Sami Lahtinen

Hyötypohjaisen tekoälyn toteuttaminen Unity-pelimoottorissa

Tradenomi
Tietojenkäsittely
Syksy 2020

KAMK • University of Applied Sciences

Tiivistelmä

Tekijä(t): Lahtinen Sami

Työn nimi: Hyötypohjaisen tekoälyn toteuttaminen Unity-pelimoottorissa

Tutkintonimike: Tradenomi (AMK), tietojenkäsittely

Asiasanat: tekoäly, pelikehityksen työkalut, Unity, C#, peliohjelmointi

Tämä opinnäytetyön tavoitteena oli ensin selvittää, miten hyötypohjainen tekoäly toimii yleisellä tasolla, ja sen jälkeen toteuttaa kyseistä tekoälyn toimintamallia hyödyntävä kokonaisuus Unity-pelimoottorissa. Käytännön vaiheen tavoite oli luoda tekoäly sekä editorityökalu, jonka avulla voi visualisoida ja muokata päätöksentekoa ohjaavaa tietoa. Opinnäytetyö toteutettiin henkilökohtaisten peliprojektien kautta Unity-moottorin 2019.3-sarjan versioilla.

Hyötypohjainen tekoäly valitsee kaikkien mahdollisten toimintojen joukosta sen, joka sopii senhetkiseen tilanteeseen parhaiten. Päätöksenteko toteutetaan numeroarvoilla suunnittelijan määrittämien laskutoimitusten perusteella. Hyötypohjainen tekoäly voi tuottaa erittäin mukautuvaa ja monitahoista käytöstä ja se sopii jatkuvaan pelikehitysprosessiin muokkautuvuutensa ansiosta.

Työn aikana luotiin tekoälyn koodiympäristö ja visuaalinen editorityökalu, jotka toimivat työn alussa asetettujen tavoitteiden mukaisesti. Uusien toimintojen ja arvoja palauttavien luokkien ohjelmointi on yksinkertaista abstraktien pohjaluokkien avulla. Päätöksenteon muokkaaminen onnistuu helposti, mutta testaus on vielä liian monimutkaista. Työkalun jatkokehityksen tärkein osa-alue on visuaalisen viankorjausnäkymän rakentaminen.

KAMK • University of Applied Sciences

Abstract

Author(s): Lahtinen Sami

Title of the Publication: Implementation of a utility-based AI in Unity game-engine

Degree Title: Bachelor of Business Administration, Business Information Technology

Keywords: artificial intelligence, game design tools, Unity, serialization, C#, game programming

The purpose of this thesis was to study the general principles of utility-based AI and implement it to Unity game engine. A functional AI framework with a visual editor tool was built during the practical portion of this thesis. It was built using the Unity-engine version 2019.3 and a personal game project was used for testing.

Utility AI calculates and chooses the best possible action for the current situation from a pool of all possible actions. Decision-making uses simple math with floating point numbers, which are used to represent the current game state from the perspective of an AI unit. Utility AI can produce dynamic and complex behavior, which looks believable from the player's perspective. It is also suitable for a continuous game development process thanks to its editability.

The AI framework and editor tool both achieved their goals set at the start of the practical portion. Creating code functionality for new AI behavior is easy thanks to abstract base classes. The editor tool is simple to use, but the resulting behavior is currently difficult to test. Building a debugging tool would be a high priority during further development.

Sisällys

Sanasto

AI	Lyhenne englannin kielen termistä ”artificial intelligence”, suomeksi tekoäly
Delta time	Numeerinen arvo, joka ilmaisee kulunutta aikaa edellisestä sovelluksen silmukan suorituksesta
Hyötyarvo	Numeerinen arvo, joka ilmaisee mahdollisen toiminnon tärkeyttä arvon laskentahetkellä
Instanssi	Yksittäinen käsiteltävissä oleva objekti
Käytöspuu	Kerroksiin asetetuista solmuista koostuva malli, joka ohjaa ohjelmiston käyttäytymistä suorituksen aikana
Normalisointi	Arvojen muuttaminen yhteisesti verrattavissa olevaan mittakaavaan
Rajapinta	Ohjelmistokokonaisuuden näkyvissä (public) olevat funktiot ja muuttujat, joiden avulla ohjelmistoa käytetään koodiympäristössä
Serialisointi	Tiedon tai objektin kääntäminen tallennettavaan muotoon
Solmu	Tietoverkoston yksittäinen piste, yhdistetty linkkien avulla toisiin solmuihin
Tilakone	Tiloista ja niiden välisistä yhteyksistä muodostuva suljettu kokonaisuus. Tilakoneen avulla voidaan määrittää, miten ohjelmisto käyttäytyy erilaisissa suoritustilanteissa.

1 Johdanto

Työkalun kehitys ja testaus alkoivat henkilökohtaisen peliprojektin kautta. Kyseinen projekti on kolmannen persoonan toimintapeli, jossa on mukana hiippailupelien mekaniikkoja. Halusin siinä luoda pelaajalle tietynlaisen kuvan pelissä tavatuista vihollisista. Tavoitteeni oli, että hahmot tuntuvat tietoisilta ympäristöstään ja yksilöt kykenevät kommunikoimaan toistensa kanssa ongelmatilanteissa. Hyötypohjainen päätöksenteko oli itselleni opinnäytetyöni alussa uusi käsite, mutta jo lyhyen tutkimustyön jälkeen sen tuomat mahdollisuudet ja käytännön esimerkit vaikuttivat kiehtovilta.

Vertailen työssä aluksi hyötypohjaista päätöksentekoa myös muihin yleisimpiin tekoälymalleihin ja pyrin selvittämään, missä tilanteissa hyötypohjainen päätöksenteko on järkevä ratkaisu. Käytännön tavoitteena oli suunnitella ja kehittää systeemi pelitekoälylle, joka hyödyntää hyötypohjaista päätöksentekoa. Käyn yksityiskohtaisesti läpi, miten tekoäly toimii pelissä ohjelmoinnin näkökulmasta ja mitkä ovat sen toimintaperiaatteet. Lisäksi esittelen tekoälyn muokkausta varten rakennetun visuaalisen editorityökalun.

Kaikki työssä käytetyt kuva- ja koodiesimerkit perustuvat Unity-pelimoottorin 2019.3-version julkaisuihin. Kaikki ohjelmointi toteutettiin C#-ohjelmointikielellä.

2 Pelitekoälyn toteutuksen yleisimmät mallit

Useimmat pelit hyödyntävät tekoälyä jollain tasolla. Tekoälyä käytetään yleisimmin yksittäisten pelihahmojen käytöksen ohjaamiseen, mutta sitä voidaan soveltaa myös hyvin erikoisissa tilanteissa. Mukautuvan päätöksenteon avulla voidaan ohjata pelihahmon ja pelaajan välisen keskustelun kulkua tai hallita taustamusiikin miksausta ja rakennetta pelitilanteen perusteella. Pelitekoälyn tärkein tehtävä yleisellä tasolla on tarjota pelaajalle mahdollisimman kiintoisa kokemus ja olla osana pelin tavoitteleman tunnetilan herättämisessä. (Kirby 2020, 1-3.) Erilaiset pelit vaativat erilaisia tekoälyjä, eikä sama päätöksenteon malli sovellu jokaiseen tilanteeseen.

2.1 Kovakoodattu tekoäly

Hyvin yksinkertaisissa tilanteissa pelihahmon käytös voidaan kirjoittaa suoraan hahmon lähdekoodiin. Kovakoodatun tekoälyn koodia on suoraviivaista luoda, halpaa suorittaa ja usein helppoa testata. Yksinkertaiset objektit, kuten peliympäristöissä sijaitsevat hissit tai tunnelmaa luovat taustaeläinhahmot, eivät vaadi juuri lainkaan päätöksentekoa, joten kovakoodattu käytös voi hyvinkin olla järkevin ratkaisu.

Kaikki mahdolliset tilat ja toiminnot määritetään suoraan koodiin, jonka ansiosta sen ylläpitäminen ja muokkaaminen muuttuvat hankalammaksi, kun uutta käytöstä lisätään. Kovakoodattu tekoäly voi myös tuottaa hyvin kankeaa käytöstä, tai jopa rikkoutua kokonaan, jos pelaajan käyttäytyminen ei sovi mihinkään koodissa määrättyyn sääntöön. Kovakoodattu tekoäly sopii siis hyvin tilanteisiin, joissa tekoälyn haluttu käytös on täysin ennalta määrättyä ja kaikki mahdolliset tilanteet otetaan huomioon. (Kirby 2010, 19-22.)

2.2 Tilakoneet

Tilakone (engl. finite-state machine) on tiedon käsittelyssä käytetty malli, joka muodostuu tilojen ja niiden välisten yhteyksien kautta. Tilakoneessa vain yksi tila voi olla aktiivisena kerrallaan. Tilasta voidaan siirtyä toiseen vain, jos niiden välille on luotu yhteys ja sille asetetut säännöt toteutuvat. (Millington 2019, 314-316.)

Tilakoneita voidaan käyttää peleissä moniin tarkoituksiin, joista yksi tärkeimmistä on tekoäly. Tekoälyn käytös voidaan pilkkoa yksittäisiin toimintoihin, joita tilakoneessa voidaan kuvata tiloina. Tekoäly voi vaihtaa nykyistä toimintoa eli tilakoneen tilaa yhteyksien kautta. Yhteyksille asetetaan erilaisia sääntöjä, joita vastaan testataan pelin senhetkistä tilaa aina päätöksenteon hetkellä. (Bourg & Seemann 2004, 229.) Tällaisia sääntöjä voivat olla esimerkiksi ”etäisyys pelaajaobjektiin alle 5 yksikköä” tai ”liikkumiskohde saavutettu”.

Tilakoneiden suurin vahvuus on niiden yksinkertaisuus. Tilakoneen rakentaminen ja muokkaaminen on helppoa ja nopeaa. Uusien tilojen lisääminen jälkeenpäin on mahdollista, mutta yhteyksien ylläpitäminen voi koitua raskaaksi, jos tilojen määrä on hyvin suuri (Colledanchise & Ögren 2018, 5, 24-25). Monimutkaisissa tilanteissa tilakoneen voi hajottaa useisiin kerroksiin. Tällaisissa tapauksissa usein ylimmän kerroksen tilat ovat itse tilakoneita, joiden alle on rakennettu käytöksen määrittävät tilat ja yhteydet. Kerrostetussa tilakoneessa kerrosten väliset yhteydet voivat silti viedä paljon vaivaa, varsinkin jos kerroksen tilojen joukossa on erityistapauksia, joiden kohdalla siirtymät määräytyvät eri sääntöjen mukaan. (Dawe, Gargolinski, Dicken, Humphreys & Mark 2013, 48-52.)

Pelaajan näkökulmasta tilakonetta hyödyntävän tekoälyn käytöstä on myöskin usein helppoa lukea ja jopa ennakoida. Pelaaja voi oppia tunnistamaan tekoälyä ohjaavia sääntöjä, jos tekoäly tuottaa toistuvasti samankaltaista käytöstä tietynlaisissa tilanteissa. Tekoälyn hahmottaminen pelin aikana on yleensä suoraviivainen ja luonnollinen prosessi, jos tilakoneen sisältämien tilojen kokonaismäärä on pieni ja pelaajalle annetaan mahdollisuuksia seurata tekoälyä erilaisessa tilanteessa. Tämä voi olla vahvuus tai heikkous riippuen pelin tyylistä ja tekoälyn tarkoituksesta. Useissa toimintapeleissä tekoälyn tulee toimia selkeiden sääntöjen mukaan ja paljastaa pelaajalle tarpeeksi tietoa miksi tietty käytös valittiin. Sen tyyppisissä peleissä pelaajan tavoitteena on havaita sääntöjä ja ennakoida tekoälyn käytöstä, jotta se voidaan päihittää.

2.3 Käytöspuut

Käytöspuu on solmuista koostuva verkosto, joka laajenee ydinsolmusta alaspäin. Yksi solmu voi esittää tekoälyn toimintoa pelissä, hallita alasolmujen suoritusta tai prosessoida solmujen välistä tietoa. Yleensä toiminnot ovat haaransa alimpia solmuja, eikä niillä ole alasolmuja. Solmujen tyypit vaihtelevat tilannekohtaisesti, mutta yleistä on, että puuta prosessoidaan pystysuunnassa ja

solmu vastaa siihen liitetyistä alapuolisista solmuista. Solmuille asetetaan myös pelin tilaan perustuvia sääntöjä, joiden avulla hallitaan solmujen suorittamista. (Colledanchise & Ögren 2018, 6-9.)

Solmu, jolla on alasolmuja, voi toteuttaa niiden suorituksen suunnittelijan määrittämien sääntöjen mukaan. Alasolmut voidaan suorittaa tietyssä järjestyksessä, satunnaisesti tai yhtä aikaa. Alapuolinen solmu palauttaa suorituksen aikana tilanneraportin yläpuoliselle solmulle. Tilanneraportti sisältää muuttujia, jotka kuvaavat suorituksen tilaa sillä hetkellä. Yleensä mahdollisia tiloja ovat onnistuminen, epäonnistuminen ja keskeneräinen suoritus. Tilakone suorittaa keskeneräisiä solmuja usean käytöspuun päivityskerran ajan, kunnes ne onnistuvat tai epäonnistuvat. (Champandard & Dunstan 2013, 76-82.)

Käytöspuuta on helppo lukea, koska sen rakenne vastaa koodin suoritusjärjestystä, joka kulkeutuu ylhäältä alaspäin 2D-tasolla. Käytöspuu on rakenteensa ansiosta hajotettu erillisiin koodin palasiin, jotka toimivat samalla tavalla kaikissa asiayhteyksissä. Tämän ansiosta samoilla solmuilla on mahdollista luoda hyvin erilaisia käytösmalleja asettamalla ne eri järjestykseen. Palasia voidaan myös käyttää uudelleen useissa eri tekoälyissä ja jopa toisissa projekteissa. Käytöspuut ovat reaktiivisia pelin senhetkiseen tilaan ja sopeutuvat jatkuvaan kehitysprosessiin laajennettavuuden ja muokkautuvuutensa ansiosta. (Colledanchise & Ögren 2018, 38.)

Käytöspuun isoimmat vahvuudet ovat sen yksinkertaisuus ja muokattavuus (Dawe ym. 2013, 53). Niiden tehokas hyödyntäminen vaatii kuitenkin jonkinlaisen työkalun, jos tekoäly ei toteuteta pelkästään koodiympäristössä. Uuden työkalun kehittäminen vaatii totta kai aikaa, mutta internetin kautta löytää useita käytöspuu-työkaluja eri kehitysalustoille, jotka ovat vapaasti käytettävissä jopa kaupallisissa projekteissa. Käytöspuu voi myös olla raskas suorittaa, jos siinä käytetyt säännöt ovat monimutkaisia tai suoritettavien solmujen kokonaismäärä on hyvin korkea.

Pelissä voi olla tilanteita, jolloin tekoälyn toimintojen tärkeys muuttuu. Käytöspuussa toimintojen tärkeysjärjestys määräytyy suoraan suunnittelijan luoman rakenteen mukaan, ja kaikki toiminnon suorittamiseen vaikuttavat muuttujat täytyy ottaa huomioon käytöspuuta rakennettaessa. (Millington 2019, 374.) Esimerkiksi jalkapallopelissä hahmon tekoälyllä voi olla toiminto, jonka avulla se yrittää potkaista pallon maaliin. Tälle toiminnolle täytyy määritellä käytöspuussa ehdot, jolloin toiminto suoritetaan. Hahmolla täytyy olla pallo hallussa ja hänen tulee olla tarpeeksi lähellä vastapuolen maalia. Toimintoa valitessa täytyy myös ottaa huomioon hahmon ja maalin välissä olevat vastapuolen puolustajat ja mahdolliset tiimitoverit, joilla on parempi tilaisuus saada pallo maaliin asti. Kaikki nämä muut häilyvät muuttujat vaikuttavat kyseisen toiminnon valitsemiseen.

Niiden purkaminen kyllä/ei -tasolle käytöspuun päätöksenteossa voi olla tässä tapauksessa haastava tehtävä.

2.4 Tavoitepohjainen tekoäly

Tavoitepohjaisessa päätöksenteossa yksilö määrittää itselleen sarjan toimintoja, joiden avulla se saavuttaa tavoitteensa. Lopullinen käytös rakentuu yksittäisistä toiminnoista, joille voidaan määrittää esivaatimuksia tai seuraamuksia. Esivaatimukset voivat vaatia yksilöä suorittamaan toisen toiminnon ensin, jonka seurauksena aikaisemmin lukittu toiminto voidaan suorittaa. Jokaiselle toiminnolle määritetään myös numeroarvo, joka kertoo, kuinka vaivalloista toiminnon suorittaminen on. (Millington 2019, 406-408.)

Tekoäly hyödyntää reitinhakualgoritmia toimintosarjan laskennassa. Haku aloitetaan lopullisesta tavoitteesta, ja siinä käytetään toimintojen vaatimuksia, seuraamuksia ja vaiva-arvoja. Lopullinen käytös on sarja toimintoja, joka täyttää tekoälyn tavoitteen pienimmällä mahdollisella yhteenlasketulla vaiva-arvolla. Päätöksenteossa käytetty hakualgoritmi voi olla raskas suorittaa, jos mahdollisten toimintojen määrä on korkea tai algoritmia ei ole optimoitu tarpeeksi. (Millington 2019, 413, 421.)

Tavoitepohjainen tekoäly välittää vain siitä, kuinka tehokkaasti se tavoittaa päämääräänsä. Tämän takia tekoälyä on hyvin vaikea ohjata suorittamaan ohjattuja tai tilannekohtaisia toimenpiteitä. Perinteisesti tavoitepohjainen tekoäly valitsee aina itselleen tehokkaimman sarjan toimintoja, joten niille asetettuja arvoja ja ehtoja täytyy hienosäätää ja testata suunnitteluvaiheen aikana. Mitä tarkempaa ja ohjatumpaa käytöstä tekoälyltä vaaditaan, sitä enemmän aikaa kuluu niin ehtojen ja seurausten määrittämiseen kuin tekoälyn testaamiseen peliympäristössä. Monioosaisten käytössarjojen testaaminen on monimutkainen prosessi, joka vaatii runsaasti aikaa ja osaamista, tai älykkäät kehitystyökalut. Jos tavoitepohjainen tekoäly sopii projektiin, sen avulla on mahdollista luoda tehokkaita, mukautuvia ja monimuotoisia tekoälyjä. (Chotrani & Jin 2018, 9.)

3 Hyötypohjainen tekoäly

Hyötypohjaisen tekoälyn yleinen toimintaperiaate on seuraava: jokaiselle yksilön mahdolliselle toiminnolle lasketaan ”hyötyarvo” ja jälkeenpäin suoritetaan toiminto, jolla on korkein arvo (Graham 2013, 113). Hyötyarvon laskenta tapahtuu reaaliaikaisesti pelin aikana erilaisten muuttujien perusteella. Muuttujien tulokset muutetaan liukuviksi numeroarvoiksi, jotta eri toimintojen hyötyarvoja voidaan suoraan verrata keskenään. Iso oppimiskynnys hyötypohjaisen tekoälyn suunnittelemisessa on kehittää tapoja kuvata pelin tilaa numeroarvoina.

Tämän toimintamallin avulla tekoäly kykenee mukautumaan senhetkiseen tilanteeseen helpommin ja pelaajan näkökulmasta käytös vaikuttaa usein ”älykkäämmältä”. Päätöksenteossa kaikki mahdolliset toiminnot otetaan huomioon laskemalla niiden hyötyarvot. Tekoäly voi siten päätöksenteon tuloksesta siirtyä mistä tahansa käytöksestä toiseen. Hyötypohjaisen tekoälyn käytöstä on vaikeampaa ennakoida kuin esimerkiksi tilakoneen kautta toimivan tekoälyn käytöstä. Suunnittelijan tehtävä on rakentaa tasapaino hyötypohjaisen tekoälyn käytökseen, jolloin sen toiminta on tekoälyn mukaan hyvä valinta, mutta myöskin pelaajan näkökulmasta reilu tai mielekäs lopputulos. (Dawe ym. 2013, 54.)

3.1 Kuinka se toimii

Hyötyarvo on pelin aikana laskettu numeroarvo, jonka tehtävänä on ilmaista mahdollisen toiminnon tärkeyttä päätöksenteon hetkellä. Hyötyarvon laskemiseen voidaan käyttää kaikenlaista muuttuvaa tietoa, jonka tekoäly muuttaa numeroarvoiksi päätöksenteon aikana. Tällaisia tiedonlähteitä ovat muun muassa yksilön elämäpisteet, etäisyydet toisiin objekteihin tai ammusten määrä. Tekoälylle voidaan mallintaa myös aisteja, joiden tarjoama tieto muutetaan päätöksenteossa käytettäviksi numeroarvoiksi. (Graham 2013, 113-115.)

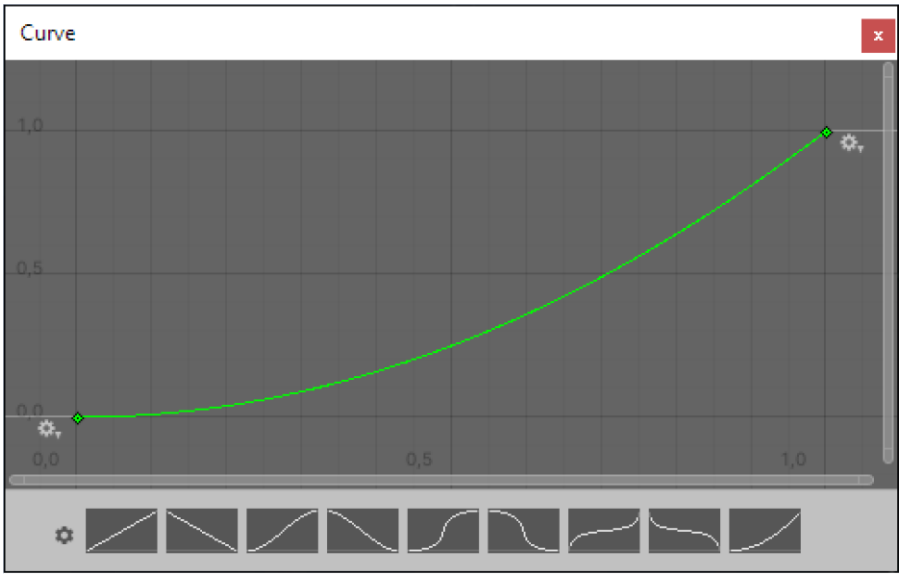
Hyötyarvon määrittämisessä käytetään erilaisia laskutoimituksia, joilla manipuloidaan tiedonlähteistä haettuja arvoja. Kaikki laskutoimituksissa käytetyt numeroarvot eivät ole peräisin tiedonlähteistä, vaan tekoälyn suunnittelija asettaa ne kehitysvaiheessa. (Graham, 2013, s.116) Päätöksenteossa käytetyimmät laskutoimitukset ovat matemaattisia perustoimenpiteitä, kuten kerto-, jako- ja yhteenlasku. Digitaalisissa piirilevyissä käytetään loogisia portteja, jotka suorittavat yksinkertaisia vertailuja sisääntulojen perusteella. (Hewes 2020.) Hyötyarvon laskennassa voidaan myös hyödyntää yksinkertaista logiikkaa. Looginen sääntö nimeltä ”Ja” vaatii, että kaikki syötetyt

arvot ovat suurempia kuin 0. Sääntö ”Tai” vaatii, että vähintään yksi syötetty arvo on suurempi kuin 0. Tekoälyn laskennassa loogiset säännöt vastaanottavat float-arvoja ja syöttävät ulos joko float-arvon 0 tai 1. Loogisten sääntöjen tuloksia voidaan käyttää muiden arvojen kertoimina, jos halutaan tietyissä tilanteissa lisätä tai poistaa tiedonlähteitä tai mahdollisia valintoja hyötyarvon laskennassa.

Yksi tärkeimmistä laskutoimituksista hyötypohjaisessa päätöksenteossa on normalisointi. Normalisointi tässä asiayhteydessä tarkoittaa, että numeroarvo esitetään muodossa arvo / maksimiarvo. Luvun normalisointi palauttaa liukuvan prosenttiarvon väliltä 0-1, joka kertoo, kuinka lähellä arvo on sen maksimiarvoa. Normalisoinnin avulla lähteistä haettuja arvoja tai lopullisia hyötyarvoja voidaan aina vertailla keskenään. (Graham 2013, 114.)

Normalisoituja arvoja on myös helppoa sijoittaa erilaisille kuvaajille. Kuvaaja on 2D-tasolle piirretty käyrä, joka visualisoi matemaattisen lauseen tuloksia. Se vastaanottaa numeroarvon, jota käytetään lauseen tuloksen laskennassa. Laskutoimituksen tulos palautetaan ja sitä voidaan käyttää hyötyarvon laskennassa. Muokattavien kuvaajien avulla voidaan visualisoida, kuinka hyötyarvo määrittyy tietyllä arvojoukolla. (Graham 2013, 119.) Kuvaajien matemaattisia lauseita voidaan luoda suoraan koodista, mutta editorityökalut kuvaajien luomista ja muokkaamista varten sallivat syvemmän hallinnan hyötyarvon laskennassa.

Unity-moottorin AnimationCurve-luokka on tehokas tapa luoda muokattavia kuvaajia. Käyrän pisteitä voidaan muokata pienessä editor-ikkunassa, joka esitetään kuvankaappauksessa. (Kuva 1.) Uusia pisteitä voidaan lisätä, mutta myös niiden paikkaa ja kierrettä voidaan muokata. Uuden käyrän kahden pisteen sivusuuntaisen akselin oletusarvot ovat 0 ja 1, jotka sopeutuvat täydellisesti normalisoitujen arvojen muokkaamiseen. AnimationCurve-luokan Evaluate-funktio ottaa vastaan float-arvon, joka kuvaa kuvan sivusuuntaista akselia tai x-arvoa. Funktio palauttaa float-arvon, joka vastaa käyrän korkeusarvoa x-arvon määrittämässä kohdassa. AnimationCurve-luokka on erittäin tehokas työkalu, jota voidaan myös soveltaa tekoälyn ulkopuolella monessa eri tilanteessa. (Unity Technologies 2020.)



Kuva 1. Kuvankaappaus AnimationCurve-luokan muokkausikkunasta Unity-editorissa

Hyötypohjaista päätöksentekoa on sovellettu peleissä jo yli 20 vuotta. Sen vahvuuksia on sovellettu erityisesti strategia-, rooli- ja toimintapeleissä. The Sims -pelisarja on yksi isoimmista ja ensimmäisistä pelisarjoista, jonka kehittäjät ovat esitelleet, miten heidän pelissään hyödynnetään hyötypohjaista päätöksentekoa. Sarjan pelien tekoäly simuloi ihmisen perustarpeita ja käyttää niitä yksilöiden toiminnan ohjaamisessa. (Zubek 2010.)

3.2 Vahvuudet

Hyötypohjaisen päätöksenteon avulla tekoäly sopeutuu tilanteeseen ja valitsee parhaimman toimintatavan. Jokainen toiminta määritetään reaaliajassa pelitilanteesta haetun tiedon perusteella, joten yksittäiset tekoälyn yksilöt pystyvät nopeasti reagoimaan samaan tilanteeseen hyvinkin erilaisella käytöksellä. Tämä mallin avulla voidaan kehittää päätöksenteoltaan erittäin tehokkaita ja pelaajan näkökulmasta älykkäältä tuntuvia tekoälyjä. (Dawe ym. 2013, 55.)

Jokainen tekoälyn toiminta-avaruuteen kuuluva toiminto vaatii oman laskutoimituksen, joka palauttaa sille kuuluvan hyötyarvon. Toiminnot eivät ole millään tavalla riippuvaisia toisistaan, joten niiden lisääminen ja poistaminen on nopeaa ja jotakuinkin turvallista. Käytösten muokkaaminen vaikuttaa toisiin käytöksiin vain hyötyarvon laskennan aikana, jolloin tärkeimmäksi osoitettu käytös valitaan kaikkien joukosta. Hyötypohjainen tekoäly voikin ongelmitta siirtyä mistä tahansa toi-

minnosta toiseen. Hyötypohjainen tekoäly sopii muokkautuvuutensa ansiosta hyvin jatkuvaan pelinkehitysprosessiin ja skaalautuu hyvin tilanteissa, joissa tekoälyllä on suuri määrä mahdollisia käytöksiä. (Dawe ym. 2013, 55.)

Käytöstä voidaan muokata muuttamalla hyötyarvojen laskutoimituksia halutulla tavalla. Useimmissa tapauksissa kaikki hyötyarvon laskenta toteutetaan float-arvoilla, joten se on laskennallisesti kevyt toteuttaa (Mark 2009, 36). Kaikki tekoälyn tekemät päätökset tehdään pelkkien numeroarvojen eli tiedon perusteella, joten se ei ole riippuvainen pelimoottorin toiminnoista. Ainoat linkit pelimoottoriin ovat tiedonlähteiden arvojen välittäminen pelistä tekoälylle ja toimintojen mahdolliset implementaatiot. Tämän ansiosta hyötypohjaisen tekoälyn pystyy mahdollisesti rakentamaan itsenäisenä koodin kokonaisuutena, joka on hyödynnettävissä eri ympäristöissä.

3.3 Heikkoudet

Hyötypohjainen päätöksenteko on tehokas ja mukautuva malli tekoälyn luomiselle, mutta sen hyödyntäminen tuo mukanaan mahdollisia vastoinkäymisiä. Hyötypohjaisen tekoälyn muokkaaminen ja laajentaminen on yksinkertaista, mutta kehityksessä käytettyjen työkalujen suunnittelu, toteutus ja testaaminen vie huomattavan osan koko tekoälyn kehitysprosessista. Pelkästään tämän takia hyötypohjainen tekoäly ei välttämättä ole paras ratkaisu projekteihin.

Jos projektin tekoäly voidaan toteuttaa yksinkertaisemmalla mallilla ilman, että sen kokonaisvaltainen laatu kärsii, on yksinkertainen versio aina kannattavampi rakentaa ensin. Tilakone tai käytöspuu on yleensä tarpeeksi tehokas työkalu lähes jokaiseen tarkoitukseen, joten on tärkeää tutustua mahdollisiin vaihtoehtoihin. Hyötypohjaisesta päätöksenteosta on saatavilla huomattavasti vähemmän opiskeluaineistoa ja esimerkkejä kuin yleisemmistä tekoälyn malleista, joten sen suunnittelu ja toteutus voi vaatia runsaasti omaperäisyyttä ja luovia ratkaisuja.

Hyötypohjainen tekoäly voi myös tuottaa käytöstä, jota on vaikeampi ennakoida. Jotta tekoälyn käytös saadaan mahdollisimman lähelle suunnittelijan tavoitetta, tarvitaan paljon pelitestausta ja hyötyarvojen laskutoimitusten säätelyä. Tämä prosessi voi viedä yllättävän paljon aikaa, johon kehittäjien on hyvä varautua. (Dawe ym. 2013, 55.)

4 Tekoälyn päätöksenteon toteuttava koodi

Työn aikana luodun AI-työkalun koodi koostuu kahdesta erillisestä osa-alueesta. Tekoälyn ajonaikainen koodi suorittaa kaiken laskennan ja päätöksenteon. Käytöksen luomista ja muokkaamista varten luotu koodi suoritetaan pelkästään Unity-editorissa eikä sitä lisätä pelin koottuun versioon. Pelinaikainen koodi valmistui jo hyvin varhaisessa vaiheessa työn aikana. Tämä luku keskittyy sen eri luokkiin ja niiden toimintoihin.

4.1 Luokkien yleiskuva ja hierarkia

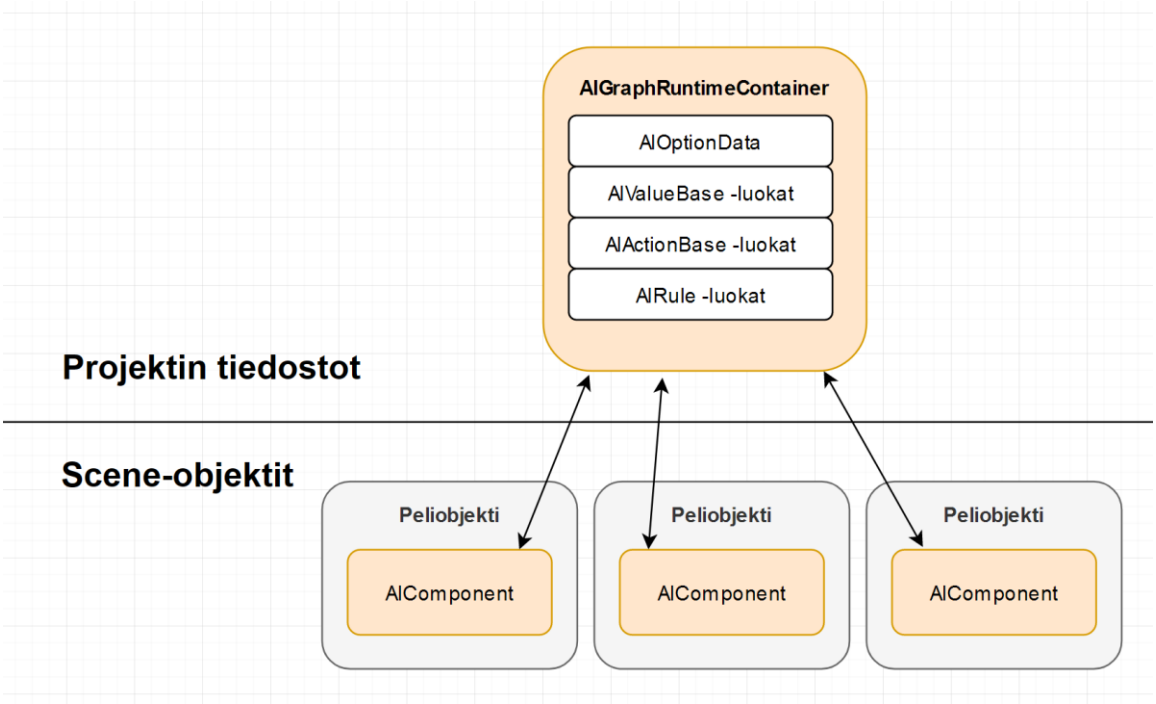
Pelinaikainen koodi koostuu seuraavassa taulukossa esitetyistä luokista. (Taulukko 1.)

Luokan nimi	Lyhyt kuvaus
AIComponent	Vastaa yksittäisen tekoälyn suorituksesta pelin aikana.
AIGraphRuntimeContainer	Levylle tallennettu objekti, joka sisältää tekoälyn toiminta-avaruuden ja sitä ohjaavan logiikan.
AIOptionData	Kuvaa yhtä huomioitavaa toiminnan vaihtoehtoa tekoälyssä. Sille lasketaan hyötyarvo ja liitettyjä toimintoja suoritetaan.
AIActionBase	Abstrakti luokka, joka on pohjana kaikille tekoälyn toiminnoille.
AIValueBase	Abstrakti luokka, josta kaikki pelin aikana laskettavia arvoja palauttavat luokat perivät.
AIRule	AIRule-nimiset luokat ovat erillisiä laskutoimituksia, joiden avulla pelin aikana lasketut arvot muokataan hyötyarvoiksi.
IGraphFloatProvider-interface	Tekoälyn hyötyarvojen laskennassa käytetty rajapinta.

Taulukko 1. Työssä toteutetun tekoälyn pelinaikaisessa suorituksessa käytetyt luokat

Luokat ovat yhteydessä toisiinsa seuraavalla tavalla. (Kuva 2.) AIComponent-luokan instanssi asetetaan scene-objektiin. AIComponent vaatii viittauksen AIGraphRuntimeComponent-objektiin, joka sijaitsee projektin tiedosto-objektien joukossa. Unity-moottori lataa tiedosto-objektin automaattisesti muistiin, jos scenessä sijaitseva objekti viittaa siihen. AIGraphRuntimeComponent-ob-

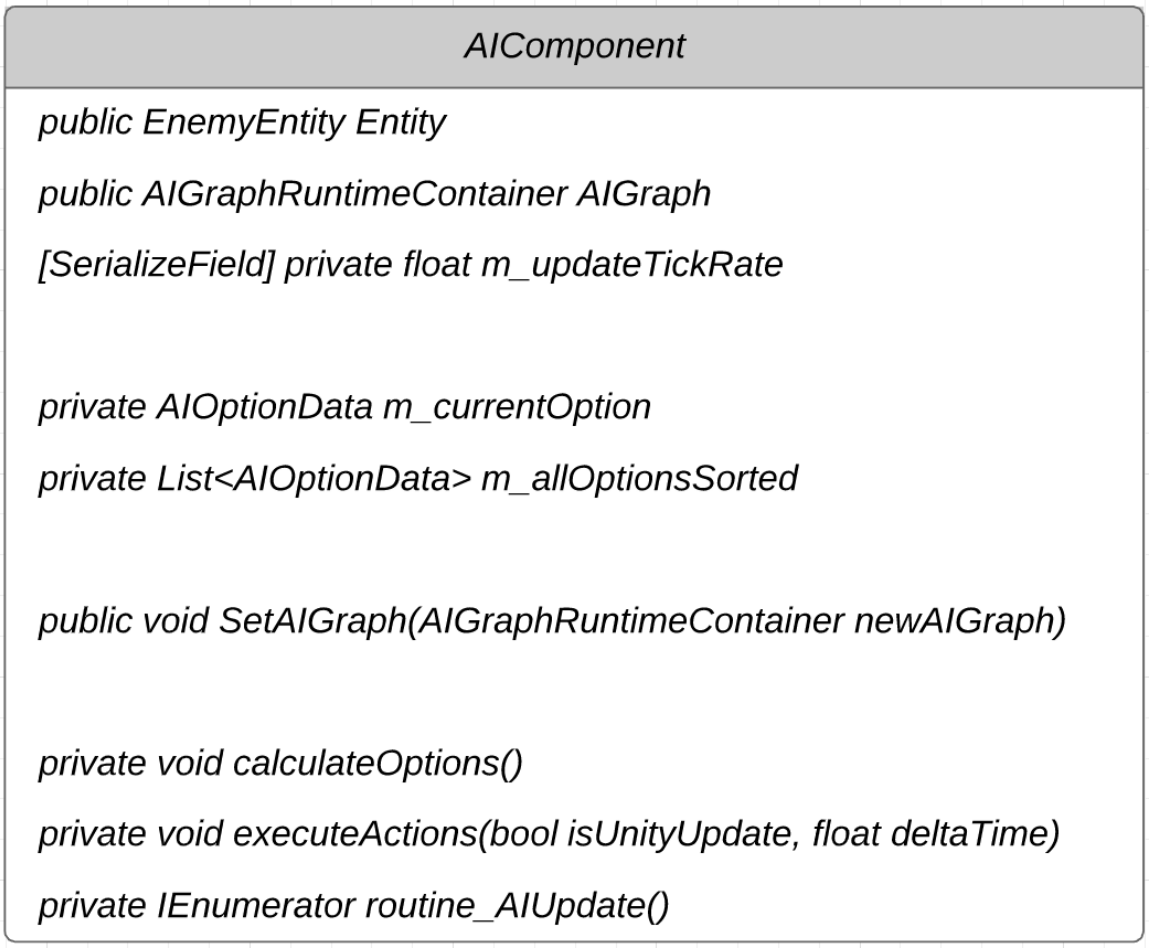
jekti poimii tarvittavat float-arvot yksittäisen AIComponent-instanssin kautta, joten samaa tiedosto-objektia voidaan käyttää usean peliobjektin tekoälyn suorittamiseen.



Kuva 2. Tekoälyn pelinaikaisessa suorituksessa käytettyjen luokkien rakenne ja yhteydet

4.2 AIComponent

AIComponent on luokka, joka hallitsee yksittäistä tekoälyä. Sen tehtävänä on suorittaa sille syötettyä AIGraphRuntimeContainer-objektia, joka sisältää kaiken tekoälyn päätöksentekoon tarvittavan logiikan. Seuraava kuva esittää luokan sisältämät tärkeimmät muuttujat ja funktiot. (Kuva 3.)



Kuva 3. AIComponent-luokan tärkeimmät muuttujat ja funktiot

AIComponent perii Unity-moottorin MonoBehaviour-luokasta. MonoBehaviour-luokka on sidottu Unity-sovelluksen elinkaareen ja voi vastaanottaa kutsuja sovelluksen suorituksen silmukan eri vaiheissa. Luokan täytyy periä MonoBehaviour-luokka, jotta sen voi liittää scene-objekteihin kuten minkä tahansa muun Unity-moottorin peruskomponentin. (Unity Technologies 2020.) MonoBehaviour-luokan sisällä voi myös hyödyntää Coroutine-funktioita. Ne ovat erityisiä funktioita, jotka voivat keskeyttää suorituksen ja odottaa useita pelin silmukan suorituksia, kunnes sille mää-

ritelty sääntö toteutuu. AIComponent hyödyntää Coroutine-käytöstä tekoälyn päivityksen laskennassa. Kun MonoBehaviour-luokan Start()-funktio kutsutaan, käynnistää AIComponent oman Coroutine-silmukan, joka suorittaa tekoälyn päätöksenteon tietyin aikavälein.

Käyttäjä voi määrittää editorissa float-arvon, joka määrää, kuinka tiheästi tekoälyn laskenta toteutetaan. Tämä arvo tulee asettaa aina tilannekohtaisesti. Oman kokemukseni mukaan noin viisi kertaa sekunnissa on riittävän tarkka arvo jopa nopeatempoisessa toimintapelissä. Päivitystiheyden harventaminen on tehokas tapa parantaa koodin suorituskykyä ilman muutosta tekoälyn käytökseen pelaajan näkökulmasta. Sen avulla jokaisen sovelluksen pääsilmukan päivityksen aikana käytetään mahdollisimman vähän aikaa yksittäisen tekoälyn päätöksentekoon ja tekoälyn kokonaisvaltainen suoritus on mahdollisimman kevyt tehtävä prosessorille.

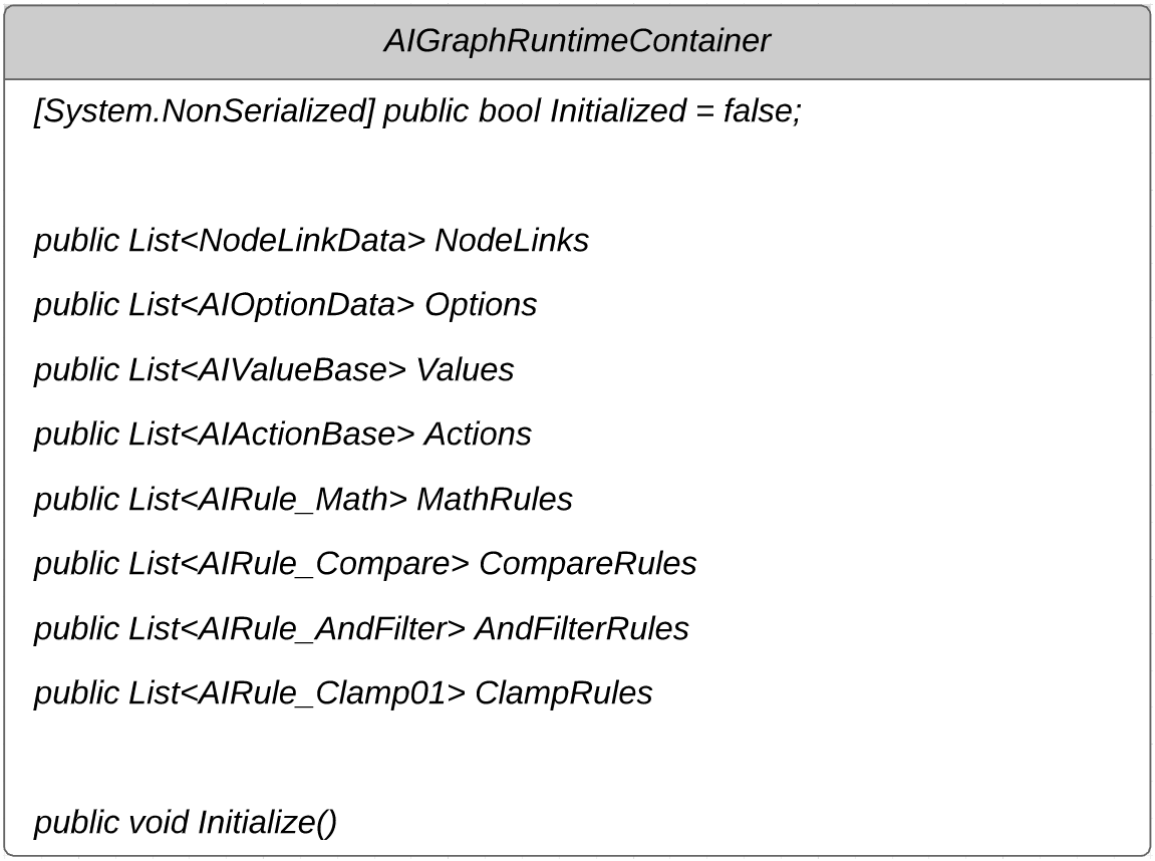
Tekoälyn päätöksenteko aloitetaan laskemalla jokaisen AIGraphRuntimeContainer-objektiin tallennetun AIOptionData-instanssin hyötyarvo. AIComponent muodostaa niistä listan (List<AIOptionData>), joka järjestetään laskevaan järjestykseen lopullisen lasketun hyötyarvon perusteella. Listan ensimmäinen AIOption valitaan. AIComponent pitää muistissa viimeksi valittua AIOptionData-instanssia. Jos päätöksenteon tulos ei vastaa muistissa olevaa yksilöä, niin vanhaan vaihtoehtoon kiinnitettyjen toimintojen OnActionEnd-funktiot suoritetaan ensin. Sen jälkeen suoritetaan uuden vaihtoehdon toimintojen OnActionBegin. Tämän jälkeen uuden vaihtoehdon toiminnot suoritetaan normaalisti.

Toimintojen suoritus tapahtuu pääosin aina päätöksenteon jälkeen. Yksittäiset toiminnot voidaan tarvittaessa myös suorittaa jokaisella sovelluksen silmukan päivityksellä. Jos AIActionBase-luokan bool-arvo ExecuteEveryFrame on tosi, sen suoritus toteutetaan myös päätöksenteon ulkopuolella AIComponent-luokan Update-funktion tahdissa. Update-funktio on osa Unity-moottorin MonoBehaviour-luokkaa, ja se suoritetaan yhden kerran jokaisen sovelluksen silmukan päivityksen aikana.

EnemyEntity on työkalun kehitysprojektissa käytetty luokka, jonka kautta tekoäly pääsee käsiksi pelin aikana laskettuihin muuttujiin. Sen kautta tekoäly myös suorittaa kaikki toiminnot. EnemyEntity koostuu pääosin viitauksista muihin komponentteihin, kuten animaatiokomponenttiin, liikkumiskomponenttiin sekä näköaistikomponenttiin.

4.3 AIGraphRuntimeContainer

AIGraphRuntimeContainer on luokka, jonka kautta AIComponent suorittaa kaiken päätöksenteon. Se sisältää kaiken editorityökaluilla luodun tiedon serialisoidussa muodossa. Alapuolen kuva esittää luokan sisällön. (Kuva 4.)



Kuva 4. AIGraphRuntimeContainer-luokan tärkeimmät muuttujat ja funktiot

AIGraphRuntimeContainer perii ScriptableObject-luokan. Tämän avulla serialisoitu AIGraphRuntimeContainer-objekti voidaan tallentaa .asset-tiedostomuotoon ja sitä voidaan käsitellä Unity Editorissa kuten mitä tahansa muuta tunnistettua tiedostomuotoa. Serialisoidun objektin viittaus voidaan asettaa AIComponent-luokan kenttään kehitysvaiheessa editorissa ja kaikki myöhemmät muutokset tiedostoon eivät pura aikaisemmin tehtyä viittausta. Usea AIComponent-instanssi voi käyttää samaa AIGraphRuntimeContainer-objektia päätöksenteossa, koska toiminnot valitaan AIComponent-luokan kautta haettujen float-arvojen perusteella. Tiedostopohjaisen toimintatavan toinen vahvuus on, että AIComponentin vaatima viittaus sen käyttämään objektiin voidaan vaihtaa myös pelin suorituksen aikana ilman ongelmia.

Jokainen AIGraphRuntimeComponent-objektin sisältämä solmu, kuten AIOption on myös tallennettu levylle ScriptableObject-luokkana .asset-tiedostomuodossa. Ne on yhdistetty AIGraphRuntimeComponent-objektin alaobjekteiksi funktion UnityEditor.AssetDatabase.AddObjectToAsset([alaobjekti], [ydinobjekti]) avulla tallennusvaiheen aikana.

AIGraphRuntimeComponent koostuu pääasiassa pelkästä tiedosta. Jokaisen editorissa luodun solmutyypin serialisoitu tieto (AIOptionData, AIValueBase jne.) on kerättynä omaan listaan. Solmujen yhteydet on myös tallennettu listaan. Tallennetussa muodossa AIGraphRuntimeComponent ei yhdistä eri solmuja toisiinsa, vaan solmujen yhteydet luodaan pelisovelluksen elinkaaren alussa. Yhteydet luodaan funktion Initialize avulla, joka suoritetaan, kun mikä tahansa AIComponent ensimmäisen kerran käsittelee AIGraphRuntimeComponent-objektin sisältöä. Yhteydet luodaan iteroimalla yhteyksien läpi ja yhdistämällä solmut toisiinsa yhteyksiin tallennettujen GUID-muuttujien avulla.

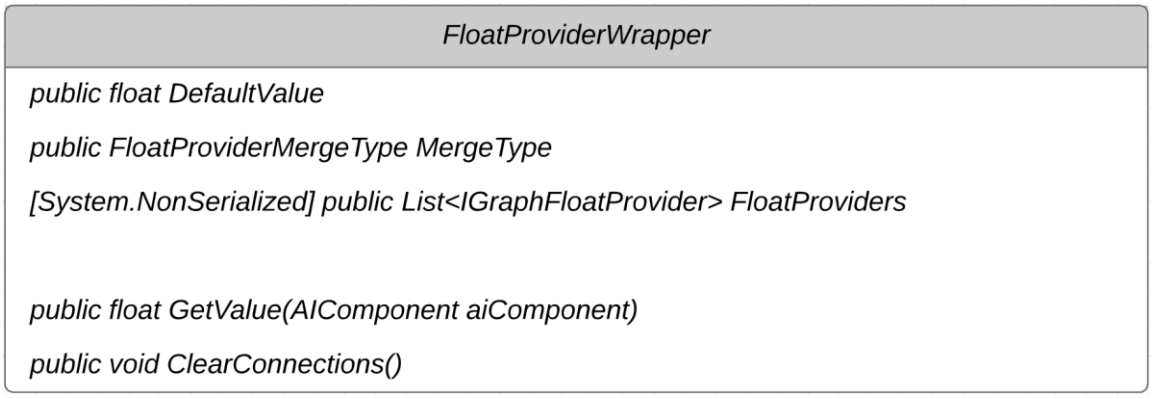
4.4 IGraphFloatProvider-rajapinta

Hyötyarvon laskeminen vaatii, että useat erilaiset luokat voivat hakea ja palauttaa arvoja toisille. IGraphFloatProvider on yksinkertainen rajapinta, joka palauttaa float-arvon AIComponent-tyyppistä parametria vastaan.

```
public interface IGraphFloatProvider
{
    float GetValue(AIComponent aiComponent);
    void ClearConnections();
}
```

Koodiesimerkki 1. IGraphFloatProvider-rajapinta

FloatProviderWrapper-luokkaa käytetään apuna IGraphFloatProvider-rajapinnasta haettujen arvojen kanssa. Se kuvastaa yhtä porttia, johon voi olla liitettynä useita eri IGraphFloatProvider-instansseja. Seuraava kuva esittää FloatProviderWrapper-luokan sisällön. (Kuva 5.)



Kuva 5. FloatProviderWrapper-luokan tärkeimmät muuttujat ja funktiot

FloatProviderMergeType on enum-arvo, joka ohjaa tapaa, miten FloatProviderWrapper yhdistää siihen liitettyjen IGraphFloatProvider-rajapintojen tulokset. Oletusarvo MergeType-muuttujalle on Total.

```
public enum FloatProviderMergeType
{
    Total,
    Average,
    Min,
    Max
}
```

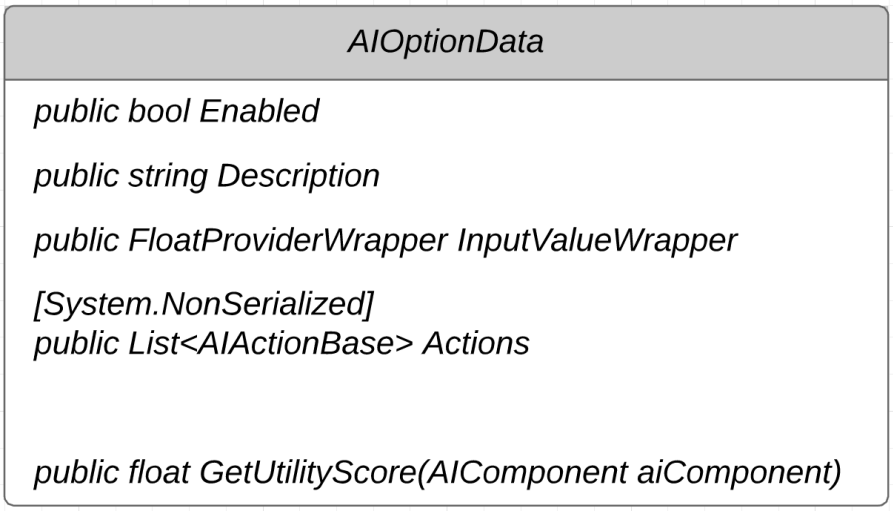
Koodiesimerkki 2. FloatProviderMergeType-enum

Jos FloatProviders-muuttujalista on tyhjä, palauttaa FloatProviderWrapper sille asetetun oletusarvon. Näitä oletusarvoja käytetään paljon esimerkiksi AIRule_Math-luokassa, jossa on kaksi syöttöporttia, jotka molemmat on koodissa toteutettu FloatProviderWrapper-muuttujina. Toinen portti voidaan jättää ilman liitosta, jolloin kyseisen portin oletusarvoa käytetään laskutoimituksessa. AIRule-tyyppisten luokkien toiminta esitellään dokumentissa tarkemmin niiden omassa kappaleessa.

Koska solmujen liitokset luodaan pelisovelluksen suorituksen alussa, on muuttuja FloatProviders merkattu attribuutilla [System.NonSerialized], jonka avulla muuttujan arvoa ei tallenneta. Kaikki IGraphFloatProvider / FloatProviderWrapper -liitokset muodostetaan AIGraphRuntimeComponentin kautta. ClearConnections-funktio kutsutaan aina ennen liitosten luomista. Se varmistaa, että FloatProvider-rajapintaa käyttävät objektit ovat aina oletustilassa, kun solmuja aletaan yhdistämään.

4.5 AIOptionData

Tekoälyn päätöksenteon valintojen vaihtoehtoina toimii luokka AIOptionData. Se perii ScriptableObject-luokan ja on tallennettu levylle AIGraphRuntimeComponent-ydinobjektin alle. Kuva 6 esittää luokan tärkeimmät funktiot ja muuttujat. (Kuva 6.)



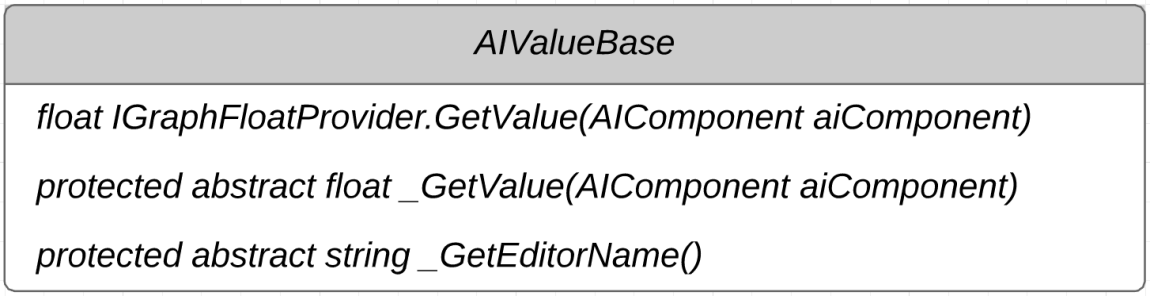
Kuva 6. AIOptionData-luokan tärkeimmät muuttujat ja funktiot

GetUtilityScore-funktio palauttaa float-arvon, jota käytetään tekoälyn hyödyllisimmän toiminnan valinnassa. Kyseinen funktio hakee arvon InputValueWrapper-muuttujasta, joka taas palauttaa siihen linkitetyn IGraphFloatProvider-rajapinnan arvon tai oletusarvon. Hyötyarvon laskennan ketju aloitetaan aina AIOptionData-objektista. Laskennassa käytettyjen funktioiden kutsujärjestys tapahtuu siis visualisoituna oikealta vasemmalle tai lopullisesta tuloksesta arvojen lähteisiin.

Enabled-muuttujan avulla voidaan lisätä tai poistaa yksittäinen valintamahdollisuus päätöksentekoprosessin kaikista valintamahdollisuuksista. Jos Enabled-muuttuja on epätosi, ei valinnalle lasketa ollenkaan hyötyarvoa eikä sen toimintoja suoriteta missään tapauksessa. Tämän muuttujan avulla on yksinkertaista pakottaa tekoäly käyttäytymään halutulla tavalla ja testata tiettyjä toimintoja. Enabled-muuttujaa ei tule käyttää osana tekoälyn lopullista päätöksenvalintaprosessia. Valinnat on parasta laskea aina hyötyarvojen pohjalta. Valintojen senhetkistä saatavuutta voi kuvata esimerkiksi kertomalla hyötyarvon viimeisenä luvulla 1 tai 0, joka käytännössä lisää tai poissulkee vaihtoehdon muiden joukosta.

4.6 AIValueBase ja siitä perivät luokat

AIValueBase on abstrakti luokka, joka on pohjana kaikille pelkkiä arvoja palauttaville luokille. Kuten kaikki muutkin solmuobjektit, se perii ScriptableObject-luokan ja tallennetaan levylle AIGraphRuntimeComponent-ydinobjektin alle. Alapuolen kuva esittää luokan funktiot. (Kuva 7.)



Kuva 7. Abstraktin AIValueBase-luokan tärkeimmät funktiot

AIValueBase implementoi IGraphFloatProvider-rajapinnan ja palauttaa sen avulla float-arvon. Kun IGraphFloatProvider-rajapinnan funktio GetValue kutsutaan, palauttaa AIValueBase siitä perivässä luokassa implementoidun _GetValue-funktion arvon.

Uusien arvoja palauttavien solmutyyppien luominen on hyvin nopeaa. Käyttäjän tarvitsee luoda uusi luokka, joka perii AIValueBase- luokan. Funktiot _GetEditorName ja _GetValue vaativat toteutukset sekä luokka täytyy merkitä serialisoitavaksi attribuutilla [System.Serializable]. Samaa toimintatapaa käytetään hyväksi myös toimintojen luomisessa.

Seuraavaksi esitetään kaksi esimerkkiä AIValueBase-luokan käytöstä. Koodiesimerkissä 3 palautetaan float-arvo 1.0f, jos pelihahmolla on hallussaan tietynlainen ase. Muuten palautetaan arvo 0.0f. (Koodiesimerkki 3.) Koodiesimerkki 4 taas palauttaa kulman pelihahmon ja taistelukohteen välillä. Palautettava arvo voidaan halutessaan normalisoida arvojen 0 ja 1 välille bool-muuttujan avulla. (Koodiesimerkki 4.)

```
[System.Serializable]
public class AIValue_HasWeaponOfType : AIValueBase
{
    public EnemyEntity.EquippedWeaponType WeaponType;

    protected override string _GetEditorName()
    {
        return "Has Weapon Of Type";
    }

    protected override float _GetValue(AIComponent aiComponent)
    {
        switch (WeaponType)
        {
            case EnemyEntity.EquippedWeaponType.LightMelee:
                if (aiComponent.Entity.HasLightMelee) return 1.0f;
                break;

            case EnemyEntity.EquippedWeaponType.LightPistol:
                if (aiComponent.Entity.HasLightPistol) return 1.0f;
                break;

            default: return 0.0f;
        }

        return 0.0f;
    }
}
```

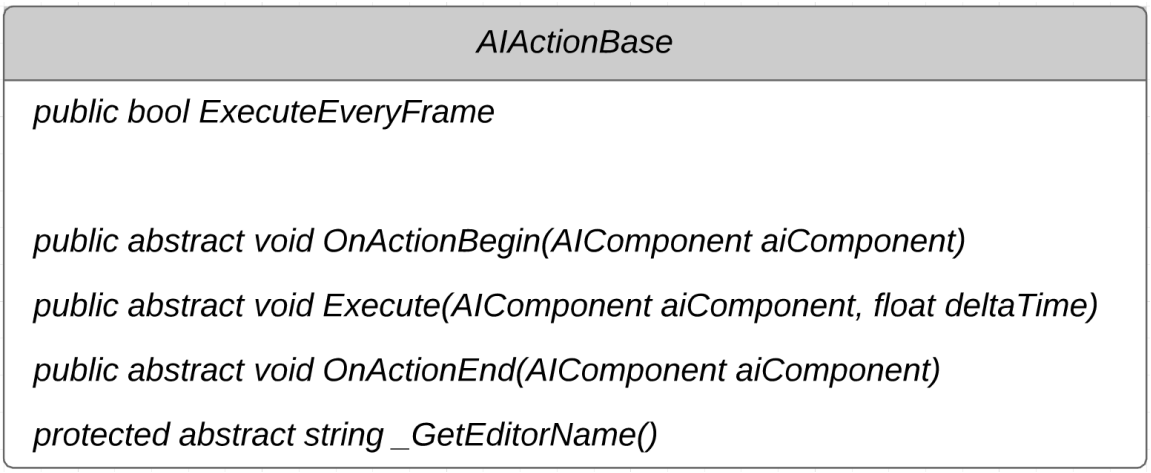
Koodiesimerkki 3. AIValueBase-luokan implementointi luokassa AIValue_HasWeaponOfType

```
[System.Serializable]
public class AIValue_AngleToTarget : AIValueBase
{
    public bool Normalize = false;

    protected override string _GetEditorName()
    {
        return "Angle To Target";
    }

    protected override float _GetValue(AIComponent aiComponent)
    {
        if (Normalize)
            return aiComponent.Entity.AngleToTarget / 180.0f;

        return aiComponent.Entity.AngleToTarget;
    }
}
```

Koodiesimerkki 4. AIValueBase-luokan implementointi luokassa AIValue_AngleToTarget

4.7 AIActionBase ja siitä perivät luokat

AIActionBase-luokkaa käytetään pohjana kaikille tekoälyn toiminnoille. Se on abstrakti luokka, joka tallennetaan ScriptableObject-luokan .asset-tiedostomuodossa AIGraphRuntimeComponent-ydinobjektin alle. AIActionBase-luokan rakenne ja käyttö vastaavat hyvin paljon AIValueBase-luokkaa. Uudet toiminnot perivät abstraktin pohjaluokan ja implementoivat tarvittavat funktiot. Seuraava kuva esittää luokan sisällön. (Kuva 8.)



Kuva 8. Abstraktin AIActionBase-luokan tärkeimmät muuttujat ja funktiot

Execute-funktio on luokan tärkein osa. Se kutsutaan AIComponent-luokasta aina päätöksenteon jälkeen. Jos AIComponent-luokan edellinen valinta ei ole sama kuin uusi tulos, suoritetaan ensin vanhan valinnan toimintojen funktio OnActionEnd. Sen jälkeen suoritetaan uuden valinnan toimintojen OnActionBegin-funktio.

Execute-funktion deltaTime-parametri määritetään ExecuteEveryFrame-muuttujan perusteella. Parametri deltaTime on numeerinen arvo, joka ilmaisee kulunutta aikaa viimeisestä sovelluksen silmukan suorituksesta. Jos muuttuja ExecuteEveryFrame on epätosi, deltaTime asetetaan AIComponent-luokan päivitystiheydeksi. Unity-moottorin Time.deltaTime-arvoa käytetään siinä tilanteessa, jos toiminto vaatii suoritusta jokaisella sovelluksen silmukan päivityksellä. Delta time-parametri mahdollistaa, että toiminnon lasketa tuottaa yhtenäisiä tuloksia muuttuvilla suoritusnopeuksilla ja erilaisissa suoritusympäristöissä.

Seuraava koodiesimerkki on yksinkertainen tapaus AIActionBase-luokan käytöstä. Toiminto asettaa suorituksen alussa hahmon liikkumisesta vastaaville komponenteille tietyn kohteen. Tästä

eteenpäin komponentit käsittelevät hahmon liikkumisen itsenäisesti, joten tekoäly suorittaa vaan funktion, joka asettaa hahmolle tilanteeseen sopivan animaation. (Koodiesimerkki 5.)

```
[System.Serializable]
public class AIAction_ChaseTarget : AIActionBase
{
    protected override string _GetEditorName()
    {
        return "Chase Target";
    }

    public override void OnActionBegin(AIComponent aiComponent)
    {
        if (aiComponent.Entity.CombatTarget == null)
        {
            aiComponent.Entity.CombatTarget
               = GameState.GetInstance().PlayerObject.transform;
        }

        aiComponent.Entity.MovementComponent.ClearPath();
        aiComponent.Entity.MovementComponent.Enabled = true;
        aiComponent.Entity.NavTarget = NavigationTarget.CombatTarget;

        aiComponent.Entity.MovementComponent.MovementSpeed
           = aiComponent.Entity.BaseSettings.MovementSpeed_Run;
    }

    public override void Execute(AIComponent aiComponent, float deltaTime)
    {
        if (aiComponent.Entity.Enabled == false) return;
        if (aiComponent.Entity.AnimationLock) return;

        aiComponent.Entity.MovementAnimationLogic();
    }

    public override void OnActionEnd(AIComponent aiComponent)
    {

    }
}
```

Koodiesimerkki 5. AIActionBase-luokan implementointi luokassa AIAction_ChaseTarget

4.8 AIRule-luokat

AIRule-nimiset luokat suorittavat laskutoimituksia, tai muuten suodattavat float-arvoja. AIRule-luokat eivät käytä pohjana abstraktia luokkaa, koska jokainen AIRule-luokka käyttäytyy eri tavalla. Jokainen AIRule kuitenkin toteuttaa rajapinnan IGraphFloatProvider ja palauttaa sen kautta float-arvon.

AIRule-luokat ovat tallennettuna samaan tapaan ScriptableObject-luokan .asset-tiedostona tekoälyn ydinobjektin alle. AIRule-tyyppisten luokkien kokonaismäärä on hyvin rajattu, joten jokainen erityinen AIRule-luokka on AIGraphRuntimeComponent-luokassa oman tyyppisessä muuttujalistassa. Abstraktin pohjaluokan tarve kasvaa siinä tilanteessa, jos erilaisten sääntöjen määrä lisääntyy.

Monet AIRule-luokat kohtelevat float-arvoja kuten bool-arvoja. Työkalun koodissa float-arvo 1.0f tai suurempi tarkoittaa, että arvo on tosi. Muilla float-arvoilla se käsitellään olevan epätosi. Näiden kyllä/ei -sääntöjen avulla voidaan ohjata mahdollisten valintojen saatavuutta hyvin tarkasti.

AIRule-luokan esimerkkinä voidaan käyttää AIRule_AndFilter-luokkaa. Sillä on yksi sisääntulona toimiva FloatProviderWrapper-muuttuja nimeltä Inputs joka voi vastaanottaa rajattoman määrän yhteyksiä. Jokainen siihen kiinnitetty IGraphFloatProvider-rajapinta täytyy palauttaa float-arvon joka on vähintään 1.0f, jotta luokka palauttaa arvon 1.0f. Muuten palautetaan 0.0f. (Koodiesimerkki 6.)

```
float IGraphFloatProvider.GetValue(AIComponent aiComponent)
{
    float _result = Inputs.GetValue(
       aiComponent,
       FloatProviderMergeType.Min);

    if (_result < 1.0f)
    {
        return 0.0f;
    }

    return 1.0f;
}
```

Koodiesimerkki 6. IGraphFloatProvider-rajapinnan GetValue-funktio luokassa AIRule_AndFilter

Toinen yksinkertainen esimerkki on AIRule_Math, joka suorittaa laskutoimituksen kahden FloatProviderWrapper-muuttujista palautetun arvon välillä. Tässä luokassa on siten kaksi syöttö-

porttia, jotka molemmat voivat vastaanottaa vain yhden linkin. Toinen portti voidaan jättää ilman linkkiä, jos halutaan suorittaa laskutoimitus editorissa määritetyn float-arvon perusteella. (Koodiesimerkki 7.)

```
        float IGraphFloatProvider.GetValue(AIComponent aiComponent)
        {
            float _result = 0;

            float _a = InputA.GetValue(aiComponent);
            float _b = InputB.GetValue(aiComponent);

            switch (OperationType)
            {
                case AIMathOperation.Add: _result = _a + _b; break;
                case AIMathOperation.Subtract: _result = _a - _b; break;
                case AIMathOperation.Divide: _result = _a / _b; break;
                case AIMathOperation.Multiply: default: _result = _a * _b; break;
            }

            return _result;
        }
```

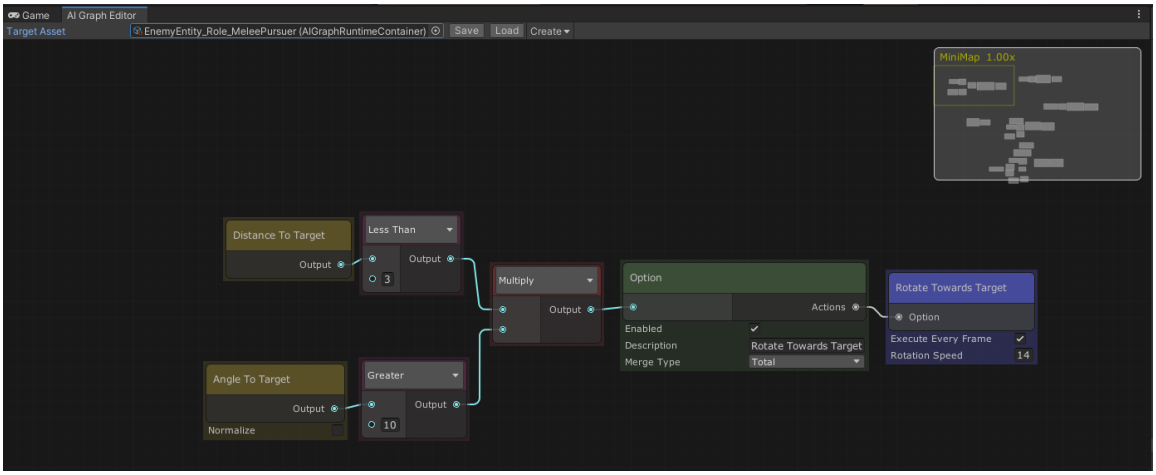
Koodiesimerkki 7. IGraphFloatProvider-rajapinnan GetValue-funktio luokassa AIRule_Math

5 Editorityökalu

Tekoälyn muokkaamista varten luotiin visuaalinen solmupohjainen työkalu. Päätöksenteon visualisointi 2D-tasolle helpottaa suoritusjärjestyksen ja yhteyksien hahmottamista. Visuaalisen työkalun käyttö ei vaadi laajaa ymmärrystä tekoälyn koodin rakenteesta. Käytöksen ymmärtäminen ja muokkaaminen pitäisi olla mahdollista kaikille, joilla on peruskäsitys siitä, miten Unity-editorissa navigoidaan yleisesti.

Työkalu toteutettiin pääosin Unity-editorin UIElements-rajapintaa hyödyntämällä. Työkalu avataan omaksi välilehti-ikkunaksi Unity-editorin työkalupalkista ja sen kokoa voi muokata kuten mitä tahansa muuta Unity-editorin ikkunaa. Kaikki yhteen AIGraphRuntimeComponent-objektiin kuuluvat solmut piirretään samalle tasolle. Solmu on työkalussa yksi neliskulmainen elementti, jonka reunoilla on muuttuva määrä portteja. Porttien välisillä viivoilla kuvataan solmujen linkkejä. Solmujen keskiosassa voi myös esiintyä muokattavia muuttujia.

Tekoälyeditorin oma työkalupalkki sijaitsee ikkunan yläreunassa. Sen kautta voi ladata ja tallentaa halutun AIGraphRuntimeComponent-objektin. Uudet solmut luodaan työkalupalkin Create-valikosta. Työkalun 2D-tason kameraa voi liikuttaa painamalla hiiren rullapainiketta pohjassa. Solmujen väliset linkit luodaan painamalla hiiren vasemmanpuoleista painiketta pohjassa ja vetämällä solmun portista yhteys toiseen porttiin. Alapuolella esitetään kuva tämän opinnäytetyön lopputuloksesta. (Kuva 9.)



Kuva 9. Kuvankaappaus Unity-editorin ikkunasta, opinnäytetyön aikana toteutetun työkalun lopputulos

5.1 UIElements-rajapinta

UIElements on uusi editorityökaluja varten rakennettu järjestelmä. Unity-editorin tuoreimmat työkalut, kuten Visual Effect Graph ja Shader Graph ovat rakennettu käyttämällä sitä (Unity Technologies 2020). UIElements-rajapinnan yksi tärkeimmistä ominaisuuksista on sen tarjoama suorituskyky, kun käyttöliittymän rakenne on hyvin kompleksi. Vanhat Unity-editorin tarjoamat rajapinnat käyttöliittymien luomiseen eivät skaalaudu hyvin tilanteissa, joissa piirretään lukuisia muokattavia elementtejä ruudulle yhtä aikaa. (Bahnassi 2019.)

UIElements-rajapinta sallii käyttäjän eritellä käyttöliittymää ajavan koodin, rakenteen ja visuaalisen tyylin erillisiin tiedostoihin. Tämä menetelmä imitoi internetsivujen rakennusprosessissa käytettyä työnkulkua, jossa XML-tiedostojen sisällön tyyli määritellään CSS-tyyppisissä tiedostoissa. (Campeanu 2019.)

UIElements-rajapinnalla luodun käyttöliittymän rakenne voidaan määrittää UXML-tyyppisten tekstitiedostojen avulla (Campeanu 2019). Tekoälyn editorin rakentamissa ei ollut tarvetta hyödyntää UXML-tiedostoja, koska kaikkien visuaalisten elementtien rakenne pysyi hyvin yksinkertaisena. Rakenteen voi myös luoda suoraan C#-koodissa ja tekoälyn työkalussa toimitaan niin. Käyttöliittymän visuaalisten elementtien tyylin voi määrittää USS-tyyppisissä tiedostoissa (Unity Technologies 2020). Ne vastaavat CSS-tiedostoja rakenteeltaan. Alapuolen koodiesimerkki esittää editorityökalun 2D-tason taustan asetukset .uss-tiedostossa. (Koodiesimerkki 8.)

```
GridBackground
{
	--grid-background-color: #181818;
	--line-color: rgba(193, 196, 192, 0.02);
	--thick-line-color: rgba(193, 196, 192, 0.012);
	--spacing: 20;
}
```

Koodiesimerkki 8. Esimerkki .uss-tiedostosta

Tekoälyn työkalussa käytetään myös Unity-editorin GraphView-luokkaa. Sen avulla toteutetaan työkalussa käytetty 2D-taso ja siinä sijaitsevat solmut. GraphView oli työn toteutuksen aikana vielä keskeneräinen rajapinta ja se kuului UnityEditor.Experimental-nimiavaruuteen.

5.2 AIGraph-luokka

Tekoälytyökalun pohjana toimii AIGraph-luokka. Sen tehtävänä on hallita työkalun ikkunaa Unity-editorissa ja suorittaa käyttäjän ohjaamat komennot.

5.2.1 Editori-ikkunan luonti ja avaaminen

Unity-editorissa uusi ikkuna voidaan luoda ja avata seuraavassa esimerkissä esitetyllä tavalla. (Koodiesimerkki 9.)

```
[MenuItem("Tools/AI Graph Editor")]
public static void OpenWindow()
{
    var _window = GetWindow<AIGraph>();
    _window.titleContent = new GUIContent("AI Graph Editor");
}
```

Koodiesimerkki 9. Funktio, jolla avataan tekoälytyökalun ikkuna Unity-editorissa

MenuItem-attribuutin avulla voi suorittaa staattisia funktioita Unity-editorissa päätyökalupalkin kautta. Attribuutille määritetty tekstiparametri asettaa tässä tapauksessa funktion Tools-nimisen kategorian alle ja painikkeelle nimen ”AI Graph Editor”. Attribuutti sisältyy UnityEditor-nimiavaruuteen. GetWindow-funktio palauttaa uuden UnityEditor.EditorWindow-luokan instanssin. Kyseinen luokka käyttäytyy samalla tavalla, kuin mikä tahansa Unity-editorin välilehti, joten se voidaan osoittimen avulla siirtää tai telakoida ruudulla käyttäjän valitsemaan paikkaan. Ikkunan kokoa voidaan myös muuttaa milloin tahansa. Seuraavaksi luodaan 2D-taso ja ruudukko.

Koodiesimerkki 10 esittää 2D-tason luontiin käytetyn koodin. (Koodiesimerkki 10.) AIGraphView-luokka on GraphView-luokasta perittävä luokka, joka suorittaa tarkemmin kaikkia 2D-tasoon liittyviä toimintoja. Muuttuja rootVisualElement tarkoittaa tässä asiayhteydessä luodun ikkunan sisältöä. Funktio StretchToParentSize sovittaa AIGraphView-elementin automaattisesti ikkunan kokoiseksi. Tarkempi dokumentaatio AIGraphView-luokalle sijaitsee kappaleessa 5.3. Ikkuna voidaan nyt avata ja sen sisälle piirretään AIGraphView-luokan uusi instanssi.

```
        private void constructGraph()
        {
            m_graphView = new AIGraphView(this)
            {
                name = "AI Graph"
            };

            m_graphView.StretchToParentSize();
            rootVisualElement.Add(m_graphView);
        }
```

Koodiesimerkki 10. AIGraphView-luokan uuden instanssin luominen ja lisääminen AIGraph- ikkunaan

5.2.2 Työkalupalkki

Käyttäjä Työkalupalkin luominen onnistuu yksinkertaisesti, koska UIElements-rajapinta sisältää jo Toolbar-luokan, jota voidaan hyödyntää tässä tilanteessa. Toolbar-luokka perii VisualElement-luokan, joka tarkoittaa, että siihen voidaan liittää alaelementtejä funktion VisualElement.Add(VisualElement child) avulla. Seuraava koodiesimerkki esittää, kuinka yksinkertaista työkalupalkin lisääminen on käytännössä. (Koodiesimerkki 11.) Työkalupalkkiin lisätään seuraavaksi kenttä AIGraphRuntimeComponent-objektien valintaa varten.

```
            var _toolbar = new Toolbar();

            // kaikki työkalupalkin sisältö voidaan lisätä tässä välissä

            rootVisualElement.Add(_toolbar);
```

Koodiesimerkki 11. Työkalupalkin luominen ja lisääminen AIGraph-luokassa

ObjectField-luokkaa käytetään, kun halutaan käsitellä UnityEngine.Object-luokasta perivää objektia editorissa. ObjectField-luokan visuaalinen muoto esitetään alapuolella kuvassa. (Kuva 10.) Bind-funktio ja bindingPath-muuttujan avulla visuaalisen elementin yhteys sidotaan serialisoituun objektiin. Muuttuja bindingPath on reitti Bind-funktiolla asetetusta objektista haluttuun muuttujaan. Se asetetaan osoittamaan AIGraph-luokan muuttujaan CurrentContainer. Kenttä voidaan pakottaa tunnistamaan pelkästään haluttuja objekteja objectType-muuttujan avulla.

Kun käyttäjä painaa hiirellä luokan visuaalista elementtiä, avautuu näkyviin hakuikkuna. Hakuikkuna näyttää kaikki projektissa sijaitsevat objektit, joiden luokka vastaa elementille määritettyä luokkaa.

Kuva 10. Kuvankaappaus ObjectField-elementistä, joka käsittelee AIGraphRuntimeContainer-objekteja

RegisterCallback-funktio on UIElements-rajapinnassa käytetty funktio, jonka avulla voidaan sitoa koodin toimintoja erilaisiin käyttäjän aiheuttamiin tapahtumiin. Yläpuolella esitetyssä tapauksessa aina kun _containerField-elementin sisältö muuttuu, koodi yrittää ladata uuden valinnan tiedon ja päivittää 2D-tason visuaaliset elementit sen mukaan. Suurin osa työkalun toiminnasta suoritetaan näiden funktioiden kautta.

Seuraavassa koodiesimerkissä luodaan uusi ObjectField-instanssi ja lisätään se aiemmin luotuun työkalupalkkiin. Elementti käsittelee AIGraphRuntimeContainer-luokan instansseja ja käyttäjän aiheuttamat muutokset käynnistävät tiedon latausoperaation. (Koodiesimerkki 12.)

```
var _containerField = new ObjectField();
_containerField.Bind(new SerializedObject(this));
_containerField.bindingPath = "CurrentContainer";
_containerField.objectType = typeof(AIGraphRuntimeContainer);
_containerField.allowSceneObjects = false;
_containerField.label = "Target Asset";
_containerField.RegisterCallback
    <ChangeEvent<UnityEngine.Object>>(evt =>
{
    CurrentContainer = (AIGraphRuntimeContainer)evt.newValue;
    loadFile();
});
_containerField.MarkDirtyRepaint();
_toolbar.Add(_containerField);
```

Koodiesimerkki 12. ObjectField-elementin lisääminen AIGraphRuntimeContainer-objektin valintaa varten AIGraph-luokassa

Seuraavaksi työkalupalkkiin lisätään näppäimet tiedostojen tallentamista ja lataamista varten. Alapuolen koodiesimerkki esittää, kuinka se toteutetaan työkalun koodissa. (Koodiesimerkki 13.) Button-luokka sisältyy myös UIElements-rajapintaan, joten sen toteutus on yksinkertaista. Luokan uusi instanssi ottaa vastaan System.Action-tyyppisen parametrin, joka tarkoittaa funktiota, joka kutsutaan näppäintä painaessa.

```
            var _saveButton = new Button( () => { saveFile(); });
            _saveButton.text = "Save";
            _toolbar.Add(_saveButton);

            var _loadButton = new Button( () => { loadFile(); });
            _loadButton.text = "Load";
            _toolbar.Add(_loadButton);
```

Koodiesimerkki 13. Tallennus- ja latauspainikkeiden luominen ja lisääminen työkalupalkkiin AIGraph-luokassa

Alapuolen koodiesimerkki esittää funktiot, joiden suoritus aloitetaan tallennus- ja lataamispainikkeita painettaessa. (Koodiesimerkki 14.) Tiedostojen varsinainen tallentaminen ja lataaminen käsitellään AIGraphSaveUtility-luokan avulla, jonka toiminnallisuuteen syvennytään tarkemmin dokumentin kappaleessa 5.5.

```
    private void saveFile()
    {
        if (CurrentContainer == null)
        {
            UnityEngine.Debug.Log("Save operation aborted.
                Current container == null");
            return;
        }

        var _saveUtility = AIGraphSaveUtility.GetInstance(m_graphView);
        _saveUtility.SaveGraph(CurrentContainer);
        CurrentContainer = _saveUtility.GetCurrentContainer();
    }

    private void loadFile()
    {
        if (CurrentContainer == null)
        {
            UnityEngine.Debug.Log("Load operation aborted.
                Current container == null");
            return;
        }

        var _saveUtility = AIGraphSaveUtility.GetInstance(m_graphView);
        _saveUtility.LoadGraph(CurrentContainer);
     }
```

Koodiesimerkki 14. Funktiot, jotka aloittavat tallennus- ja lataustoiminnot AIGraph-luokassa. Näitä funktioita kutsutaan, kun käyttäjä painaa työkalupalkissa niille tarkoitettuja painikkeita.

Työkalupalkki vaatii enää toiminnot erilaisten solmujen lisäämiseen. Ensin luodaan erityinen painike nimeltä Create. Kyseinen painike hyödyntää UIElements-rajapinnan luokkaa ToolbarMenu.

Se on erityinen painike, joka painamishetkellä avaa pudotusvalikon. Pudotusvalikoon voidaan lisätä sisältöä AppendAction-funktion avulla. Funktion ensimmäinen, string-tyypin parametri määrittää painikkeen polun ja nimen. Seuraavan parametrin avulla voidaan suorittaa koodia aina, kun painiketta painetaan. Oletuksena kaikki uudet painikkeet ovat käyttäjälle näkyvissä ja painettavissa. Kolmannen parametrin avulla voidaan hallita painikkeen tilaa ja näkyvyyttä. Pudotusvalikkoon luodaan myös tyhjä käyttäjälle näkymätön valinta. Tämä valinta suoritetaan, jos käyttäjä painaa hiiren vasemmanpuolista painiketta pudotusvalikon ulkopuolella. Alapuolen koodiesimerkki esittää pudotusvalikon luonnin koodissa. (Koodiesimerkki 15.)

```
var _dropDownButtonCreate = new ToolbarMenu
{
    name = "Create",
    text = "Create"
};

_dropDownButtonCreate.menu.AppendAction(
    "Default (do nothing)",
    action => { },
    action => DropdownMenuAction.Status.None);

_dropDownButtonCreate.menu.AppendAction(
    "General/Option Node",
    action => m_graphView.CreateNode(
        typeof(AIGraphOptionNode),
        typeof(AIOptionData),
        null,
        false));

// muut solmutyypit lisätään tässä välissä

_toolbar.Add(_dropDownButtonCreate);
```

Koodiesimerkki 15. Pudotusvalikon luominen ja lisääminen työkalupalkkiin luokassa AIGraph

5.2.3 Abstraktit luokat pudotusvalikossa

AIGraph-luokan funktio CreateNode vaatii 4 parametria, joista yksi on solmun logiikan implementaatioluokan tyyppi. Useimmissa tapauksissa solmun luokka ja implementaation luokka syötetään funktioon samalla tavalla, kuin OptionNode-luokan esimerkissä. AIOptionData-luokan lisäksi kaikki AIRule-luokat ovat koodissa asetettu pudotusvalikkoon yksi kerrallaan, koska niiden lukumäärä on hyvin pieni ja staattinen. Abstrakteja luokkia hyödyntävät toteutukset, kuten toiminnot ja arvoja palauttavat luokat (AIActionBase-luokat ja AIValueBase-luokat) kuitenkin vaati-

vat, että työkalun koodi tukee uusien luokkien jatkuvaa lisäämistä ja uudelleennimeämistä. Koodin täytyy aina olla tietoinen kaikista kyseisistä pohjaluokista perivistä implementaatioluokista, jotta työkalun käyttö on mahdollisimman vaivatonta.

Koodissa hyödynnetään rajapintoja System.Reflection ja System.Linq, joiden avulla luodaan listat kaikista luokista, jotka perivät tietyistä abstrakteista luokista. Funktio System.Reflection.Assembly.GetTypes lukee projektin kootun koodin viimeisintä versiota ja palauttaa kootun taulukon kaikista projektin luokista. System.Linq-toimintojen avulla tulokset suodatetaan halutun pohjaluokan perusteella ja järjestetään aakkosjärjestykseen. Kaikista luokista luodaan myös muistiin instanssit, jotta niiden funktioita voidaan kutsua. Niitä käytetään myöhemmin myös System.Object-tyyppisinä muuttujina solmujen luonnissa. (Koodiesimerkki 16.)

```
        private IEnumerable<AIValueBase> m_allValueTypes;
        private List<AIValueBase> m_valueTypeList;

        private IEnumerable<AIActionBase> m_allActionTypes;
        private List<AIActionBase> m_actionTypeList;

        private void generateClassNameReferences()
        {
            m_allValueTypes = typeof(AIValueBase)
                .Assembly.GetTypes()
                .Where(t => t.IsSubclassOf(typeof(AIValueBase)) && !t.IsAbstract)
                .Select(t => (AIValueBase)Activator.CreateInstance(t));
            m_valueTypeList = m_allValueTypes.ToList();
            m_valueTypeList.OrderByDescending(o => o.name);

            m_allActionTypes = typeof(AIActionBase)
                .Assembly.GetTypes()
                .Where(t => t.IsSubclassOf(typeof(AIActionBase)) && !t.IsAbstract)
                .Select(t => (AIActionBase)Activator.CreateInstance(t));
            m_actionTypeList = m_allActionTypes.ToList();
            m_actionTypeList.OrderByDescending(o => o.name);
        }
```

Koodiesimerkki 16. Kaikkien abstrakteista pohjaluokista perivien luokkien etsiminen ja kerääminen muistiin luokassa AIGraph

Jokainen toimintoluokka ja arvoluokka lisätään edellisen esimerkin mukaisella tavalla pudotusvalikkoon. (Koodiesimerkki 17.) AppendAction-funktion neljäs parametri userData välittää minkä tahansa System.Object-tyyppisen muuttujan suoritettavalle funktiolle. UserData-parametria käytetään vain, jos funktion ainoa parametri on tyyppiä DropDownMenuAction, joka kuuluu myös UIElements-rajapinnan toiminnallisuuteen. Tämän toimintatavan ansiosta työkalu on aina

tietoinen kaikista käyttäjän luomista implementaatioluokista ja osaa välittää oikean luokan eteenpäin työkalupalkin nappia painettaessa.

```
            for (int i = 0; i < m_valueTypeList.Count; i++)
            {
                IGraphSaveable _saveable = (IGraphSaveable)m_valueTypeList[i];

                _dropDownButtonCreate.menu.AppendAction(
                    $"Value/{_saveable.GetEditorName()}",
                    onButtonCreateValueNode,
                    action => DropdownMenuAction.Status.Normal,
                    m_valueTypeList[i]);
            }
```

Koodiesimerkki 17. AIValueBase-luokasta perivien implementaatioluokkien lisääminen työkalupalkin pudotusvalikkoon AIGraph-luokassa

5.3 AIGraphView-luokka

Editorityökalun 2D-tasoa ja sinne luotuja solmuja hallitsee AIGraphView-luokka. Se perii abstraktin GraphView-luokan, joka on osa UIElements-ympäristöä. VisualElement-luokkaa käytetään UIElements-rajapinnassa kaikkien ruudulle piirrettävien objektien pohjana. Tekoälytyökalussa työkalupalkki, 2D-taso, solmut ja niiden väliset yhteydet pohjautuvat kaikki tähän luokkaan. VisualElement-luokan styleSheets-muuttujan avulla voidaan piirtää elementin sisältö käyttäjän määrittämien tyyliasetusten mukaan. Ylläolevassa esimerkissä lisätään ikkunan pohjaelementtiin, eli 2D-tason tyhjään taustaan .uss-tiedostoon tallennettu StyleSheet-objekti. Seuraavassa esimerkissä esitetään AIGraphView-luokan konstruktori. (Koodiesimerkki 18.)

```
        public AIGraphView(EditorWindow window)
        {
            m_editorWindow = window;

            styleSheets.Add(Resources.Load<StyleSheet>("StyleSheet_AIGraph"));

            var _grid = new GridBackground();
            Insert(0, _grid);
            _grid.StretchToParentSize();

            this.AddManipulator(new ContentDragger());
            this.AddManipulator(new SelectionDragger());
            this.AddManipulator(new RectangleSelector());

            SetupZoom(0.1f, ContentZoomer.DefaultMaxScale);
        }
```

Koodiesimerkki 18. AIGraphView-luokan uuden instanssin luominen ja tarvittavien lisätoimintojen lisääminen

AIGraphView-luokan uuteen instanssiin lisätään UIElements-ympäristöön sisältyviä valmiiksi käytettäviä toimintoja. AddManipulator-funktion avulla lisätä työkaluja, jotka hallitsevat käyttäjän ja visuaalisten elementtien välistä vuorovaikutusta. Lopuksi funktio SetupZoom mahdollistaa ikkunan sisällön suurennustason hallinnan hiiren rullan avulla. (Koodiesimerkki 18.)

5.3.1 Solmujen luonti

AIGraphView-luokan funktio CreateNode käsittelee uuden solmun luomiseen liittyvät toiminnot. Funktio esitetään alapuolen koodiesimerkissä. (Koodiesimerkki 19.) Työkalun solmut koostuvat kahdesta luokasta; solmun editoriluokasta ja tekoälyn päätöksenteossa käytetystä tietoluokasta. Koodissa solmun sisältöä, eli tietoluokkaa kutsutaan implementaatioksi. CreateNode-funktio luo uuden instanssin solmun luokasta, luo sille portit toteutuksen asettamien vaatimusten perusteella ja piirtää implementaatioluokan sisällön muokattavaksi. Lopuksi solmu lisätään 2D-tasolle. Implementaatiosta luodaan solmuun aina uusi instanssi, koska muuten käyttäjän tekemät muutokset päivittyisivät levylle tallennettuun objektiin ilman mahdollisuutta perua muutoksia. Tallennus- ja lataustoimintojen avulla synkronoidaan tieto 2D-tason ja levylle tallennetun objektin välillä.

```
        public AIGraphNodeBase CreateNode(
            Type nodeType,
            Type implementationType,
            ScriptableObject implementationInstance,
            bool isExistingData)
        {
            AIGraphNodeBase _node =
                (AIGraphNodeBase)Activator.CreateInstance(nodeType);

            _node.styleSheets.Add
                (Resources.Load<StyleSheet>("StyleSheet_AIGraphNode"));

            if (isExistingData)
            {
                implementationInstance =
                    ScriptableObject.Instantiate(implementationInstance);
            }
            else
            {
                implementationInstance =
                    ScriptableObject.CreateInstance(implementationType);
            }

            _node.Implementation = implementationInstance;
            _node.title = _node.GetEditorName();
            _node.GenerateLabel(_node.titleContainer);

            generatePorts(_node);


            SerializedObject _serializedImplementation =
                new SerializedObject(_node.Implementation);
            drawProperties(_node, _serializedImplementation);

            _node.RefreshExpandedState();
            _node.RefreshPorts();

            AddElement(_node);

            ... lopuksi asetetaan solmun paikka 2D-tasolla ...

            return _node;
        }
```

Koodiesimerkki 19. CreateNode-funktio AIGraphView-luokassa

Solmun paikka asetetaan Node-luokan SetPosition-funktiolla. Se vaatii parametrinä Rect-tyyppisen muuttujan, joka sisältää solmun koon ja sijainnin 2D-tasolla. Sijainti tallennetaan solmun implementaatioon, jonka avulla solmut palautetaan ladattaessa käyttäjän luomaan asetelmaan. Uutta solmua luotaessa aloitussijainniksi lasketaan aina piste, joka on hieman työkalupalkin alapuolella.

5.3.2 Solmun sisällön piirtäminen

Solmujen implementaatiot sisältävät muuttujia, joiden muokkaus täytyy olla käyttäjälle mahdollista. Jokainen muokattava luokka sisältää vaihtelevan määrän erilaisista muuttujia, joten niiden käsittelyssä täytyy ottaa huomioon kaikki mahdolliset tilanteet. AIGraphView-luokan funktiota drawProperties käytetään kaikkien työkalussa esiintyvien solmujen sisällön piirtämiseen. (Koodiesimerkki 20.)

```
        private void drawProperties(Node node, SerializedObject serializedObject)
        {
            PropertyField _propertyField;

            SerializedProperty _serializedProperty =
                serializedObject.GetIterator();

            while (_serializedProperty.NextVisible(true))
            {
                if (_serializedProperty.displayName == "Script") continue;

                _propertyField = new PropertyField();
                _propertyField.label = _serializedProperty.displayName;
                _propertyField.bindingPath = _serializedProperty.propertyPath;
                _propertyField.BindProperty(_serializedProperty);

                node.Add(_propertyField);
            }
        }
```

Koodiesimerkki 20. Solmun sisällön piirtäminen drawProperties-funktion avulla luokassa AIGraphView

SerializedObject-luokan avulla voidaan UnityEditor-ympäristössä käsitellä mitä tahansa serialisoitavaa objektia yleisellä tasolla. Luokka SerializedProperty puolestaan esittää SerializedObject-instanssin sisältämää yksittäistä muuttujaa. Solmun sisällön piirtäminen aloitetaan luomalla uusi SerializedObject solmun implementaatiosta. Implementaation jokainen serialisoitava muuttuja käydään läpi ja niille luodaan solmuun uusi PropertyField-tyypin visuaalinen elementti. Luokka PropertyField sisältyy UIElements-rajapintaan ja sitä käytetään käsittelemään ja piirtämään erityyppisiä muuttujia. Iteraatiovaiheessa suodatetaan piirrettävien muuttujien joukosta viittaus implementaatioluokan lähdekoodin tiedostoon.

5.3.3 Porttien luonti

Porttien avulla muodostetaan yhteyksiä solmujen välille. UIElements-ympäristön Node-luokassa on funktio, jonka avulla voidaan luoda uusi instanssi portista.

```
        public virtual Port InstantiatePort(
            Orientation orientation,
            Direction direction,
            Port.Capacity capacity,
            Type type);
```

Koodiesimerkki 21. InstantiatePort-funktio luokassa Node

InstantiatePort-funktio luo vaatii neljä parametria, jotka ohjaavat portin toimintaa ja ulkoasua. Enum-tyyppinen Orientation-muuttuja kertoo, asetetaanko uudet portin solmun reunoille pysty- vai vaakasuunnassa. Direction on myös tyyppiä enum, ja se määrittää portin sisään- tai ulostuloksi. Port.Capacity-muuttuja määrää, kuinka monta liitosta portissa voi olla samaan aikaan. Type-muuttujaa asettaa vaatimuksen liitosten luomiselle. Uutta liitosta luodessa molempien porttien täytyy olla samaa tyyppiä, jotta yhteys voidaan muodostaa. (Koodiesimerkki 21.)

Jokainen solmutyyppi tarvitsee tietyn määrän sisään- ja ulostuloportteja. AIGraphNodeBase-luokassa on abstrakti funktio, joka palauttaa tiedon tarvittavista porteista. (Koodiesimerkki 22.) Tämän funktion implementoivat kaikki solmuluokat. PortGeneratorInfo sisältää kaikki yhden portin luomiseen tarvittavat tiedot.

```
        public abstract List<PortGeneratorInfo> GetPorts
```

Koodiesimerkki 22. GetPorts-funktio AIGraphNodeBase-pohjaluokassa

Float-tyyppisten porttien oletusarvo on muokattavissa, jos portissa ei ole yhtään liitosta. Arvon visuaalinen elementti piirretään tai piilotetaan aina, kun käyttäjä muokkaa portin liitoksia. Tämän ansiosta käyttäjä voi halutessaan hyödyntää editorissa asetettuja float-arvoja päätöksenteossa. Kaikki visuaalisten elementtien päivitykset suoritetaan koodissa UIElements-ympäristön Callback-funktioiden kautta.

Seuraava koodiesimerkki esittää GetPorts-funktion käytön AIGraphOptionNode-luokan asiayhteydessä. (Koodiesimerkki 23.) Kyseisessä solmussa on yksi sisääntulo ja yksi ulostulo. Sisääntulona toimiva portti vastaanottaa float-arvoja ja ulostuloporttiin yhdistetään toimintoja.

```
public override List<PortGeneratorInfo> GetPorts()
{
    List<PortGeneratorInfo> _ports = new List<PortGeneratorInfo>();

    PortGeneratorInfo _portIn = new PortGeneratorInfo();
    _portIn.Capacity = Port.Capacity.Multi;
    _portIn.Direction = Direction.Input;
    _portIn.PortConnectionType = typeof(float);
    _portIn.Name = "Utility Score";
    _portIn.FloatProvider =
        ((AIOptionData)Implementation).InputValueWrapper;

    PortGeneratorInfo _portOut = new PortGeneratorInfo();
    _portOut.Capacity = Port.Capacity.Multi;
    _portOut.Direction = Direction.Output;
    _portOut.PortConnectionType = typeof(AIGraphActionNode);
    _portOut.Name = "Actions";

    _ports.Add(_portIn);
    _ports.Add(_portOut);

    return _ports;
}
```

Koodiesimerkki 23. GetPorts-funktion käyttö luokassa AIGraphOptionNode

5.4 AIGraphNodeBase ja sen käyttö

Jokainen tekoälyn editorityökalussa käytetty solmutyyppi toteuttaa abstraktin luokan AIGraphNodeBase. Se perii UIElements-ympäristön Node-luokan, joka tarjoaa laajan pohjan solmupohjaisen työkalun luomiseen. AIGraphNodeBase sisältää viittauksen tekoälyn päätöksenteossa käytettävään implementaatioluokkaan, kuten esimerkiksi luokkaan AIOptionData. Lisäksi se sisältää seuraavat alapuolella esitetyt funktiot. (Koodiesimerkki 24.)

GetEditorName-funktio palauttaa nimen, joka piirretään solmun yläreunaan. Funktiota GenerateLabel käytetään, jos nimen sijasta halutaan esittää erilaisia elementtejä. GetImplementationType-funktio palauttaa päätöksenteossa käytetyn luokan System.Type-muuttujana. SetGUID-funktio käsittelee tekoälyn tallentamisessa ja lataamisessa käytettyjä GUID-muuttujia. Tarkempi dokumentaatio GUID-muuttujista ja IGraphSaveable-rajapinnasta sijaitsee kappaleessa 5.5. (Koodiesimerkki 24.)

```
        public void SetGUID(string guid)
        {
            IGraphSaveable _saveable = (IGraphSaveable)Implementation;

            if (string.IsNullOrEmpty(guid))
            {
                guid = System.Guid.NewGuid().ToString();
            }

            _saveable.GUID = guid;
        }

        public abstract string GetEditorName();
        public abstract Type GetImplementationType();
        public abstract List<PortGeneratorInfo> GetPorts();
        public abstract void GenerateLabel(VisualElement labelElement);
```

Koodiesimerkki 24. AIGraphNodeBase-luokan tärkeimmät funktiot

Seuraava esimerkki esittää AIGraphNodeBase-luokan toteutuksen luokassa AIGraphMathNode. (Koodiesimerkki 25.) Solmun yläreunan elementti korvataan enum-muuttujaan yhdistetyllä visuaalisella elementillä. Normaalisti solmun yläreunaan piirretään solmun nimi, mutta GenerateLabel-funktion avulla voidaan sen tilalle luoda tilanteeseen sopiva visuaalinen elementti.

```
public class AIGraphMathNode : AIGraphNodeBase
{
    public override void GenerateLabel(VisualElement labelElement)
    {
        labelElement.Clear();

        EnumField _enumField =
            new EnumField(((AIRule_Math)Implementation).OperationType);

        _enumField.label = "";
        _enumField.RegisterValueChangedCallback(evt =>
        {
            ((AIRule_Math)Implementation).OperationType =
                (AIMathOperation)evt.newValue;
        });
        _enumField.StretchToParentSize();

        labelElement.Add(_enumField);
    }

    public override Type GetImplementationType()
    {
        return typeof(AIRule_Math);
    }

    public override List<PortGeneratorInfo> GetPorts()
    {
          ...
    }

    public override string GetEditorName()
    {
        return "Math";
    }
}
```

Koodiesimerkki 25. AIGraphNodeBase-luokan käyttö AIGraphMathNode-luokan tapauksessa. GetPorts-funktion sisältö on supistettu esitysmuodon parantamisen vuoksi.

5.5 AIGraphSaveUtility-luokka

Luokka AIGraphSaveUtility suorittaa työkalun avulla muokatun tiedon tallentamisen levylle AIGraphRuntimeContainer-objektiin. Lataustoiminto taas luo solmut ja yhteydet levylle tallennetun objektin perusteella.

5.5.1 IGraphSaveable-rajapinta

Rajapintaa IGraphSaveable hyödynnetään kaikkien solmujen tallentamisessa ja lataamisessa. Sen implementoivat kaikki pelinaikaisessa koodissa käytetyt luokat, eli solmujen implementaatiot. Rajapinta vaatii, että luokat sisältävät GUID-muuttujan, solmun sijainnin 2D-tasolla ja funktion, joka muodostaa yhteyden implementaatioiden välillä ennen pelinaikaisen koodin suoritusta. (Koodiesimerkki 26.)

```
public interface IGraphSaveable
{
    string GUID { get; set; }
    Vector2_NonUnity GraphPosition { get; set; }

    string GetEditorName();

    void CreateDataConnection(
        IGraphSaveable node,
        int portIndex,
        int direction);
}
```

Koodiesimerkki 26. IGraphSaveable-rajapinta

Seuraavassa koodiesimerkissä luodaan yhteydet toisiin objekteihin AIOptionData-luokan asiayhteydessä. Funktio tunnistaa parametrina vastaanotetun objektin tyypin ja asettaa objektin viittauksen asiaan kuuluvaan tietorakenteeseen. (Koodiesimerkki 27.) Jokaisen solmutyypin CreateDataConnection-funktio on erityinen, joten

```
public void CreateDataConnection(
    IGraphSaveable node,
    int portIndex,
    int direction)
{
    if (node is IGraphFloatProvider)
    {
        IGraphFloatProvider _floatProvider = (IGraphFloatProvider)node;

        if (InputValueWrapper.FloatProviders.
            Contains(_floatProvider) == false)
        {
            InputValueWrapper.FloatProviders.Add(_floatProvider);
        }
    }
    else if (node is AIActionBase)
    {
        AIActionBase _action = (AIActionBase)node;

        if (Actions.Contains(_action) == false)
        {
            Actions.Add(_action);
        }
    }
}
```

Koodiesimerkki 27. Implementaatioluokkien yhdistäminen luokassa AIOptionData

5.5.2 Tallentaminen

Tekoäly tallennetaan levylle Unity-moottorin ScriptableObject-luokan .asset-tiedostomuotona. AIGraphSaveUtility aloittaa tallentamisen keräämällä kaikki työkaluikkunan solmut ja niiden väliset yhteydet listoihin muistiin. AIGraphRuntimeContainer-objektiin tallennetaan vain solmujen implementaatiot ja yhteydet. Implementaatioluokat perivät myös Unity-moottorin ScriptableObject-luokan, joten ne voidaan tallentaa ydinobjektin alle .asset-tiedostomuodossa funktion UnityEditor.AssetDatabase.AddObjectToAsset avulla. Seuraava koodiesimerkki muuttaa solmujen väliset yhteydet tallennettavaan muotoon. (Koodiesimerkki 28.)

```
        for (int i = 0; i < m_edges.Count; i++)
        {
            Node _outputNode = m_edges[i].output.node;
            Node _inputNode = m_edges[i].input.node;

            int _portIndex_From = m_edges[i].output.node.
                outputContainer.IndexOf(m_edges[i].output);

            int _portIndex_To = m_edges[i].input.node.
                inputContainer.IndexOf(m_edges[i].input);

            AIGraphNodeBase _nodeBase_Out = (AIGraphNodeBase)_outputNode;
            AIGraphNodeBase _nodeBase_In = (AIGraphNodeBase)_inputNode;

            m_targetContainer.NodeLinks.Add(new NodeLinkData
            {
                GUID_NodeFrom =
                    (_nodeBase_Out.Implementation as IGraphSaveable).GUID,

                PortName = m_edges[i].output.portName,

                GUID_NodeTo =
                    (_nodeBase_In.Implementation as IGraphSaveable).GUID,

                PortIndex_From = _portIndex_From,
                PortIndex_To = _portIndex_To
            });
        }
```

Koodiesimerkki 28. NodeLinkData-luokkien muodostaminen tallennusoperaation aikana luokassa AIGraphSaveUtility

NodeLinkData-luokka kuvaa yhtä solmujen välistä yhteyttä AIGraphRuntimeContainer-objektissa. Se sisältää solmujen GUID-muuttujat sekä porttien osoitenumerot. Näiden muuttujien avulla voidaan pelin aikana yhdistää implementaatiot toisiinsa ja ladata editorityökalussa aikaisemmin tallennettu objekti solmunäkymään. Solmujen implementaatiot kopioidaan yksinkertaisesti AIGraphRuntimeContainer-luokan sisältämiin listoihin.

Tallennusoperaation viimeinen vaihe hyödyntää UnityEditor-rajapinnan funktioita. AIGraphRuntimeContainer-objekti merkitään funktiolla SetDirty. Sen avulla AssetDatabase.SaveAssets-funktio tunnistaa objektiin tehdyt muutokset ja tallentaa ne levylle. AssetDatabase.Refresh-funktio pakottaa Unity-editorin tarkistamaan uudet tai muokatut tiedostot ja päivittää lopuksi editorinäkymän. (Koodiesimerkki 29.)

```
            EditorUtility.SetDirty(m_targetContainer);
            AssetDatabase.SaveAssets();
            AssetDatabase.Refresh();
```

Koodiesimerkki 29. AIGraphRuntimeContainer-objektin muutoksien tallentaminen levylle luokassa AIGraphSaveUtility

5.5.3 Lataaminen

Lataustoiminto suoritetaan, jos käyttäjä valitsee uuden AIGraphRuntimeContainer-objektin ikkunan työkalupalkista. Lataus voidaan suorittaa myös Load-nimisen painikkeen kautta. Ensin senhetkisen ikkunan sisältö tyhjennetään, eli kaikki solmut ja yhteydet poistetaan. Sen jälkeen luodaan uudet solmut tallennetun objektin perusteella. Seuraava koodiesimerkki esittää uusien solmujen luonnin AIOptionData-luokan tilanteessa. (Koodiesimerkki 30.) Samankaltainen toimenpide suoritetaan jokaiselle objektiin tallennetulle solmutyypille.

```
    for (int i = 0; i < m_targetContainer.Options.Count; i++)
    {
        if (m_targetContainer.Options[i] == null)
        {
            UnityEngine.Debug.Log(
                $"m_targetContainer.Options[{i}] == null");
            continue;
        }

        m_targetGraphView.CreateNode(
            typeof(AIGraphOptionNode),
            typeof(AIOptionData),
            m_targetContainer.Options[i],
            true);
    }
```

Koodiesimerkki 30. Uusien solmujen luonti levylle tallennettujen AIOptionData-instanssien perusteella luokassa AIGraphSaveUtility

Viimeiseksi luodaan visuaaliset elementit jokaiselle solmun väliselle yhteydelle. Objektiin tallennetut NodeLinkData-luokat iteroidaan läpi. Ennen yhteyden luomista etsitään oikeat solmut ja portit, jonka jälkeen luodaan visuaalinen elementti. Seuraava koodiesimerkki esittää funktion, joka käy kaikki solmujen väliset yhteydet läpi, valitsee asiaan kuuluvat solmut tallennetun tiedon perusteella ja yhdistää oikeat portit toisiinsa. (Koodiesimerkki 31.)

```
    private void connectNodes()
    {
        for (int i = 0; i < m_targetContainer.NodeLinks.Count; i++)
        {
            Node _nodeFrom =
                findNode(m_targetContainer.NodeLinks[i].GUID_NodeFrom);

            Node _nodeTo =
                findNode(m_targetContainer.NodeLinks[i].GUID_NodeTo);

            if (_nodeFrom == null)
            {
                UnityEngine.Debug.Log("Node From == null");
                continue;
            }
            if (_nodeTo == null)
            {
                UnityEngine.Debug.Log("Node To == null");
                continue;
            }

            Port _portFrom = (Port)_nodeFrom.
                outputContainer[m_targetContainer.NodeLinks[i].PortIndex_From];

            Port _portTo = (Port)_nodeTo.
                inputContainer[m_targetContainer.NodeLinks[i].PortIndex_To];

            generateEdge(
                _portFrom,
                _portTo,
                m_targetContainer.NodeLinks[i].PortName);
        }
    }
```

Koodiesimerkki 31. Funktio connectNodes luokassa AIGraphSaveUtility. Funktio findNode etsii oikean solmun kaikkien solmujen joukosta GUID-muuttujan avulla.

Funktio generateEdges yhdistää kaksi valittua porttia toisiinsa. Luokka Edge sisältyy UIElements-ympäristöön ja se toimii solmujen välisen yhteyden visuaalisena elementtinä. (Koodiesimerkki 32.)

```
    private void generateEdge(Port outputPort, Port inputPort, string name)
    {
        Edge _newEdge = new Edge
        {
            output = outputPort,
            input = inputPort,
            name = name
        };

        _newEdge.input.Connect(_newEdge);
        _newEdge.output.Connect(_newEdge);

        m_targetGraphView.AddElement(_newEdge);
    }
```

Koodiesimerkki 32. Funktio generateEdge luokassa AIGraphSaveUtility

6 Lopputuloksen arviointi ja jatkokehitys

Työn aikana toteutettu tekoäly ja sen työkalu toimivat suunnitelmien mukaisesti. Tekoälyn pelinaikainen koodi on hyvin suorituskykyinen, eikä se aiheuta pelin suorituksen aikana ollenkaan C#-roskaa. Testasin tekoälyä kymmenillä yksilöillä yhtä aikaa ja profiloin suorituskykyä Unity-moottorin tarjoamilla työkaluilla. Jos yksilön tekoälyn päivitystahdin rajoittaa muutamiin kertoihin sekunnissa, niin mobiililaitteillakin voidaan käyttää tarvittaessa suurta määrää aktiivisia tekoälyjä.

Toiminto- ja arvojen palautusluokkia on helppo lisätä abstraktien pohjaluokkien avulla ja ne tunnistetaan automaattisesti työkalun käyttöliittymässä. AIRule-tyyppiset luokat ovat tällä hetkellä täysin erillisiä luokkia, eikä niille luotu pohjaluokkaa. Tämän takia AIGraphRuntimeContainer-objektiin täytyy luoda niille erilliset tallennusosiot. AIRule-pohjaluokan luominen olisi siksi seuraava tärkeä parannus pelinaikaiseen koodiin. Olen kokonaisuutena tyytyväinen pelinaikaisen koodin lopputulokseen, ja uskon, että työn lopputulosta voidaan hyödyntää varsinaisissa peliprojekteissa.

Editorityökalu ajaa asiansa, enkä törmännyt odottamattomiin ongelmiin kehitystyön viimeisen vaiheen jälkeen. Uskon, että sen käyttöliittymä ja tekoälyn rakenne ovat luettavissa kenelle tahansa. UIElements-rajapinta oli koodin kirjoittamisen aikana kokeellinen ja keskeneräinen rajapinta, eikä internetistä löytynyt kovin paljon lähdemateriaalia. En myöskään hyödyntänyt työkalun rakentamisessa lähes ollenkaan USS- tai UXML-tiedostotyyppejä, jotka ovat yksi UIElements-rajapinnan tärkeimmistä ominaisuuksista. Onnistuin kuitenkin saavuttamaan lopputuloksen, joka vastaa visuaalisesti alkuperäistä suunnitelmaa.

Isoin heikkous työkalun nykyisessä tilassa on, että tekoälyn päätöksentekoa ei voi vielä helposti visualisoida testausta ja viankorjausta varten. Jatkokehityksen tärkein tavoite olisi jotain kautta ilmaista reaaliaikaisesti valitun yksilön jokaisen mahdollisen valinnan hyötyarvo laskevassa järjestyksessä. Omassa testiprojektissani loin tekoälylle yksinkertaisen UI-elementin, joka piirretään pelimaailmassa tekoälyn yksilön yläpuolelle. Se ilmaisee tekstillä nykyisen valitun toiminnon ja sen hyötyarvon. Lisäksi hyödynsin runsaasti Unity-editorin konsolia, jonne voi koodista lähettää tekstimuotoisia viestejä. Hyötypohjaisessa tekoälyssä monimuotoinen testaaminen on osa suunnitteluprosessia, joten on tärkeää kehittää tapoja myös sen yksinkertaistamiseen.

Tämän dokumentin luvussa 3.1 esiteltiin kuvaajat, joiden käyttöä työkalussa ei kirjoitushetkellä vielä tueta. AnimationCurve-luokka olisi mahdollista toteuttaa yksinkertaisesti uutena solmutyyppinä, joka ottaa vastaan float-arvon ja palauttaa sen perusteella kuvaajasta uuden float-arvon.

Tekoälyn toteutuksen aikana omaksuin tapoja lähestyä ohjelmointia, jotka aikaisemmin olivat itselläni vähemmän käytössä. Hyödynsin tekoälyssä paljon abstrakteja luokkia ja rajapintoja, jotka molemmat tarjoavat mahdollisuuksia käyttää samaa koodia uudelleen eri tilanteissa. Työn koodi täytyi samalla jakaa itsenäisiin toisista riippumattomiin osa-alueisiin. Unity-moottorissa on erittäin tärkeää erotella editorissa ja pelissä suoritettava koodi, koska editor-koodia ei pakata mukaan koottuun pelisovellukseen (Unity Technologies 2020). Koodin erittely pienempiin suljettuihin kokonaisuuksiin helpottaa myös testaamisessa ja jatkokehityksessä.

Tekoälyn kehityksessä ja testaamissa hyödynsin henkilönkohtaista toimintapeliprojektia. Pelissä pelaaja voi pysyä tekoälyltä näkymättömissä tai taistella sitä vastaan tuliaseiden avulla. Tekoälyltä vaaditaan siis hyvin paljon erilaisia toimintoja ja aisteja. Sen täytyy myös välittömästi siirtyä toteuttamaan taistelutoimintoja, kun pelaaja havaitaan. Tekoäly voi myös tutkia pelaajan aiheuttamia ärsykkeitä, kuten ääniä.

Suurimman osan työn kehityksen ajasta lisäsin tekoälyn uudet toiminnot samaan AIGraphRuntimeContainer-objektiin. Käyttöliittymä alkoi täyttyä solmuista, ja halutun toiminnon muokkaaminen ja testaaminen hidastui. Päätin jakaa tekoälyn toiminnan kolmeen eri tilaan: rutiinitila, tutkimistila ja taistelutila. Kaikki tilat ovat erillisiä AIGraphRuntimeContainer-objekteja, jotka sisältävät pelkästään niissä toteutettavat toiminnot. Nykyisen tilan asettaa erillinen AIGraphRuntimeContainer, joka hyödyntää samaa päätöksentekoa. Sovelsin siis kerrostetun tilakoneen rakennetta hyötypohjaisessa päätöksenteossa. AIComponent-luokassa oli työn lopussa kaksi AIGraphRuntimeContainer-muuttujaa: StateGraph ja ActionGraph. Nykyinen tila laskettiin ensin ja toiminto sen jälkeen. Tämä ratkaisu osoittautui hyväksi tasapainoksi muokkaamisen, testaamisen ja hyötyarvojen määrittämisen helppouden kannalta. Tilat sisältävät vain 4-8 mahdollista toimintoa. Niiden säännöt ovat yksinkertaisia, jonka takia myös testaaminen on nopeampaa.

Rakensin projektiin usean tekoälyn ohjaamista varten myös luokan nimeltä AICoordinator. Sen avulla yksilöt voivat kommunikoida keskenään ja toteuttaa tiettyjä toimintoja yhdessä. Tekoälyt voivat muodostaa pareja, kun aistiärsykettä lähdetään tutkimaan ja taistelun aikana voidaan rajoittaa hyökkäystoimintojen suorittamista. Projektissa vain yksi tekoäly voi ampua aseella kerrallaan ja kaksi voi jahdata pelaajaa lähitaisteluaseiden, kuten veitsien kanssa. Tekoäly rekisteröi it-

sensä AICoordinator-luokalle toimintojen OnActionBegin ja OnActionEnd avulla. Tämä järjestelmä on yksinkertainen, mutta tekee taisteluista mielenkiintoisempia pelata, vaikka pelaaja on aina paljon vahvemmassa asemassa. AICoordinator-tyyppistä luokkaa voisi hyödyntää paljon enemmän peleissä, jossa tekoälyjen ryhmätyö on oleellinen osa niiden käytöstä. Koordinaattori voisi toiminnassaan hyödyntää hyötypohjaista päätöksentekoa tai painotettua valintaa, esimerkiksi antamalla yksilöille määräyksiä hyökätä tai liikkua tiettyihin pisteisiin. Mielestäni hyötypohjainen päätöksenteko sopii tällaisiin tilanteisiin mainiosti.

Lähteet

Bahnassi, W. (2019). *The science behind the UIElements Renderer*. Videoluento.
Viitattu 07.04.2020:
https://youtu.be/zeCdVmfGUN0
Luennossa käytetyt kalvot saatavilla 07.04.2020:
https://www.slideshare.net/unity3d/built-for-performance-the-uielements-renderer-unite-copenhagen-2019

Bourg M. & Seemann G. (2004). *AI for Game Developers*. O'Reilly Media. Sebastopol, CA.

Campeanu D. (2019). *What's new with UIElements in 2019.1*. Blogikirjoitus.
Viitattu 21.03.2020 osoitteesta:
https://blogs.unity3d.com/2019/04/23/whats-new-with-uielements-in-2019-1/

Champandard A. & Dunstan P. (2013). *The Behavior Tree Starter Kit* -kappale*, Game AI Pro.*
CRC Press. Boca Raton, FL.
Viitattu 21.03.2020. http://www.gameaipro.com/

Chotrani, A & Jin D. (2018). *What is GOAP, and why is it not already mainstream.* Tutkimus.
Viitattu 19.05.2020:
https://www.slideshare.net/AakashChotrani/what-is-goap-and-why-is-it-not-already-mainstream

Colledanchise M. & Ögren P. (2018). *Behavior Trees in Robotics and AI: An Introduction*.
Saatavilla 17.04.2020 https://arxiv.org/abs/1709.00084

Dawe M., Gargolinski S., Dicken L., Humphreys T. & Mark D. (2013).
Behavior Selection Algorithms -kappale*, Game AI Pro.* CRC Press. Boca Raton, FL.
Viitattu 21.03.2020. http://www.gameaipro.com/

Graham D. (2013). *An Introduction to Utility Theory* -kappale*, Game AI Pro.*
CRC Press. Boca Raton, FL.
Viitattu 21.03.2020. http://www.gameaipro.com/

Hewes J. (2020) *Logic Gates.* Internet-artikkeli piirilevyjen logiikkaporteista.
Viitattu 19.05.2020. https://electronicsclub.info/gates.htm

Kirby N. (2010). *Introduction to Game AI.* Cengage Learning. Boston, MA.

Mark D. (2009). *Behavioral Mathematics for Game AI.* Cengage Learning. Boston, MA.

Millington, I. (2019). *AI for Games*. CRC Press. Boca Raton, FL.

Unity Technologies. Unity-pelimoottorin UIElements-työkalun käyttöopas.
Viitattu 21.03.2020 osoitteesta:
https://docs.unity3d.com/Manual/UIElements.html

Unity Technologies. Unity-pelimoottorin C# ohjelmointirajapinnan dokumentaatio. Viitattu 21.03.2020 osoitteesta:
https://docs.unity3d.com/ScriptReference/

Zubek, R. (2010). *Needs-Based AI* -kappale, *Game Programming Gems 8.* (editointi: Lake A.) Cengage Learning. Boston, MA.