Antti Leinonen

KONEOPPIMISEN HYÖDYNTÄMINEN PELIKEHITYKSESSÄ

Opinnäytetyö
Tietojenkäsittely

2020

Kaakkois-Suomen ammattikorkeakoulu

Tekijä/Tekijät	Tutkinto	Aika
Antti Leinonen	Tradenomi (AMK)	Toukokuu 2020
Opinnäytetyön nimi Koneoppimisen hyödyntäminen pelikehityksessä		29 sivua 0 liitesivua
Toimeksiantaja -		
Ohjaaja Jukka Selin		

Tiivistelmä

Koneoppiminen on viimevuosina noussut yhdeksi kysytyimmistä osa-alueista tietotekniikan parissa. Opinnäytetyössäni käsittelen mahdollisuuksia pelikehityksen tiimoilta. Työn tavoite oli selvittää, onko Unityn Machine Learning Agents -paketilla järkevää rakentaa strategiapainiotteisen RPG-pelin tekoäly.

Työn teoriaosuudessa kerron pelikehityksen perusteista sekä sen parissa tarvittavia käsitteitä ja konsepteja. Käsittelen tarkemmin AI:n osuutta peleissä, mitä järjestelmiä se vaatii vuorovaikutukseen pelimaailman kanssa sekä millaisia menetelmiä ja konsepteja sen luomisessa normaalisti käytetään.

Käytännönosuudessa selostan vaiheita, jotka koneoppimisen ympäristön asentaminen, sekä sen implementointi Unityllä luotuun peliin vaati. Testien yhteydessä pohdin, mitä hyötyjä ja haittoja koneoppimisesta on kyseistä peliä varten sekä minkä tyyppisiin peleihin koneoppiminen voisi parhaiten soveltua.

Työn tuloksena on toimiva koneoppimisympäristö, jossa python ympäristössä ajettava Machine Learning Agents voi vuorovaikuttaa aiemmin Unitylla luomamme pelin kanssa. Ympäristön avulla koulutusta voisi jatkaa pelikehityksen edetessä. Testailun perusteella kävi kuitenkin ilmi, että muutaman tapaustamme varten kriittisen rajoituksen sekä teknisen epävakauden takia paketin hyödyntäminen kyseisessä projektissa ei ole järkevää.

Koneoppiminen voi vielä mullistaa pelialan ja nykyiseltäänkin se soveltuu tiettyihin tapauksiin todella hyvin. On kuitenkin vielä matkaa ennen kuin kaikki peleissä tarvittavat tekoälytapaukset on järkevää kouluttaa koneoppimisella, eikä luoda perinteisin menetelmin.

Asiasanat

koneoppiminen, pelikehitys, tekoäly

South-Eastern Finland University of Applied Sciences

Author (authors)	Degree	Time
Antti Leinonen	Bachelor of Business Administration	May 2020
Thesis title		
Using machine learning in game development		29 pages 0 pages of appendices
Commissioned by		
-		
Supervisor		
Jukka Selin		

Abstract

Machine learning has become one of the big things in the software field. This thesis examined the possibilities machine learning would bring in terms of game development. Main goal of this thesis was to figure out if it would be worthwhile to develop the AI of a strategy-RPG game using the Machine Learning Agents toolkit developed by Unity.

The theory part of this thesis touched upon the basics of game development and the related concepts and terms. The study also discussed the importance of AI in games and what kind of systems they require to interact with the game world. Additionally, I examined the methods by which AI for games has normally been created.

The practical part went through the steps required to setup a Python-based machine learning environment and how to get it to interact with a game world created with Unity. I also attempted to train an agent to play the RPG game this thesis was based on. Based on these tests I examined what benefits and challenges machine learning brings to our specific case. I also contemplated what sort of games the Machine Learning Kit would suit the best.

The result was a functioning test environment that could be used to further train the AI when the development of the game continues. However due to certain limitations of the Kit and a few technical issues it would not be worth using in this project.

Machine learning could be becoming the next big thing within game development and even now it's a useful tool for many scenarios. There is still a way to go before it becomes the go-to solution for every type of game AI development case.

Keywords

machine learning, game development, artificial intelligence

SISÄLLYS

1 JOHDANTO

Koneoppiminen on yksi nykyajan tärkeimmistä aiheista tietotekniikan saralla. Sen luomia tietoteknisiä mahdollisuuksia yritetään hyödyntää lähes jokaisella alalla, lääketieteestä autoteollisuuteen. Koneoppiminen mahdollistaa monia lähes tieteisfiktiomaisia ratkaisuja ongelmiin, jotka ennen vaativat valtavia määriä koodia tai olivat lähes ratkaisemattomissa. Mustavalkokuvia pystytään värittämään erinomaisin lopputuloksin (Djudjic 2019). Sydänsairauksia voidaan ennustaa potilastietojen pohjalta (Kukara 1999). Autot osaavat pian ajaa itse koneoppimisen avulla (Eady 2019). Ei olekaan mikään ihme, että myös pelikehityksessä on alettu tutkia koneoppimisen luomia mahdollisuuksia.

Avoin esimerkki koneoppimiskokeiluista pelialalla on pelimoottorikehittäjä Unityn vielä työnalla oleva työkalu ”Unity Machine Learning Agents Toolkit”. Tämän työkalun avulla voidaan yhdistää TensorFlow (Googlen kehittämä tekoälykirjasto) -pohjainen koneoppiminen Unityllä luotuihin pelimaailmoihin. Tämä mahdollistaa koneoppimisella koulutettujen tekoälyjen käytön pelien vihollisina, kumppaneina ja muina pelimaailmassa vaikuttavina voimina. (Juliani ym. 2018.)

Tämän opinnäytetyön tavoitteena on selvittää, kuinka käytännöllistä on kehittää vuoropohjaisen RPG-strategiapelin tekoäly koneoppimisen avulla. Toimeksianto tulee kirjoittajan omasta peliprojektista, jota varten tekniikkaa on tarkoitus testata. Samalla pyrin myös selvittämään, mitä hyötyjä ja haittoja tästä tekniikasta on ja minkälaisiin peleihin kyseinen tekniikka sopii.

2 PELIMOOTTORI

Pelimoottori on pohja alusta, jolla pelit rakennetaan. Se pitää yleensä sisällään peleissä tarvittavat fysiikat, valaistukset, renderöintimoottorin sekä ääni- ja animaatiojärjestelmät. Pelimoottori pitää huolen myös muistin ja muiden resurssien käytöstä. Pelimoottorin mukana tulee yleensä myös graafinen käyttöliittymä ja useita työkaluja, joilla pelejä voidaan rakentaa. (Ward, J. 2008.)

2.1 Nykyajan pelimoottorit

Ennen lähes jokaiselle pelille pystyttiin rakentamaan oma moottori, mutta nykyään pelit ovat laajentuneet, käytettävät fysiikat sekä törmäyslaskennat ovat monimutkaistuneet ja yleiset graafiset vaatimukset peleille ovat kasvaneet. Pitkälle kehitetyt pelimoottorit mahdollistavat myös helpomman lähestymisen pelikehitykseen vasta-alkajille, sillä komplekseihin taustajärjestelmiin ei tarvitse keskittyä ollenkaan. (Halpern 2019.)

Tänä päivänä isommatkin yritykset käyttävät samaa pelimoottoria todella pitkään ja yrittävät vain päivittää sen ominaisuuksia mahdollisuuksien mukaan, koska modernin pelimoottorin tekeminen vie todella pitkään (Ramsay 2018). Suosituimpia vapaasti saatavilla olevia pelimoottoreita ovat Unity, Unreal ja CryEngine. Kaikki edellä mainitut moottorit ovat kilpailukykyisiä isojen pelistudioiden moottoreiden kanssa, ja ne ovat harrastelijoille sekä pienille studioille ilmaisia.

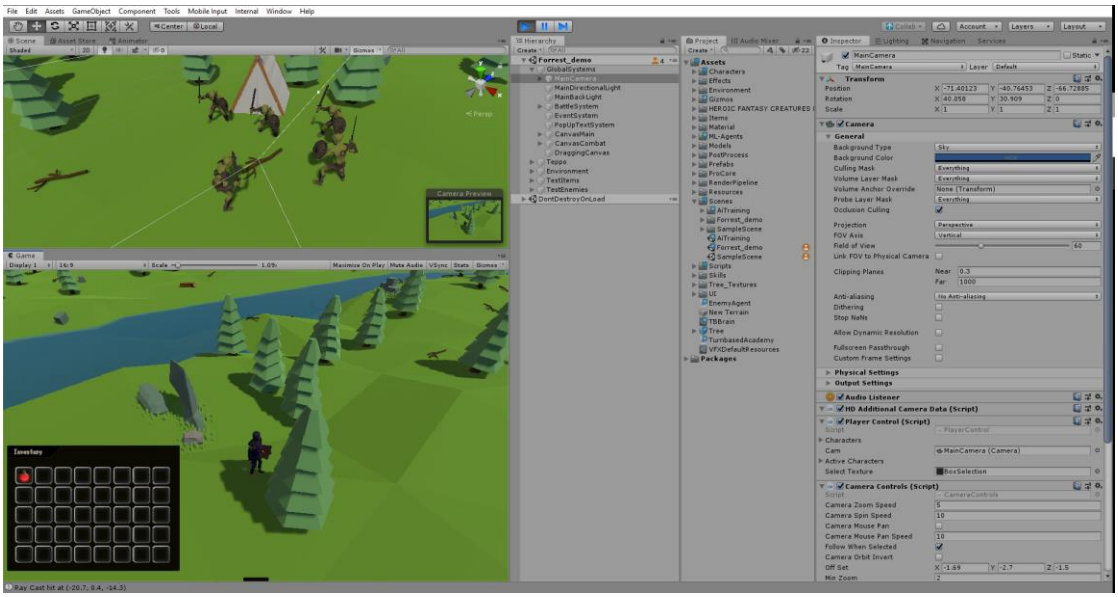
Pelimoottorin valinta on tärkeä, sillä vaikka moottorit ovat kustomoitavissa, tiettyjä ominaisuuksia kuten valaistusta ja fysiikoita voi muokata vain rajoitetusti. Moottorin ominaisuudet vaikuttavat myös siihen, minkä tyylisiä pelejä sillä kannattaa rakentaa ja kuinka raskaita valmiista peleistä tulee. Pelimoottorin valinta päättää myös mitä ohjelmointikieltä käytetään, millä alustoilla peli toimii ja mitä peleissä tarvittavia mekaniikkoja löytyy suoraan pelimoottorista ja mitä joudutaan ohjelmoimaan peliä varten. Tätä opinnäytetyötä varten käytämme Unityä koska se on ainut moottori, jolle käyttämämme koneoppimisympäristö on tehty suoraan yhteensopivaksi. Unity on myös pelimoottori, jolla RPG-peli, jota varten tätä tutkimusta teen, on rakennettu.

2.2 Unity

Unity on järjestelmäriippumaton pelimoottori, joka on yksityishenkilöille ja pienille studioille ilmainen. Unity on aiemmin ollut varteenotettava vaihtoehto lähinnä pienille projekteille, koska sen tuottamat pelit eivät olleet visuaalisesti yhtä näyttäviä kuin muilla moottoreilla, esim. Unreal Engine, toteutetut pelit. Unity on viimeisen muutaman vuoden aikana kuitenkin ottanut isoja harppauksia ollakseen oikeasti kilpailukykyinen suurempien pelitalojen moottoreiden kanssa, myös visuaalisesti.

Visuaalisuutta on parannettu uudella Post-Prosessing -järjestelmällä (Post-processing overview 2019). Uusi avoin Render Pipeline järjestelmä mahdollistaa kokonaan oman renderöinti järjestelmän tekemisen tai valmiiksi tehtyjen järjestelmien muokkaamisen (Lagarde 2018). Myös tehokkaan koodin tekeminen on tehty paljon helpommaksi uusilla ECS- ja Jobs-järjestelmillä, jotka mahdollistavat tietokoneiden ytimien käyttämisen tehokkaammin.

Unityn suurin vahvuus on kuitenkin tukimateriaali, jota sitä varten on toteutettu. Unityn omasta Asset Storesta löytyy paljon ilmaista- ja maksulista materiaalia, jonka seasta löytyy teksturoituja malleja, hyödyllisiä scriptejä ja jopa valmiita pohjia peleille. Tutoriaaleja ja muita apuvideoita löytyy paljon YouTubesta ja Unityn oma dokumentaatio on yleisesti todella hyvä.



Kuva 1. Unity pelimoottorin käyttöliittymä

Tämän takia Unity on aloittavalle peli kehittäjälle selkeä valinta ja edellä mainittujen parannuksien takia toimii myös suuremmissa projekteissa kokeneiden kehittäjien käsissä nykyään melko hyvin. Vastaavaa koneoppimispakettia ei muilta pelimoottoreilta löydy, joten Unity on selkeä valinta tätä opinnäytetyötä varten.

3 PELITEKOÄLY

Tekoäly eli AI on periaatteessa mikä tahansa informaation pohjalta päätöksiä tekevä ohjelma. Yleensä sillä kuitenkin tarkoitetaan monimutkaisempaa kokonaisuutta, joka ottaa huomioon monia eri muuttujia päätöksiä tehdessä. Tekoälyn määritelmä on todella laaja, ja sille on monia eri tulkintoja, koska älykkyyttä itsessään on vaikea määritellä.

Pelien yhteydessä tekoälyn määritelmä on selkeämpi. Se tarkoittaa lähes aina NPC-hahmoja eli hahmoja, joita pelaaja ei kontrolloi sekä ohjailevaa koodia, joka reagoi pelaajan liikkeisiin.

3.1 Tekoälyn rooli peleissä

Barreran ym. mukaan (2018, 7) tekoälyn rooli on tehdä peleistä hauskoja luomalla kiinnostavia vastavoimia, joiden kanssa pelaaja voi kilpailla sekä NPC-hahmoja, jotka käyttäytyvät realistisesti pelimaailman sääntöjen sisällä. Oman pelaamiskokemukseni mukaan hyvä tekoäly luo maailmasta elävän ja immersiivisen, kun taas huono tekoäly voi pilata kokemuksen kokonaan, tai vähintään vetää pelaajan hetkellisesti pois pelin luomasta maailmasta.

Pelitekoäly poikkeaa muista tekoälytapauksista siten, että päätösten pitää vaikuttaa inhimillisesti mahdollisilta, kun taas monissa muissa tapauksissa tekoälystä halutaan niin erehtymätöntä kuin vain on mahdollista tehdä. Jos yksi vihollinen huomaa pelaajan ja yhtäkkiä kaikki kartan viholliset tietävät missä pelaaja on, tulee pelistä erittäin huono kokemus pelaajalle. Monessa tapauksessa on myös parempi, että tekoälyn ”älykkyys” on rajoitettu, jotta pelaaja voi voittaa sen.

Yleensä pelien tekoälyn päätöksenteko on suhteellisen yksinkertaista, ja tärkeämpää on luoda illuusio älyllisistä olennoista, kuin emuloida älykkyyttä täydellisesti (Barrera ym. 2018, 7). Liiallinen yksinkertaisuus voi kuitenkin johtaa epäinhimilliseen käytökseen. Esimerkiksi RPG-peli Skyrimissa lohikäärme hyökkää kylään ja kaikki tarpeeksi lähellä olevat NPC:t liittyvät taisteluun välittämättä siitä onko heidän aseinaan ainoastaan puulusikat. Tämän voisi korjata ottamalla NPC:n oman varustuksen huomioon taisteluun liittymispäätöstä tehtäessä.

3.2 Pohjajärjestelmät

Ennen AI:n logiikan tekemistä kannattaa miettiä mitä AI:n kontrolloimien hahmojen pitää maailmassa tehdä ja luoda niistä eri järjestelmät, joita päätöksen teko voi kontrolloida. Järjestelmien erottelulla saadaan koodista helpommin luettavaa, korjattavaa ja muokattavaa. Hyvin eroteltua koodia on myös nopeampi rakentaa isommalla tiimillä, sillä eri koodi osuudet voidaan jakaa jäsenten kesken ja niitä voidaan kehittää samanaikaisesti.

Pohjajärjestelmiä suunnitellessa kannattaa myös miettiä, halutaanko että kaikki hahmot toimivat samoilla menetelmillä ja animaatioilla, vai onko pelaajalla esimerkiksi tarkempi kontrolli hahmostaan. Strategiapeleissä halutaan yleensä käyttää lähes täysin samoja mekaniikkoja pelaajan- sekä vihollisen yksiköihin, kun taas monissa muissa peleissä pelaaja liikkuu omilla säännöillään ja vihollisilla voi olla hyvin erilaiset säännöt ja käyttäytymismekaniikat.

Mihin tahansa lopputulokseen päätyy, on silti tärkeää tehdä käytetyistä ratkaisuista aina mahdollisimman universaaleja ja uudelleenkäytettäviä. Tällä ajattelulla vältytään monilta huonoilta ratkaisuilta, jotka voivat aiheuttaa ongelmia myöhemmässä vaiheessa kehitystä.

AI:ta varten tarvitaan lähes aina jokin järjestelmä, joka vastaa hahmojen liikkumisesta. Liikkumisen laskemiseen peleissä käytetään monesti A* -polunlaskenta-algoritmia. Kyseinen algoritmi on paljon käytetty, koska se on nopea ja tarkka. Polunlaskentaa varten kartta jaetaan tasaisiin osiin ja jokaiselle osalle annetaan painoarvo. Painoarvojen pohjalta voidaan laskea halvin, eli nopein reitti kohteen luo, jossa ei ole esteitä. Algoritmin avulla luodun polun perusteella voi AI hahmo liikkua haluamaansa kohteeseen esteitä vältellen.

Liikkumisen lisäksi tarvitaan animaatiojärjestelmä, joka huolehtii hahmojen liikuttamisesta luonnollisen näköisesti. Hyvä animaatiojärjestelmä mahdollistaa myös kommunikoinnin toiseen suuntaan, jos halutaan että tietyssä animaation vaiheessa tapahtuu jokin pelimaailman sisäinen laskenta. Nykyaikaisen 3D-animaatiojärjestelmän tekeminen alusta asti on todella työlästä. Onneksi

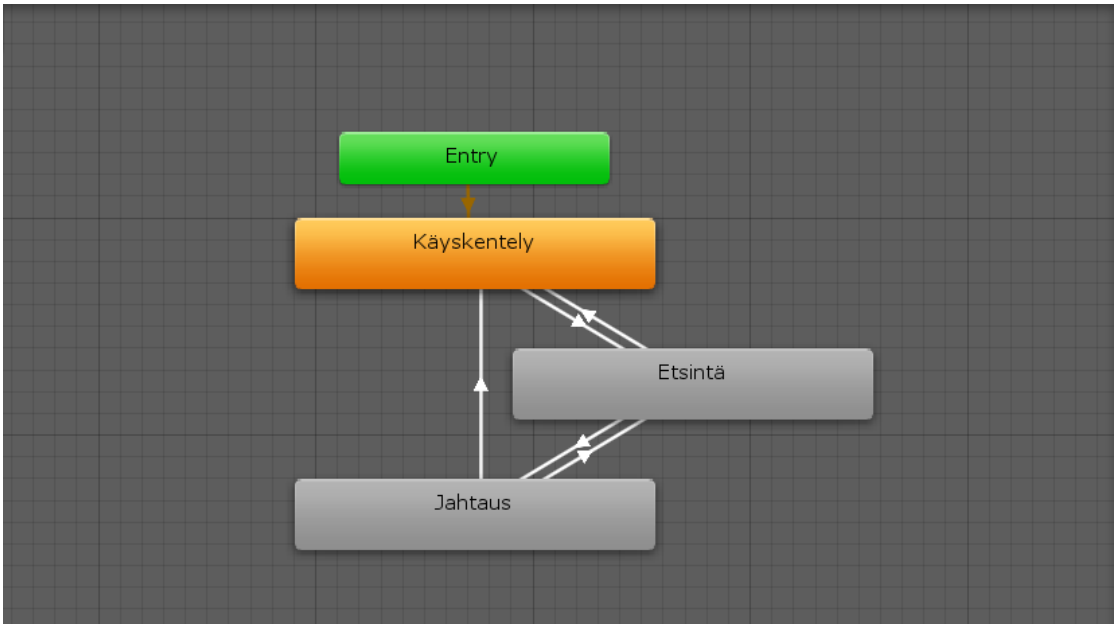
kaikki nykyaikaiset pelimoottorit sisältävät animaatiojärjestelmän valmiina ja esimerkiksi Unityssä se on varsin riittävä.

Monessa pelissä tarvitaan myös järjestelmiä hoitamaan vihollisten aisteja. Yksinkertaisimmillaan tämä tarkoittaa toimintoa, joka tunnistaa, kun pelaaja lähestyy tietyn matkan päähän ja aktivoi esimerkiksi pelaajan kimppuun hyökkäämisen. Monimutkaisemmissa peleissä aistit ovat kokonaisuus eri järjestelmiä, jotka simuloivat ihmisaisteja. Näköaistin voi toteuttaa vihollisen pään mukana liikkuvan kenttänä, joka antaa tiedon pelaaja hahmosta vasta kun pelaaja on oikean matkan päässä, valaistus on tarpeeksi hyvä eikä mikään estä näkyvyyttä hahmoon. Monesti pelien vihollisilla on myös kuuloaisti, joka toimii myös etäisyyspohjaisesti. Kuuloaisti yleensä havaitsee pelaajan, jos pelaaja esimerkiksi juoksee tai hyppii tarpeeksi lähellä kuuntelevaa vihollishahmoa.

3.3 Tilakone

Tilakone (Finite state machine) on matemaattinen malli, jossa koneella on eri tiloja ja se voi olla vain yhdessä tilassa kerrallaan. Siirtyminen tilojen välillä tapahtuu tietyin ennalta määritellyin kriteerein. Kriteereitä tarkastellaan, kun jokin tapahtuma laukaisee tarkastelun. FSM-mallilla voidaan kuvata esimerkiksi liiketoiminnan eri vaiheita. Pelikehitystä varten kiinnostavinta on kuitenkin, kuinka FSM-mallin avulla on mahdollista simuloida vaiheittaista logiikkaa.

Esimerkiksi vartija, jonka ohi pelaajan pitää päästä, aloittaa käyskentelytilasta. Käyskentelytilassa vartija on rauhallinen ja kävelee kenttää ympäri hitaasti. Jos pelaaja pitää liikaa meteliä tai vilahtaa vartijan näkökentässä, siirtyy vartija etsimistilaan. Etsimistilassa vartija ottaa nopeampia askelia kohti havaintojaan ja katselee ripeästi ympärilleen. Jos pelaaja löytyy etsinnän aikana, siirtyy vartija jahtaamistilaan, jossa se juoksee täyttä vauhtia kohti pelaajaa ja yrittää ottaa tämän kiinni. Jos jahtaamistilassa ei ole tullut näköhavaintoja hetkeen, siirtyy vartija takaisin etsintätilaan, josta viimein takaisin käyskentelytilaan, jos etsintä ei tuota tulosta. Jos jahtauksen aikana pelaaja saadaan kiinni, siirtyy vartija suoraan takaisin käyskentelytilaan.



Kuva 2. Esimerkki yksinkertaisesta Finite State Machine mallista, joka kuvaa vartijan tekoälyä

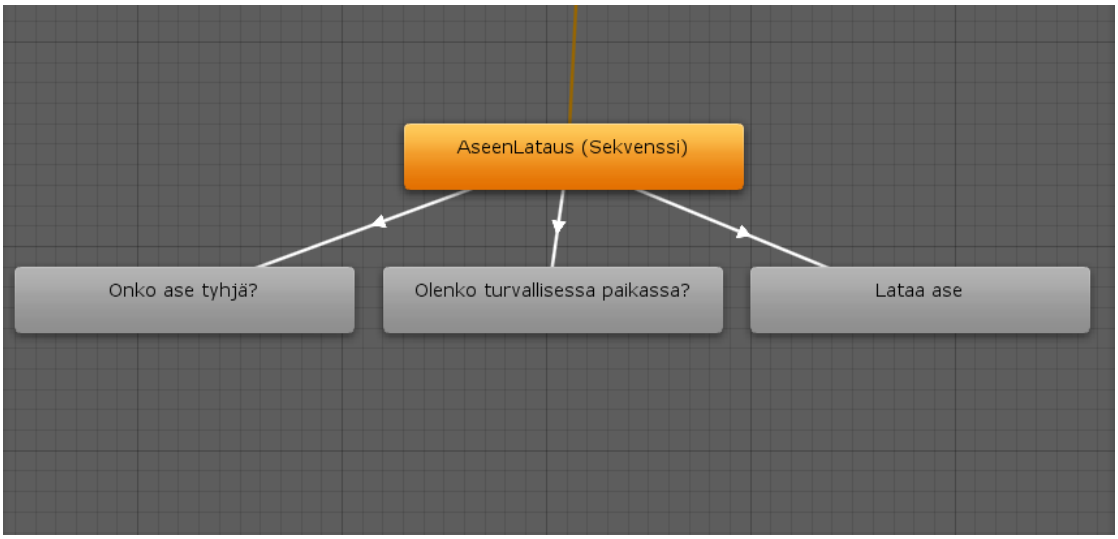
Tämän tyylinen esittämistapa toimii suhteellisen yksinkertaisissa tapauksissa hyvin, mutta jos tiloja on paljon, tulee mallin hallitsemisesta vaikeaa ja työlästä, sillä jokainen siirtymä tilojen välillä pitää määritellä erikseen.

3.4 Käytöspuut

Edellä käsiteltyyn yksinkertaiseen esimerkkiin FSM on helppo ja selkeä ratkaisu, mutta suuremmissa tapauksissa jokaisen eri siirtymän määrittäminen tilojen välille on haasteellista ja aikaa vievää. Näissä tapauksissa käytöspuu (Behaviour Tree) tuo lisää modulaarisuutta ja muokattavuutta logiikan tekemiseen. Käytöspuu ajaa samaa roolia kuin FSM-malli, mutta käsittelee käyttäytymistä puussa sijaitsevien eri tyyppisten solmujen avulla. Puun solmujen ideana on, että jokainen niistä palauttaa aina puunjuurta kohti joko Running, True tai False. Palautetun arvon perusteella solmut johtavat puun aina johonkin tilaan kuten FSM-mallissa. (Barrera ym. 2018, 20.)

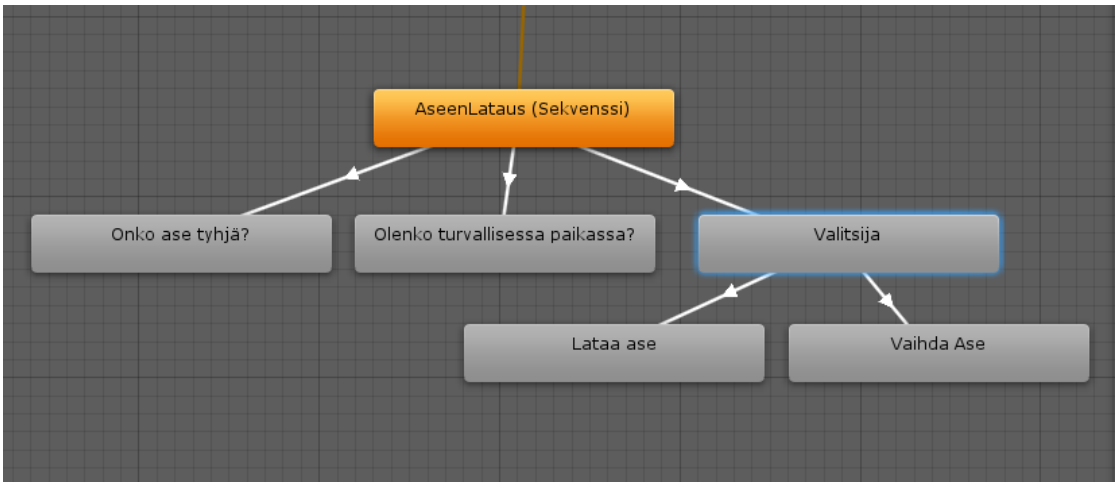
Yhdistelmäsolmut ovat erilaisia solmuja puussa, jotka prosessoivat lapsisolmujaan. Yleisin yhdistelmäsolmu on sekvenssi. Sekvenssillä voidaan esimerkiksi kuvata tarkistuksia, jotka pitää tehdä ennen kuin jokin toiminto voidaan toteuttaa. Jos esimerkin vartijalla olisi ase, jota hän yrittäisi ladata, voitaisiin se kuvata sekvenssisolmulla. Aseen lataus -sekvenssisolmun alla olisi ensin solmu, joka tarkistaa onko ase tyhjä ja sitten solmu, joka tarkistaa onko vartija

tarpeeksi turvallisessa tilanteessa ladatakseen aseen ja lopuksi solmu, joka lataa aseen (kuva 3). Kaikki sekvenssin tarkistukset pitää mennä läpi, ennen kuin vartija aloittaa aseen lataamisen.



Kuva 3. Aseen lataamista kuvaava käytöspuu

Toinen yhdistelmäsolmu on valitsijasolmu, joka sekvenssin lailla käy lapsensa vasemmalta oikealle läpi. Valitsijasolmu palauttaa kuitenkin heti ”true” kun yksikin lapsi onnistuu. Kuvan 3 puuta voi muokata, niin että lataa ase solmun tilalle tulee valitsija, joka yrittää ensimmäisenä ladata asetta. Mikäli aseen lataaminen ei onnistu, siirrytään vaihda ase solmuun.



Kuva 4. Kuvaan 1 lisätty valitsija solmu, joka valitsee aseen lataamiseen ja vaihtamiseen välillä

Muunnossolmu (Decorator node) on solmu, jonka tarkoitus on joko muokata lapsi solmusta tulevaa vastausta, toistaa sitä, tai tuhota se. Muunnossolmulla

voi olla vain yksi lapsisolmu. Yleinen käytetty muunnossolmu on kääntäjä, joka kääntää lapsen positiivisen vastauksen negatiiviseksi.

Lehtisolmut ovat, oksien päässä ja niillä ei voi olla lapsia. Lehtisolmuista kuitenkin koostuu puun toiminnallisuus ja ne ovat ikään kuin Finite State Machine mallin tilat. Kuvan 4 puussa lehtisolmuja ovat kaikki paitsi sekvenssisolmu ja valitsijasolmu.

4 KONEOPPIMINEN

Koneoppiminen on tekoälyn alahaara, jossa tekoälyä ei suoraan koodata vaan rakennetaan ohjelma, joka osaa itse luoda sopivan mallin datan pohjalta. Tämä on hyödyllistä tilanteissa, joissa dataa on saatavilla paljon ja ratkaisun käsin rakentaminen olisi liian monimutkaista. Koneoppimista hyödynnetään paljon esim. kuvantunnistuksessa, koska pikseleitä tunnistavan koodin rakentaminen käsin voi tilanteen mukaan olla todella työlästä.

Tämä toimii peliohjelmoinnin parissa siten, että koneelle syötetään dataa tai kuvia pelin tilanteesta ja samalla koneelle annetaan tietyt muuttujat kontrolloitaviksi. Käytän jatkossa koneelle syötetystä datasta termiä Input ja koneen kontrolloimista muuttujista termiä Output. Kone tarkastelee input- ja output-arvojen välisiä yhteyksiä ja rakentaa niiden pohjalta komplekseja malleja, joiden mukaan AI toimii.

4.1 Machine Learning Agents

Luku pohjautuu Juliani ym. 2018 dokumentaatioon Machine Learning Agents -paketista.

Machine Learning Agents on avoimenlähdekoodin lisäosa Unityyn. Siihen kuulu kehitystyökalu (SDK), joka mahdollistaa koulutusympäristöjen luomisen pelimoottorin sisällä sekä Python-paketti, joka mahdollistaa määritettyjen ympäristöjen kanssa tehtävän vuorovaikutuksen. Koulutuksessa on mahdollista käyttää joko valmiiksi mukana algoritmia tai vaikka luoda oma. (Juliani ym. 2018.)

Machine Learning Agents -pakettiin Unityn sisällä kuuluu kolme pääkomponenttia: Agent, Brain ja Academy. Nämä komponentit hoitavat koulutuksen eri osa-alueita ja jokainen niistä pitää olla implementoitu näkymään, jossa koulutusta halutaan toteuttaa. Brain-komponentti on nimensä mukaisesti aivot, joita skenaarion agentit käyttävät. Siihen määritellään, mitä toimintoja ja havaintoja agentit voivat tehdä. (Juliani ym. 2018.)

Agentti puolestaan kerää havainnot, lähettää ne aivoille ja toteuttaa aivoilta tulleet päätökset. Agentteihin itseensä pitää määritellä miten aivoilta tulleita käskyjä toteutetaan, mitä palkintoja mistäkin toiminnosta tulee ja milloin koulutus episodi pitää aloittaa alusta. Yleensä koulutuksessa käytetään montaa samanaikaista agenttia, jotka kaikki lähettävät havaintoja omasta perspektiivistään aivoille. (Juliani ym. 2018.)

Academy-komponentti huolehtii koulutuksesta yleisesti. Sinne voidaan määrittää, kuinka nopealla aika kertoimella koulutusta pyöritetään. Academy huolehtii myös kommunikoinnista Python API:n kanssa. Academyyn määritellään myös, mitkä aivot toteuttavat koulutusta skenaariossa. (Juliani ym. 2018.)

Jokaista Unityllä koulutettua aivoa varten voidaan säätää monia eri parametrejä siitä, miten koulutus suoritetaan ja mitä eri tekniikoita siinä sovelletaan. Machine Learning Agents hakee asetukset trainer_config.yaml tiedostosta ja käyttää sen default arvoja, jos valitulle aivolle ei ole omia asetuksia. Asetuksia on yli kaksikymmentä, joten käyn tässä niistä läpi vain tärkeimmät.

```yaml
default:
    trainer: ppo
    batch_size: 1024
    beta: 5.0e-3
    buffer_size: 10240
    epsilon: 0.2
    gamma: 0.99
    hidden_units: 128
    lambd: 0.95
    learning_rate: 3.0e-4
    max_steps: 5.0e4
    memory_size: 256
    normalize: false
    num_epoch: 3
    num_layers: 2
    time_horizon: 64
    sequence_length: 128
    summary_freq: 1000
    use_recurrent: false
    use_curiosity: false
    curiosity_strength: 0.01
    curiosity_enc_size: 128
```

Kuva 5. Harjoitus ohjelmaan muokattavissa olevat arvot

Curiosity-niminen asetus tekee aivoista kokeilunhaluisemmat. Tämä auttaa aivoja pääsemään eteenpäin sellaisissa tilanteissa, joissa palkintoa ei tule välittömästi. Otetaan esimerkiksi tilanne, jossa AI:n tehtävänä on etsiä juustoa supermarketista. Mikäli millään läheisellä hyllyllä ei näy juustoa, ei AI saa palkintoa, ja saattaa jäädä pitkäksi aikaa pyörimään kehää samojen hyllyjen ympärille. Curiosity-asetuksen avulla AI pyrkii hyllyväleihin, joissa se ei ole vielä ollut. (Machine Learning Glossary 2020.)

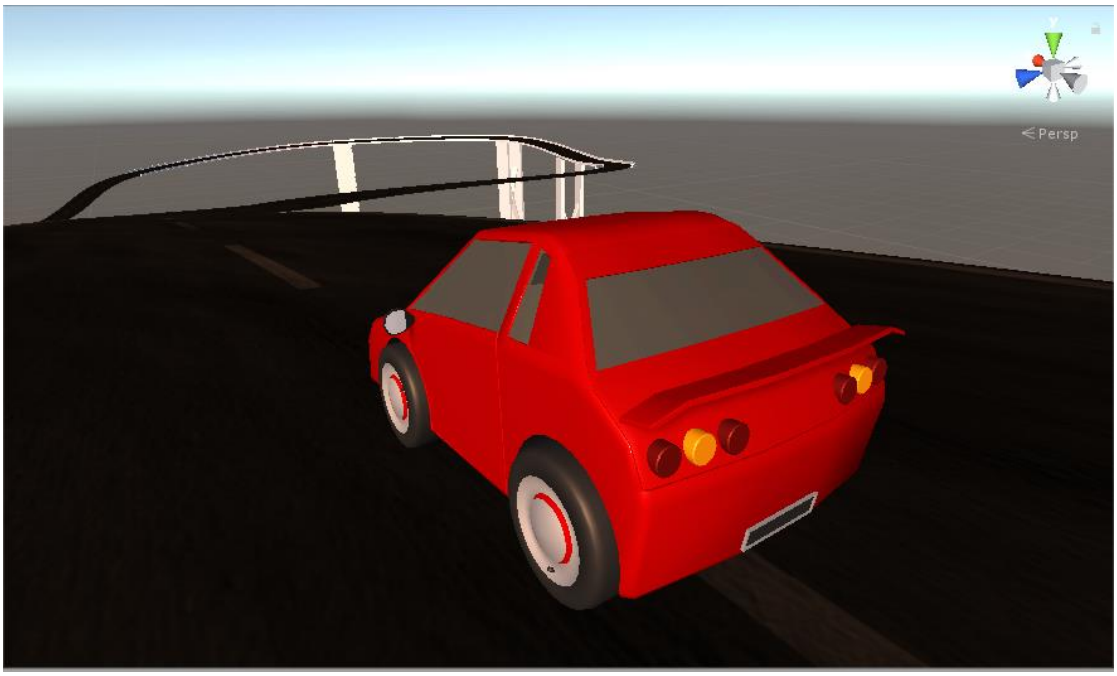
Toinen tärkeä asetus on normalisointi. Asetuksen ollessa päällä agenttien keräämiä observaatio arvoja yritetään normalisoida. Asetus on hyödyksi monimutkaisissa ongelmissa, mutta haitallinen yksinkertaisissa ongelmissa. (Machine Learning Glossary 2020.)

Time horizon -numeroarvo vastaa siitä, kuinka monta kokemusta yksittäinen agentti kerää ennen kuin se vaikuttaa AI:n malliin. Jos arvo ylittyy koulutus episodin aikana, arvioidaan agentin saama palkinto nykytilasta. Pienempi arvo soveltuu tilanteisiin, jossa palkintoja tulee usein tai koulutus kohtaus on todella suuri. (Machine Learning Glossary 2020.)

4.2 Reinforcement Learning ja Imitation Learning

Unityn koneoppimisen yhteydessä eniten käytetty metodi on Reinforcement Learning (vahvistava oppiminen). Tämän metodin perusideana on, että AI:n annetaan kokeilla tilannetta monia kertoja ja jokaisesta suorituksesta annetaan palkinto sen mukaan, miten hyvin AI on suorittanut tilanteen. Tämän perusteella AI osaa painottaa tekemiään ratkaisuja ja valitsemaan kerta kerralta paremmat ratkaisut. Palkintorakenteen hiominen on isoin osa AI:n koulutusta tällä menetelmällä. (Savinov & Lillicrap 2018.)

Toteutin nopean esimerkin ajopelistä havainnollistaakseni tekoälyn koulutusta. Ajopelin AI:n koulutuksessa koneelle annetaan tietoina auton sijainti, nopeus ja 5 seuraavaa pistettä tien keskeltä. Pisteet tulevat tielle luodun NavMeshin kautta, joka on osa Unityn navigointi järjestelmää. Hallittaviksi muuttujiksi kone saa auton kaasun väliltä -1 ja 1, jossa -1 on pakki ja 1 on täysi kaasu. Kone saa hallintaansa myös ohjauksen samaan tapaan.

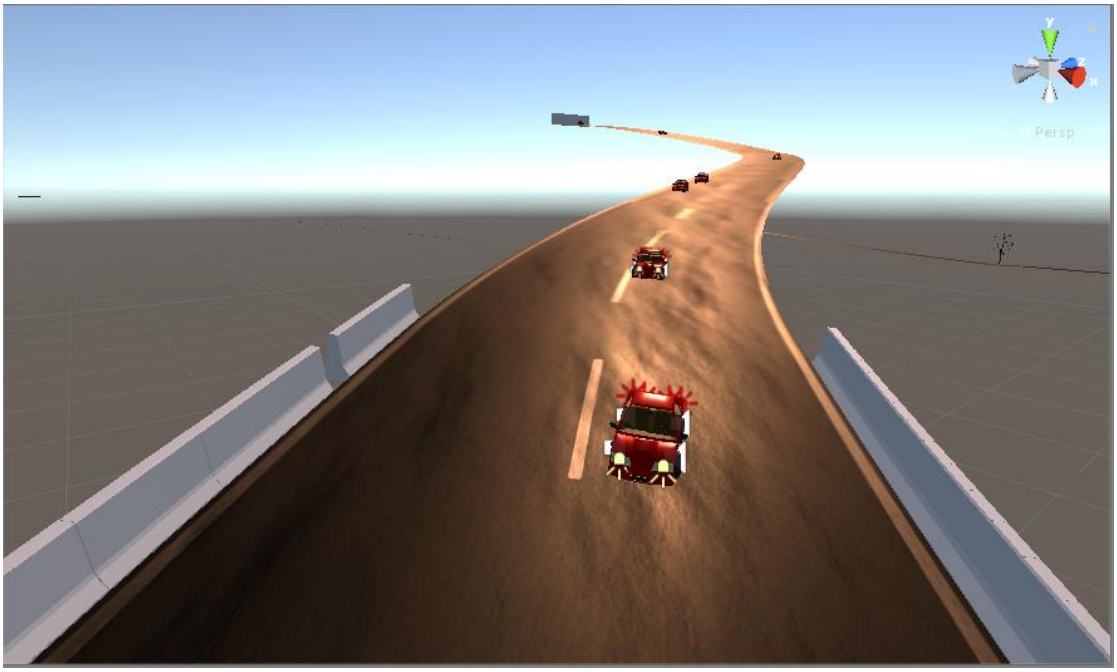


Kuva 6. Testiympäristö, jossa kone oppii ajamaan autoa radan läpi

Auto-esimerkin palkintoon tulee miinusta siitä, jos auto ei pysy tiellä ja sitä isompi palkinto, mitä nopeammassa ajassa auto ajaa radan pisteiden ohi.

Kuva 7. Opetuksen ensimmäiset minuutit, joiden aikana autot hiljalleen liikkuvat radan alku-metrejä läpi

Alussa autot harhailevat hitaasti kokeillen mitä inputeista tapahtuu, mutta melko nopeasti AI saa jonkinlaisen idea siitä, mitä skenaariossa on tehtävä ja autot pääsevät jo ensimmäisen mutkan ohi.



Kuva 8. Opetuksen lopputilanne, jossa kone osaa ohjata autoja sujuvasti radan läpi

Koulutuksen voisi myös toteuttaa ositettuna. Tätä kutsutaan niin sanotuksi koulutusohjelmaksi (Curriculum Training). Tässä metodissa AI:lle luodaan eri skenaariot, joihin se siirtyy saadessaan tarpeeksi ison palkinnon. Koulutus

aloitettaisiin yksinkertaisesta radasta, jossa autoa tarvitsee ajaa vain suoraan. Koulutuksessa AI kokeilee ajaa rataa läpi satoja kertoja pelimoottorin kautta ja selvittää, miten palkinnosta saadaan mahdollisimman iso. Kun kone pääsee tarpeeksi suureen palkintoon suoralla radalla, vaihdetaan rata vähän vaikeampaan, jossa esimerkiksi muutamia mutkia. Mutkat opittuaan voidaan AI:ta kouluttaa vielä samoilla radoilla, joissa pelaajat ajaisivat. AI:n kouluttaminen koulutusohjelman avulla nopeuttaa AI:n koulutusta huomattavasti.

Toinen tapa, jolla AI:ta voi opettaa on käyttää pelaajaa tai muilla tavoilla rakennettua AI:ta esimerkkinä. Tästä tekniikasta käytetään termiä ”Imitation Learning”. Nimensä mukaan tällä tekniikalla AI pyrkii matkimaan valmiina olevaa esimerkkiä halutusta toimintatavasta. Jos esimerkiksi autopelin auto annetaan pelaajan haltuun, voidaan pelaajan suoritukset tallentaa demoina, joista kone pystyy tekemään tarvittavat päätelmät inputtien ja radan kurvien välillä suoraan, ilman tarvetta koulutusohjelmalle.

AI:n koulutus tähän tyyliin on joissain tilanteissa paljon nopeampi ratkaisu. Imitation Learning nopeuttaa varsinkin koulutuksen alkuvaihetta merkittävästi, sillä yleensä AI haparoi melko pitkään ennen kuin sen toiminta liikkuu haluttuun suuntaan. On myös mahdollista aloittaa AI:n koulutus Imitation Learningillä ja jatkaa rakennettua mallia myöhemmin Reinforcement Learningillä. Näin saadaan molempien tekniikoiden hyödyt esiin.

4.3 Käyttö tänä päivänä

Tämä luku perustuu DeepMind (2019) blogiin, jossa AlphaStar tiimi kertoo projektista tarkemmin.

Koneoppimisen implementaatiot ovat vasta kehitteillä pelialan saralla, mutta muutamia esimerkkejä tästä on jo olemassa. Googlen DeepMind on osana tutkimustyötään kehittänyt StarCraft 2 -nimiseen strategiapeliin tekoälyn koneoppimisen avulla. Tekoälyä esiteltiin ensimmäisen kerran Blizzcon -tapahtumassa marraskuussa 2018 ja seuraavan vuoden tammikuussa se pelasi ammattilaispelaajia, ”TLO”ta sekä ”MaNa”a vastaan ja onnistui voittamaan molemmat 5-0. (AlphaStar: Mastering the Real-Time Strategy Game StarCraft II 2019.)

StarCraftiä on pidetty yhtenä vaikeimmista peleistä AI kehityksessä, koska pelissä pitää tehdä kauaskantoisia suunnitelmia ja sopeuttaa niitä vihollisen liikkeisiin. Pelissä on todella tärkeää yrittää tiedustella omia joukkoja hyväksikäyttäen, mitä rakennuksia ja teknologioita toinen pelaaja kehittää. Tämä johtuu siitä, että eri teknologioilla ja joukoilla on ”Kivi paperi sakset” -tyyppinen suhde toisiinsa. Esimerkiksi lentäviä yksiköitä vastaan tarvitaan joko omia lentäviä joukkoja, tai joukkoja, jotka pystyvät ampumaan ylöspäin. Koska peliä pelataan pääasiassa kilpailullisesti, tulisi AI:n toimia kuten ihmispelaajan, eikä esimerkiksi huijaten tietää mitä rakennuksia vastapuolella on. (AlphaStar: Mastering the Real-Time Strategy Game StarCraft II 2019.)

AI:n kehitystä hankaloittaa myös se, että pelin pelaajat kehittävät uusia strategioita jatkuvasti ja pelin päivityksissä muokataan joukkojen soveltuvuutta eri tilanteisiin. AI:n pitäisi jotenkin pystyä sopeutumaan muutoksiin ja muokata toimintatapojaan. Tämän takia koneoppiminen on varteenotettava vaihtoehto tämän tyyppiselle pelille.

Googlen AlphaStaria testaillaan silloin tällöin ihmisiä vastaan. Näihin testeihin on mahdollisuus päästä mukaan satunnaisesti, jos vaihtoehdon on hyväksynyt pelissä. Pelaajille ei kerrota, jos he pelaavat AphaStaria vastaan, sillä sen pelätään vaikuttavan siihen, miten ihmiset pelaisivat peliä. Kenties tulevaisuudessa AlphaStarista saadaan ammattilaispelaajille hyvä harjoittelukumppani.

4.4 Tulevaisuuden potentiaali

Tämä osio perustuu Buttner, M. (2018) Blogi kirjoitukseen, jossa kerrota Kinematica projektista tarkemmin.

Koneoppimista on AI kehityksen lisäksi kaavailtu myös pelitestaukseen, animaatioiden luontiin lennossa sekä pelikenttien rakentamiseen. Animaatioiden saralla Unity näytti ”Kinematica”-nimisestä projektista demon elokuussa 2018. Kyseisen projektin ideana on käyttää koneoppimista animaatioiden yhdistelemiseen ja muokkaamiseen lennossa. Perinteisesti jokaiselle animaatiolle pitää luoda siirtymät kaikkiin animaatioihin, jotka sitä voivat seurata. Esimerkiksi juokseva hahmo voi hypätä, kyykistyä, pysähtyä tai vaihtaa juoksun suuntaa.

Perinteisin menetelmin kaikki edellä mainitut siirtymät pitää luoda ja viilata erikseen, jotta animaatiot saadaan toimimaan ja näyttämään hyvältä. Kinematican avulla AI:lle annetaan ennakoivia tietoja siitä mihin animoitava hahmo on matkalla ja näiden tietojen perusteella se yhdistelee käyttäjän antamasta animaatio kirjastosta oikeanlaiset animaatiot. ”Kinematica”-projektista ei tosin ole kuulunut hetkeen mitään, mutta ilmeisesti se on edelleen työn alla. (Buttner 2018.)

5 KONEOPPIMISEN TESTAUS KEHITTÄMÄSSÄMME PELISSÄ

Pyrin työlläni testaamaan, sopiiko Unityn tämänhetkisellä koneoppimisympäristöllä koulutettu tekoäly vuoropohjaisen RPG-strategiapelin tekoälyksi. Tavoitteena on selvittää, onko tekoälyä järkevä toteuttaa tällä menetelmällä, sekä kartoittaa mitä hyötyjä ja haittoja menetelmän käytöstä on. Meitä kiinnostaa lähinnä saada tekoäly toimimaan taisteluissa. On myös tärkeää arvioida tekniikan käytännöllisyyttä sekä potentiaalia myöhemmille muutoksille, joita peliä kehittäessä saatetaan toteuttaa.

Peli, jota varten tätä tutkimusta teen, toimii taisteluissa mekaniikoiltaan siten, että vuorollaan jokainen hahmo voi liikkua tietyn määrän perustuen hahmon ominaisuuksiin. Liikkumisen lisäksi hahmo voi myös lyödä tai loitsia sekä tehdä jonkun pienemmän toiminnon, kuten poimia tavaran maasta tai käyttää kädenulottuvilla olevaa esinettä. Näitä mekaniikkoja hoitaviin järjestelmiin täytyy siis rakentaa jonkin näköinen yhteys koneen agentille.

Jotta toteutus olisi onnistunut, pitää AI:n pystyä liikuttamaan hallussaan olevaa hahmoa ja lyömään pelaajan hahmoja, jos joku niistä on tarpeeksi lähellä. Edellä mainittu riittää alkuun, mutta myöhemmin AI:n olisi hyvä oppia myös käyttämään eri esineitä. Tähän pisteeseen asti AI olisi helppo toteuttaa käyttäen perinteisiä menetelmiä, mutta toivon että koneoppimista hyödyntämällä AI oppii myös taktikoimaan ja reagoimaan tilanteisiin mielenkiintoisemmin kuin perinteinen tekoäly. Tämä määrä koulutusta riittää mielestäni myös testaamaan tekniikan toimivuuden ja käytön järkevyyden.

5.1 Pelin pohjajärjestelmät

Pelin pohjajärjestelmät on pyritty luomaan teoriaosuudessa mainitsemieni hyvien koodikäytänteiden mukaisesti. Eri järjestelmät on erotettu toisistaan ja järjestelmien visuaalista puolta sekä taustalla tapahtuvaa laskentaa on pyritty erottelemaan, jotta UI suunnittelijoilla on selkeämmät kokonaisuudet. Näiden pohjalta voi rakentaa käyttöliittymää ilman ohjelmistokehittäjien avustusta.

Meidän tapauksessamme NPC-hahmojen pitää pystyä liikkumaan, lyömään pelaajaa sekä käyttämään esineitä ja kykyjä. Lisäksi jokainen hahmo sisältää tietoja itsestään kuten elämäpisteiden määrän, liikkumisnopeuden, lyöntivoima yms.

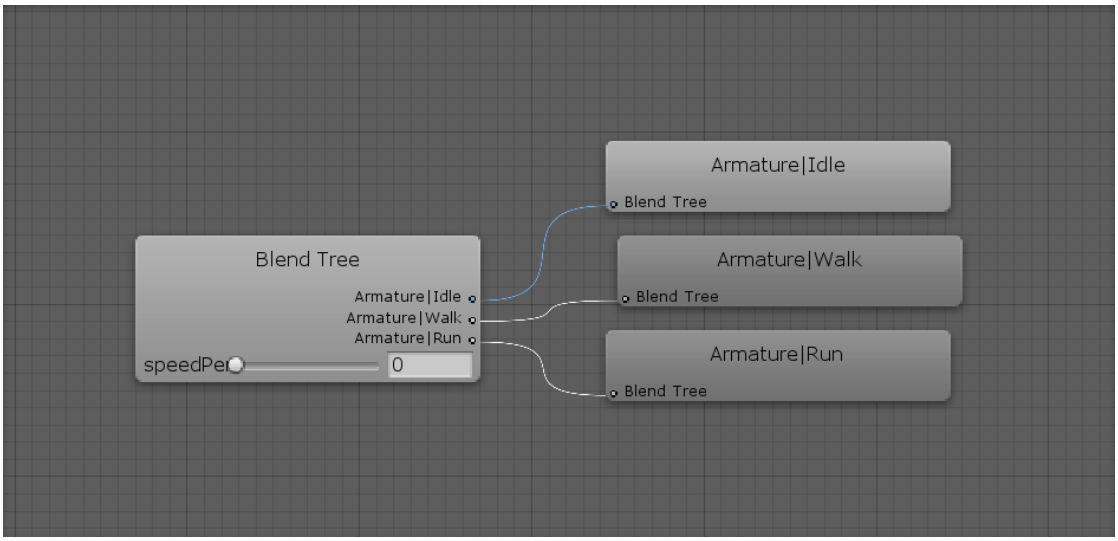
Koska haluamme pelaajahahmojen ja AI:n kontrolloimien hahmojen toimivan yhtenäisesti, pelimme on toteutettu niin, että jokainen järjestelmä periytyy yhteisestä luokasta AI:lle ja pelaajalle. Esimerkiksi hahmojen liikkumiselle pohja luokkana on Character Movement josta periytyy Player Movement ja AI movement luokat.

```
public abstract class CharacterMovement : MonoBehaviour
{
    public bool IsMoving;

    public NavMeshAgent Agent;

    protected NavMeshObstacle navObstacle;

    protected CharacterCombatController combat;

    public Vector3[] GetPath(Vector3 destinationPoint)...

    public Vector3 GetPointOnNavMesh(Vector3 point)...

    public void GoTo(Vector3 position)...

    public IEnumerator GetPathEnumerator(Vector3 destinationPoint, System.Action<Vector3[]> callback)...

    public void Stop()...

    public void LookAt(Transform target)...

    public void LookAt(Vector3 target)...

    protected void Super()...

    protected void Update()...

    private IEnumerator AgentMove(Vector3 targetPosition)...
}
```

Kuva 9. CharacterMovement abstrakti luokka, johon AI liikkuminen sekä pelaajan liikkuminen pohjautuu

Liikkumista toteuttaessamme Unityllä käytimme hyväksi valmista NavMesh-järjestelmää, joka luo staattisten objektien päälle pinnan, jonka päällä NavMeshAgentit voivat liikkua. Järjestelmän tekemän pinnan avulla hahmo voi välttää esteet ja liikkua haluamaansa paikkaan. (NavMesh Agent 2020.) NavMeshAgent-ärjestelmä on kustomoitavissa, mutta silti se rajoittaa toteutusta hieman. Meidän tapaukseemme se toimi kuitenkin erittäin hyvin.

Animaatiopuut on rakennettu Unityn omalla Animation Controller -järjestelmällä. Järjestelmä mahdollistaa muuttujien käyttämisen animaatioiden käynnistämisessä ja sekoittamisessa keskenään. Esimerkiksi hahmon liikkumisnopeudesta on tehty muuttuja, jonka pohjalta animaatiojärjestelmä sekoittaa paikallaan olo-, kävely- ja juoksuanimaatioita. Näin animaatio mukailee luonnollisennäköisesti nopeutta, jolla hahmo liikkuu. (Blend Trees 2020.)



Kuva 10. Animation Blend tree joka yhdistää hahmon liikkumisnopeuden perusteella paikallaan olo, kävely ja juoksu animaatioita

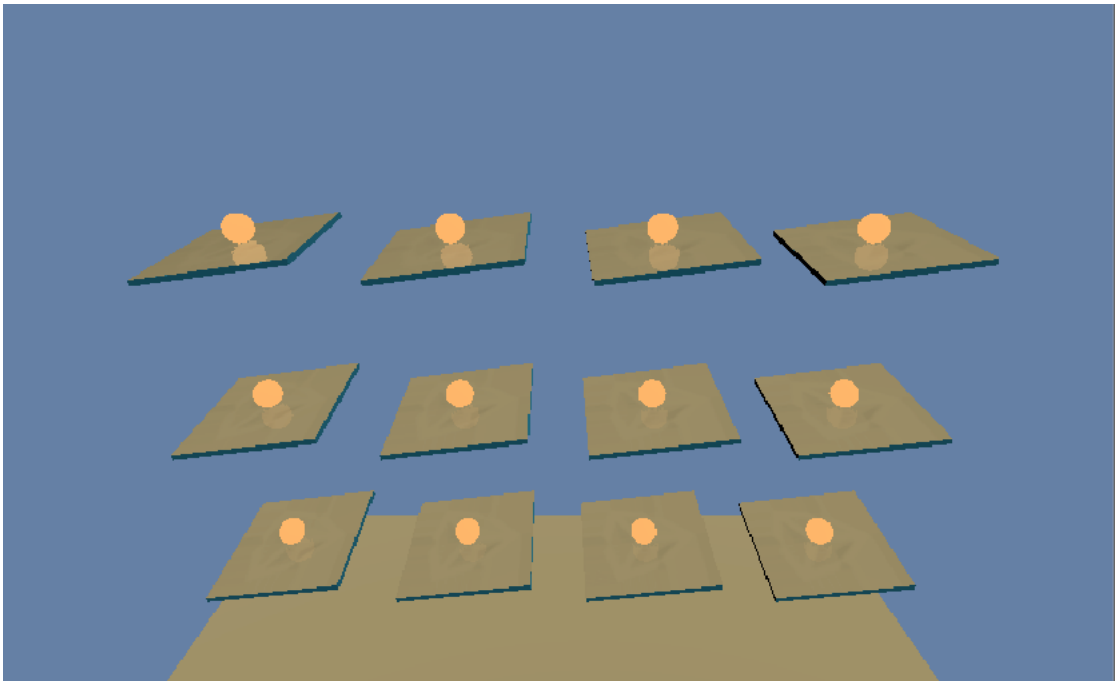
Valmiiden järjestelmien käyttö nopeuttaa kehitystä. Kompleksien järjestelmien rakentaminen vie niin paljon resursseja, että se ei ole edes mahdollista pienempää peliä varten. Huono puoli valmiiden järjestelmien käyttämisessä on, että eri peleistä voi tulla hyvin saman tuntuisia, jos kaikki käytetyt järjestelmät ovat valmiita. Tähän toki vaikuttaa paljon kunkin järjestelmän muokattavuus ja pelikehittäjien vision vahvuus viedä mekaniikkoja erilaisiin- ja kiinnostaviin suuntiin.

5.2 Ympäristön asennus

Voidaksemme käyttää Machine Learning Agents -työkalua, täytyy sen erilliset vaatimukset asentaa. Koska kyseessä on beetaversiossa oleva työkalu, se ei tule suoraan Unityn perusasennuksen mukana, eikä ole ladattavissa suoraan kaikkine vaatimuksineen. Asennusta hankaloittaa myös se, että TensorFlow-kirjastot ja itse Machine Learning Agents -työkalu tarvitsevat Python-ympäristön toimiakseen. Helpoin tapa saada kaikki tarvittavat järjestelmät toimimaan oli asentaa Anaconda ja luoda siihen Python 3.6 -ympäristö. Tämän jälkeen luotuun ympäristöön piti asentaa pip- paketinhallintajärjestelmä, itse TensoFlow- kirjastot. Tämän jälkeen pitää ladata itse Machine Learning Agents git repo ja asentaa se pip-paketinhallintajärjestelmää käyttäen, joka huolehtii lopuista Machine Learning Agents -työkalun tarvitsemista paketeista.

Kuva 11 Anacondalla luotu python ympäristö, jossa Machine Learning Agents pyörii

Seuraavaksi piti varmistaa, että ympäristö toimii ja osaa ottaa yhteyden Unity:n komponentteihin. Ladatusta ml-agents-kansiosta löytyy testiskenaarioita, joilla voi varmistaa, että kaikki toimii kuten pitää.



Kuva 12. Pallojen tasapainottelu demo, jossa AI ohjaa mailan kulmaa ja koittaa saada pallon pysymään mahdollisimman pitkään mailan päällä

Nopean testin jälkeen pallojen tasapainottelu alkaa onnistua AI:lta, joten kaikki näyttää toimivan oikein. Seuraavaksi toimimaan saatu ympäristö pitäisi saada kommunikoimaan meidän oman pelimaailmamme kanssa.

5.3 Yhdistäminen pelimaailmaan

Kun ympäristö oli asennettu ja sen toiminta testattu, piti se yhdistää meidän pelimaailmaamme, jotta pystyin testaamaan toimintaa oikeassa pelissä. Implementoin Agentti abstrakti luokasta EnemyAgent-nimisen luokan. Kyseinen luokka huolehtii palkinnoista sekä liikkeistä, joita agentti voi tehdä. Agentissa on erilliset ylikirjoitettavat funktiot observaatioiden keräämiselle, käskyjen vastaanottamiselle sekä sille mitä tapahtuu, kun agentti palautuu alkutilaan koulutusjaksojen välissä.

Seuraavaksi loin akatemian, jonka nimesin TurnBasedAcademyksi. Akatemia on ainoastaan implementointi abstraktista luokasta, eikä sisällä muuta toiminnallisuutta.

Lisäksi loin aivot, joiden pohjalta AI-mallit luodaan. Nimesin aivot TurnBasedBrainiksi. Aivoissa pitää määritellä, minkä tyyppisiä päätöksiä kone tekee. Alustavan suunnitelmani mukaan agentti voisi suoraan valita tarkan kohdan,

johon se liikkuu NavMeshin päällä. Tämänhetkinen koneoppimisen implementointi Machine Learning Agents -paketissa ei kuitenkaan tue ennalta määritettyjen toimintojen (Discrete Actions) ja vapaiden toimintojen yhdistämistä (Continuous Actions). Tämä rajoittaa toteutustamme AI hahmon liikkumisen osalta. Päädyin siis toteuttamaan aivoihin neljä yksittäistä toimintoa hahmon liikuttamista varten.

5.4 Vihollishahmon liikuttaminen

Kuten edellä mainitsin, Machine Learning Agents ei tukenut suoraan sellaista päätöksentekoa, jota pelimme olisi kaivannut. Paras keksimäni keino, jolla vajetta voi paikata, oli luoda liikkumiselle valmiita funktioita, joista koneoppiminen voi valita tilanteeseen sopivimman.

Liikkumis-funktioiksi loin aluksi Charge- ja Runaway-funktiot. Koneella on siis valittavanaan liikkumatta olemisen lisäksi päälle ryntääminen ja karkuun juokseminen. Myöhemmin voitaisiin lisätä funktiot esimerkiksi suojan etsimiselle.

Hyvä puoli tässä tavassa toteuttaa liikkuminen on se, että liikkumisen koulutus on paljon nopeampaa. Liikkumismekaniikka olisi voinut olla todella hidas opettaa koneelle. Pääsemme myös eroon valtavasta määrästä ympäristöhavaintoja, joilla olisimme selvittäneet agenteille ympäristön muodot ja esteet. Rajoitamme kuitenkin todella paljon agentin päätöksien vapautta ja ennalta-arvaamattomuutta, jota koneoppiminen olisi tuonut peliin.

5.5 Pelaajahahmon lyöminen

Liikkumiskoulutuksessa ilmenneiden rajoitteiden takia varsinainen koulutus alkaa tästä. Tavoitteena on saada AI liikkumaan ja lyömään pelaajahahmoa. Tässä koulutusvaiheessa pelaajahahmo ohittaa aina vuoronsa, sillä tavoitteenamme on vain varmistaa, että alustavat koulutusparametrit sekä ympäristömme toimivat suurin piirtein halutulla tavalla. Muutamia pelin koodeja joudin hiukan muokkaamaan, jotta koulutusympäristö toimisi oikealla tavalla.

Lisänä edellisiin liikkumisfunktioihin koneella on nyt myös hallussaan komento, jolla se voi lyödä vastustajaa sekä komento, jolla vuoro lopetetaan. Oikea ratkaisu, joka koneen pitäisi tässä vaiheessa pystyä tekemään on, että

pelaajaa kohti pitää ensin liikkua ja sitten pelaajaa pitää lyödä. Kun liikkuminen ja lyönti on käytetty, pitää AI:n lopettaa vuoro. Palkinto tulee sen mukaan, mitä vähäisemmillä päätöksillä kone saa pudotettua pelaajan elämäpisteet alle kymmeneen.

Kuva 13. Koulutettava AI hyökkää pelaajan kimppuun ja tekee ruudussa näkyvät 6 vahinkoa lyömällä pelaajaa

Testeissä tuli selväksi, että AI oppii suhteellisen nopeasti kaiken mitä tässä kohtaa sen tarvitsinkin oppivan. Tässä vaiheessa huomasin harmikseni, että AI:n koulutus on epävakaata näin suureen projektiin yhdistettynä. Ympäristö jää todella usein jumiin tallentaessaan uutta mallia. Tallennustiheyttä hidastamalla ja katkaisemalla koulutus manuaalisesti voi toki tallentaa mallin, mutta tämä tekee koulutuksesta vaivanloista.

En näe tarpeellisena yrittää kouluttaa AI:ta tämän pidemmälle. Rajoitteiden sekä kaatuilun takia se ei selkeästi ole oikea valinta tätä peliä varten.

5.6 Lopputulokset ja pohdinta

Perinteiseen AI:n toteutukseen syvällisemmin perehdyttyäni tämän opinnäytetyön teoria osuutta varten tuli itselleni hyvin selväksi, kuinka AI kannattaisi rakentaa kyseistä tapausta varten. Machine Learning Agents -paketilla luotuun AI:hin verrattuna visuaalinen käytöspuu näyttää kehittäjälle selkeästi, mitä AI

tekee ja miksi. Käytöksen selkeys auttaa ratkaisemaan ongelmia kehitysvaiheessa ja helpottaa myöhempiä muutoksia. Perinteistä pelikokemusta rakennettaessa Machine Learning Agents -paketilla luotu AI vaikeuttaa pienten muokkausten tekemistä, sillä jos AI:n pitäisi omaksua jokin uusi ominaisuus, joka peliin on toteutettu, pitää AI:ta kouluttaa ja palkintorakennetta miettiä uudestaan.

Nykyisellään paketti sopii selkeästi paremmin pienempiin, rajatumpiin pelikokonaisuuksiin, joissa AI:lla on käytettävissään tarkasti rajattu määrä niin kontrolleilta kuin havaintoja. Paketti tuntui soveltuvan paljon paremmin teoriaosuutta varten rakentamaani auton ohjaus esimerkkiin. Tämä johtuu siitä, että kyseisessä tapauksessa AI:lla on kontrolloitavissaan nuolinäppäimet, jolla autoa ohjataan ja se saa tietoonsa 5 seuraavaa pistettä tiessä. Tämä tekee paljon selkeämmän rajauksen toimintoihin ja havaintoihin verrattuna RPG-peliin, jossa toimintoja ja hyödyllisiä havaintoja pitää syöttää todella monesta eri asiasta ja niistä on paljon vaikeampi muodostaa järkevää kokonaisuutta, joka kertoo AI:lle tarpeeksi olematta liian kompleksi.

Tästä kokeilusta viisastuneena näen kyllä mahdollisuuden rakentaa toteutus abstraktimmalla tasolla, jolloin jokainen osa voisi olla oma koulutettu agenttinsa ja nämä kaikki toimisivat suoraan parametrien perusteella. Näin parametrit voisivat olla myös koulutuksen jälkeen helposti säädeltävissä ja yhtä osaa muokattaessa ei tarvittaisi toteuttaa kokonaan uutta koulutus ja palkintorakenne sykliä.

Vaikka testausosuus jäi lyhyemmäksi kuin olin toivonut sain siitä kuitenkin vastauksen haluamaani kysymykseen. Testauksen kanssa olisin myös voinut käyttää enemmän aikaa ongelmien kiertämiseen, mutta yksi päätavoitteistani oli selvittää, onko tekniikan käyttö käytännöllistä, johon nousseiden ongelmien määrä antoi vastauksen. Tunnen myös syventäneeni tietouttani ja valmiuksiani toteuttaa AI koneoppimisella tai perinteisillä menetelmillä. Itselleni tuli myös hyvin selväksi minkälaisia pelejä tehdessä kannattaa Machine Learning Agents -koneoppimispakettia soveltaa. Lisäksi perehtyminen tilakoneisiin ja käytöspuihin on avannut ajatuksiani helpoille tavoille kuvata ja toteuttaa päätösten tekoa. Näiden tekniikoiden hallitseminen auttaa minua varmasti jatkossa pelikehityksen parissa.

LÄHTEET

AlphaStar: Mastering the Real-Time Strategy Game StarCraft II. 2019. DeepMind. Blogi kirjoitus. Saatavissa: https://deepmind.com/blog/article/alphastar-mastering-real-time-strategy-game-starcraft-ii [viitattu 16.10.2019]

Barrera, R. Kyaw, S. Peters, C. Swe, T. 2017. Unity AI Game Programming. Birmingham: Packt Publising Ltd.

Blend Trees. 2020. Unity Technologies. Dokumentaatio. Saatavissa: https://docs.unity3d.com/Manual/class-BlendTree.html [viitattu 25.4.2020]

Buttner, M. 2018. Announcing Kinematica: Animation meets machine learning. Blogi Kirjoitus. Saatavissa: https://blogs.unity3d.com/2018/06/20/announcing-kinematica-animation-meets-machine-learning/ [viitattu 16.10.2019]

Djudjic, D. 2019. Colourise.sg the best Ai-Based colorizing tool so far. Artikkeli. Saatavissa: https://www.diyphotography.net/colourise-sg-the-best-ai-based-colorizing-tool-so-far/ [viitattu 14.10.2019].

Eady, T. 2019. Tesla's Deep Learning at Scale: Using Billions of Miles to Train Neural Networks. Artikkeli. Saatavissa: https://towardsdatascience.com/teslas-deep-learning-at-scale-7eed85b235d3 [viitattu 16.10.2019].

Halpern, J. 2019. Developing 2D Games with Unity: Independent Game Programming with C#. New York: Springer Science+Business Media

Juliani, A., Berges, V., Vckay, E., Gao, Y., Henry, H., Mattar, M., Lange, D. 2018. Unity: A General Platform for Intelligent Agents. Dokumentaatio. Saatavissa: https://github.com/Unity-Technologies/ml-agents. [viitattu 16.10.2019]

Kukara, M., Kononenkoa, Igor., Grošeljb C., Kralja K., Fettich Jure. 1999. Analysing and imporivng the diagnosis of ischaemic heart disease with machine learning. Tutkielma. Saatavissa: https://www.sciencedirect.com/science/article/pii/S0933365798000633 [viitattu 14.10.2019]

Lagarde, S. 2018. The High Definition Render Pipeline: Focused on visual quality. Blogi Kirjoitus. Saatavissa: https://blogs.unity3d.com/2018/03/16/the-high-definition-render-pipeline-focused-on-visual-quality/ [viitattu 16.10.2019]

Machine Learning Glossary. 2020. Google. Dokumentaatio. Saatavissa: https://developers.google.com/machine-learning/glossary/ [viitattu 25.4.2020]

NavMesh Agent. 2020. Unity Technologies. Dokumentaatio. Saatavissa: https://docs.unity3d.com/Manual/class-NavMeshAgent.html] [viitattu 25.4.2020]

Post-processing overview. 2019. Unity Technologies. Dokumentaatio. Saatavissa: https://docs.unity3d.com/Manual/PostProcessingOverview.html [viitattu 14.1.2019]

Ramsey, R. 2018. Bethesda Will Keep the Same Fundamental Game Engine for The Elder Scrolls VI, Starfield. Saatavissa: http://www.pushsquare.com/news/2018/11/bethesda_will_keep_the_same_fundamental_game_engine_for_the_elder_scrolls_vi_starfield [viitattu 16.10.2019]

Savinov, N. & Lillicrap, T. 2018.Curiosity and Procrastination in Reinforcement Learning. Blogi kirjoitus. Saatavissa: https://ai.googleblog.com/2018/10/curiosity-and-procrastination-in.html [viitattu 16.10.2019]

Ward, J. 2008. What is a Game Engine? Artikkeli. Saatavissa: https://www.gamecareerguide.com/features/529/what_is_a_game_.php [viitattu 16.10.2019]