in Game Development. Blogijulkaisu 1.6.2020. Haettu 21.12.2020 osoitteesta https://medium.com/@pmchrist/imitation-learning-in-game-development-759d70998f69

The conversation. (2016) Understanding the four types of AI, from reactive robots to self-aware beings. Haettu 10.12.2020 osoitteesta https://theconversation.com/understanding-the-four-types-of-ai-from-reactive-robots-to-self-aware-beings-67616

Tsung, D. (2016) Don't Re-invent Finite State Machine: How to Repurpose Unity's Animator Blogijulkaisu 31.8.2016. Haettu 14.12.2020 ositteesta https://medium.com/the-unity-developers-handbook/dont-re-invent-finite-state-machines-how-to-repurpose-unity-s-animator-7c6c421e5785

Unity. (2020) Graphs, Machines and Macros. Manuaali. Haettu 1.12.2020 osoitteesta https://docs.unity3d.com/bolt/1.4/manual/bolt-graphs-machines-macros.html

Unity. (2020) Flow States and Super States. Manuaali. Haettu 1.12.2020 osoitteesta https://docs.unity3d.com/bolt/1.4/manual/bolt-states.html

Unity. (2020) Coding in C# in Unity for beginners. Manuaali. Haettu 10.12.2020 osoitteesta https://unity3d.com/learning-c-sharp-in-unity-for-beginners

Unity documentation. (2020) Animation State Machines. Manuaali. Haettu 14.12.2020 osoitteesta https://docs.unity3d.com/Manual/AnimationStateMachines.html

Unity documentation. (2020) Building a NavMesh. Manuaali. Haettu 16.12.2020 osoitteesta https://docs.unity3d.com/Manual/nav-BuildingNavMesh.html

Unity documentation. (2020) State Machine Basics. Manuaali. Haettu 11.12.2020 osoitteesta https://docs.unity3d.com/2019.3/Documentation/Manual/StateMachineBasics.html

Unity. (2020) Verkkosivusto. Haettu 24.11.2020 osoitteesta https://unity.com/

Steiger, A. (2020) How Unity is Democratising AI: Unity ML-Agents is a Game Changer. Blogijulkaisu 12.3.2020. Haettu 21.12.2020 osoitteesta https://unitydevelopers.co.uk/how-unity-is-democratising-ai-unity-ml-agents-is-a-game-changer/

Syahputra, M. Arippa, A. Rahmat, R, Andayani, U. (2019). Journal of Physics: Conference Series. teoksessa M. Syahputra (toim.) *Historical Theme Game Using Finite State Machine for Actor Behaviour.* Englanti: IOP Publishing, ss. 3. https://www.researchgate.net/publication/334630752_Historical_Theme_Game_Using_Finite_State_Machine_for_Actor_Behaviour

VionixStudio. (2020) Bolt vs Playmaker which is the best in Unity. Haettu 30.11.2020 ositteesta https://vionixstudio.com/2020/05/01/bolt-vs-playmaker-which-is-the-best-in-unity/

Liite 1: Aineistonhallintasuunnitelma

Opinnäytetyötä varten luotiin OneDrive-pilvipalveluun varakopio opinnäytetyöstä mahdollisen tietokoneen rikkoutumista varalta. Opinnäytetyössä käytettävää lähdeaineistoa haettiin internetistä löytyvistä tieteellisistä julkaisuista, lehtien artikkeleista ja Unity-pelimoottorille löytyvistä manuaaleista. Aineistoa säilytettiin Chrome-nettiselaimen kirjanmerkeissä, josta aineistoa hallinnoitiin erilaisilla aihealueita kuvastavilla kansioilla. Opinnäytetyössä luotiin visuaalisen ohjelmoinnin oppimateriaali, jota säilytettiin tietokoneen C- ja D-asemilla, ja josta tehtiin varmuuskopiointi OneDriveen.