Pelien tekoälyn toteutus Unity-pelimoottorilla

Ammattikorkeakoulututkinnon opinnäytetyö

Tietojenkäsittelyn koulutusohjelma

Hämeenlinna, kevät 2017

Jimi Petramaa

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Tietojenkäsittely, tradenomi
Visamäki, Hämeenlinna

Tekijä Jimi Petramaa **Vuosi** 2017

Työn nimi Pelien tekoälyn toteutus Unity-pelimoottorilla

Työn ohjaaja/t Tommi Saksa

TIIVISTELMÄ

Opinnäytetyön tarkoitus on selvittää, kuinka tekoälyä voidaan toteuttaa Unity-pelimoottorilla tehtyihin peleihin. Työn tavoitteena on demonstroida yksinkertaisilla esimerkeillä pelien tekoälyä.

Opinnäytetyön teoriaosuudessa avataan tekoälyn suurimpia virstanpylväitä. Työssä käydään läpi myös tekoälyn tyyleistä peliohjelmoinnissa, Unity-pelimoottorista ja reitinetsintäalgoritmeista.

Ensiksi käytännönosuudessa käydään läpi kaksi erilaista reitinetsintää demonstroivaa esimerkkiä. Toiseksi käsitellään tasohyppely-peliin soveltuvan tekoälyn vihollishahmolle. Viimeiseksi käydään vielä ylhäältäpäin kuvatun 2D-pelin hahmon tekoäly. Tavoitteisiin päästiin hyvin ja työn tuloksia voitaisiin jatko kehittää.

Avainsanat **Tekoäly, reitinetsintä, heuristiikka, algoritmi, Unity**

Sivut 26

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

ABSTRACT

Degree Programme in Business Information Technology
Visamäki, Hämeenlinna

Author Jimi Petramaa **Year** 2017

Subject Artificial intelligence implementation in the Unity game engine

Supervisors Tommi Saksa

ABSTRACT

The purpose of this thesis was to find out how artificial intelligence can be implemented in the games made in Unity. The goal was to demonstrate simple game examples with artificial intelligence.

The theoretical part of the thesis is about the largest milestones of artificial intelligence. The theoretical part also goes through artificial intelligence programming styles, Unity game engine and pathfinding algorithms.

The practical part of the thesis consists of two ways to implement pathfinding algorithms to Unity games. It also consists of two 2D-games with artificial intelligence characters. The tasks were accomplished and could be further developed.

Keywords **Artificial intelligence, pathfinding, heuristic, algorithm, Unity**

Pages 26

SISÄLLYS

1 JOHDANTO 1

2 TEKOÄLYN HISTORIA 2

2.1 Turingin testi 2
2.2 Tekoälyn historia peleissä 3

3 TEKOÄLY PELEISSÄ 6

3.1 Tyylit 6
3.2 Reitinetsintä 9
3.2.1 Leveyssuuntainen läpikäynti 9
3.2.2 Dijkstran algoritmi 10
3.2.3 A* 10
3.3 Rajoitukset 10

4 UNITY-PELIMOOTTORI 12

5 TAVOITTEET 13

6 PELITEKOÄLYN TOTEUTTAMINEN 14

6.1 Reitinetsintä sokkelossa 14
6.1.1 Reitinetsinnän toteuttaminen A* Pathfinding projectilla 14
6.1.2 Toteuttaminen Unity Navigation -ominaisuudella 18
6.1.3 Vertailu 19
6.2 Tasohyppely-esimerkki 19
6.3 Hyökkäys-esimerkki 22

7 YHTEENVETO 25

LÄHTEET 26

1 JOHDANTO

Opinnäytetyöni aiheeni on tekoäly peliohjelmoinnissa. Olen ollut aikaisemmin kiinnostunut peliohjelmoinnista ja halusin valita opinnäytetyökseni aiheen, joka liittyi jotenkin peliohjelmointiin. Pelitekoäly oli siis kiinnostava aihe. Tekoälyn historia ja tekoälyn syntymisen tutkiminen olivat mielenkiintoisia, sillä niistä löytyi paljon artikkeleja ja tutkimuksia. Tekoälyesimerkkejä tehdessä huomasin sen olevan vaikeaa ja välillä sai miettiä kuinka pulmat ratkoisi koodissa.

Alkuun kerron tekoälyn synnystä ja historiasta. Avaan hieman, milloin tekoälyn tutkiminen lähti käyntiin. Kerron myös muutamista suurimmista virstanpylväistä, joita tekoälyn tutkimisessa on tehty. Tekoälyn historian jälkeen kerron pelitekoälyn kehityksestä 1970-luvun alusta nykypäivään.

Kolmannessa luvussa kerron pelitekoälyn tyyleistä, jotka ovat: ”*Hacking”*, heuristiikka ja algoritmit. Myös millaista tekoälyä peliohjelmoinnissa käytetään, sekä reitinetsintäalgoritmeista. Reitinetsintäalgoritmeista valitsin kolme erilaista vaihtoehtoa: leveyssuuntainenläpikäynti, Dijkstran algoritmi ja A* -algoritmi.

Neljännessä luvussa kerron hieman Unity-pelimoottorista. Kerron myös, kuinka Unityllä saadaan aloitettua projekti. Lopuksi kerron myös Unityn käyttöjärjestelmästä ja mitä kaikkea siihen sisältyy.

Teoriaosuuden jälkeen tuotan muutaman demonstraation tekoälystä peliohjelmoinnissa. Nämä esimerkit toteutan Unity-pelimoottorilla, käyttäen koodauskielenä C#-kieltä (CSharp). Esimerkki-pelit ovat grafiikalta erittäin yksinkertaisia, sillä niillä on tarkoitus vain demonstroida tekoälyä peliohjelmoinnissa. Ensimmäisenä toteutin kaksi erilaista reitinetsintään (pathfinding) soveltuvaa esimerkkiä. Algoritmeja en tehnyt näihin itse, vaan latasin ja asensin valmiit algoritmit. Myöhemmin työssä vertailen näitä kahta reitinetsintäalgoritmia. Vertailen asennusten helppokäytettävyyttä ja minkälaisissa tilanteissa niitä on hyvä käyttää. Seuraava esimerkki jonka tein on sivustapäin kuvattu ”*platform*” (tasohyppely)-peli. Tein esimerkkiin vihollishahmon, joka kävelee edestakaisin, kääntyen aina, jos vastaan tulee seinä tai kuilu, jonka yli hyppäämiseen ei riitä vauhti. Jos kuilun toisella puolella on lattia mihin vihollishahmon vauhti riittää hyppäämään, niin silloin hahmo hyppää.

Tutkimuskysymykseni opinnäytetyössäni ovat: kuinka tekoälyä käytetään peliohjelmoinnissa, kuinka reitinetsintää käytetään ja tuotetaan Unity-pelimoottorissa?

2 TEKOÄLYN HISTORIA

Tekoälyä on tutkittu jo 1940-luvulta asti, ja nykypäivänä tekoälyä löytyy melkein kaikkialta. Tekoälyihin törmää nykypäivänä kodinkoneissa, tietokonepeleissä ja autoissa. Joissain tekoäly ajattelee ja toimii itsenäisesti, kun taas toisissa se vain nopeuttaa tai helpottaa käyttäjää toimimaan. Teknologian kehittyminen eteni 1940-luvulla pisteeseen, jolloin tulivat ensimmäiset tietokoneet.

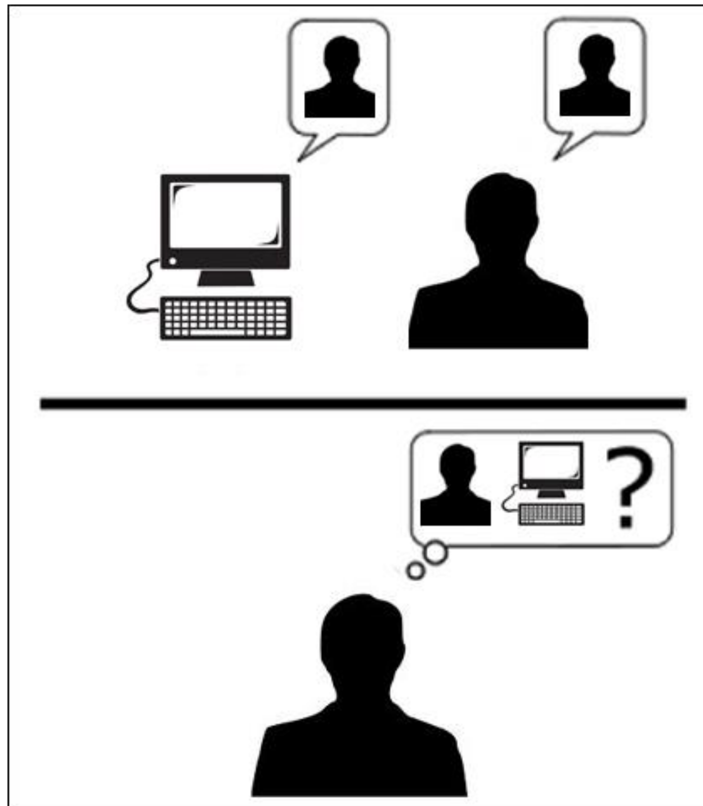
Sanaa ”tekoäly” ruvettiin käyttämään vuonna 1956 Dartmouthin konferenssissa, kun John McCarthy ideoi termin. John McCarthy kutsui tähän konferenssiin tärkeimpiä tutkijoita eri tieteen aloilta. Samassa konferenssissa tutkijat Newell, Simon ja Shaw loivat The Logic Theorist -ohjelman. Monet pitävät tätä ohjelmaa ensimmäisenä tekoälyllisenä ohjelmana. Ohjelman idea oli esittää ongelmat päätöspuumallina ja ratkoa niitä valitsemalla haara joka todennäköisimmin johtaisi oikeaan ratkaisuun. (William Stewart 2000.)

Vuonna 1957 sama kolmikko, Newell, Simon ja Shaw olivat tuottaneet entistä paremman ohjelman *General Problem Solver* (GPS). GPS pystyi ratkomaan entistä paremmin perusongelmia kuin aikaisempi The Logic Theorist -ohjelma. McCarthy julkisti vuonna 1958 seuraavan läpimurron tekoälyn historiassa. John oli kehitellyt uuden ohjelmointikielen, LISP. LISP tulee sanoista List Processing ja siitä tuli suosittu ohjelmointikieli tekoälyä tutkivien joukossa. (Heino 2003.)

2.1 Turingin testi

Alan Turing suunnitteli vuonna 1951 testin nimeltä *”The Imitation Game”.* Alkuperäinen testi ei sisältänyt lainkaan tekoälyä. Testissä on kolme huonetta, kaikki huoneet ovat yhteydessä toisiinsa tietokoneilla. Yhdessä huoneessa on mies, toisessa huoneessa nainen ja kolmannessa huoneessa on tuomari. Testissä tuomarin tehtävänä on yrittää selvittää, kumpi tietokoneella keskustelevista henkilöistä on mies. Miehen tehtävä on auttaa tuomaria todistamaan itsensä mieheksi. Nainen koettaa huijata tuomaria luulemaan toisin. (Reingold 1999.)

Alan Turing muokkasi testiä siten, että hän teki siitä version, jossa miehen ja naisen sijaan testissä olikin ihminen, tietokone ja tuomari. Kuvassa 1 tuomarin tehtävänä oli nyt selvittää, kumpi näistä testattavista oli ihminen ja kumpi tietokone. Turing ehdotti, että jos tuomari olisi valinnut yhtä varmasti tietokoneen kuin ihmisen, niin tietokone olisi kelvollinen simulaatio ihmisen mielestä. Nykyään testissä on vain tuomari ja hänen tehtävänsä on selvittää, onko kyseessä tietokone vai ihminen. (Reingold, 1999.)



Kuva 1. Turingin testin idea (acvoice.com 2014).

2.2 Tekoälyn historia peleissä

Vuoden 1956 Dartmouthin konferenssissa oli mukana kaksi insinööriä IBM:ltä: Nathaniel Rochester ja Arthur Samuel. Arthur Samuel oli tutkinut koneiden oppimista (Machine Learning). Koneoppi oli algoritmeja, jotka saisivat koneet itsenäisesti ratkomaan ongelmia kuin ohjelmoijan kovakoodatut ratkaisut. Hän keskittyi kehittämään ohjelmaa, joka pystyisi pelaamaan tammea. Ei mennyt kauaakaan, kunnes Arthur Samuelilla oli ohjelma, joka pelasi tammea. Pelitasoltaan ohjelma voitti vain aloittelijoita. (Kurenkov 2016.)

Tammen jälkeen tutkijat rupesivat työskentelemään shakin parissa. Asiantuntijat pitävät yhtenä suurimpana virstanpylväänä tekoälylle, kun vuonna 1997 IBM:n kehittämä kuvassa 2 oleva Deep Blue -tietokone voitti shakin maailmanmestarin Gary Kasparovin lukemin 3,5 – 2,5. (Kurenkov 2016.)

Kuva 2. Deep Blue-tietokone tietokoneiden historian museossa. (Jim Garadner, 2007.)

AI Game Programmers Guild on jakanut pelien tekoälyhistorian kuuteen aikakauteen.

Basic Patterned

Basic Patterned tyyliä käytettiin ensimmäisissä kolikkopeleissä ja tietokonepeleissä 1970 – 1980-luvuilla. Vuonna 1970 Atarin perustajat Nolan Bushnelli ja Ted Dabney julkaisivat Computer Space -kolikkopelin, jossa vihollishahmot liikkuivat käyttäen tekoälyä. (AI Game Programmer Guild 2011.)

Viholliset käyttivät kaavoja liikkumiseen ja ampumiseen, jotka olivat toistuvia, yksinkertaisia ja lyhytkestoisia. Pelaajan kontrollit eivät vaikuttaneet mitenkään vihollisen kaavoihin. Esimerkki tämäntasoisista videopeleistä on alkuperäinen Space Invaders ja Donkey Kong. (AI Game Programmer Guild 2011.)

Simple Hard-coded Rules

Simple Hard-coded Rules. Tässä tyylissä pelihahmoille on määritetty liikkuminen tai tapahtuma algoritmeilla. Vihollishahmoihin vaikuttaa pelaajan tekeminen, mutta usein vain liikkumalla päiten tai liikkumalla poispäin.

Tätä tekoälyntyyliä käytettiin kolikkopeleissä ja tietokonepeleissä 1970-luvulta – 1990-luvun alkuun. (AI Game Programmer Guild 2011.)

Esimerkki Pac-Man-pelissä jokaisella haamulla on oma tekoäly liikkumiseen sokkelossa ja ne myös lähtevät pelaajaa karkuun pelaajan syötyään erikoispisteen. (AI Game Programmer Guild 2011.)

Advanced Patterned

Kolmanneksi tulee *Advanced Patterned* ja sitä käytettiin lähinnä 1980-luvulla. Vihollishahmoja liikutetaan edelleen valmiiksi määritetyillä koodeilla, mutta ovat hieman kehittyneempiä. Muun muassa Zeldassa ja Super Mario Brothersissa käytettiin tätä tyyliä. (AI Game Programmer Guild 2011.)

Tactical Reaction

Tactical Reaction tekoälyllä on lyhytaikaisia reaktioita vieressä tapahtuviin asioihin. Moni taistelupeli, kuten Street Fighter, käyttää lyhytaikaista reaktiota pelaajan hyökkäyksen torjumiseen ja omaan hyökkäykseen. Tätä tyyliä ruvettiin käyttämään 1980-luvun puolenvälin ja käytetään edelleen nykypäivänä. (AI Game Programmer Guild 2011.)

Tactical Reasoning

Viidentenä tulee *Tactical Reasoning.* Yksittäiset tekoälyn ohjaamat hahmot tekevät edistyneempiä päätöksiä, ottaen jopa ympäristössä tapahtuvan huomioon. Tämä tyyli eroaa *Tactical Reactionista* siten, että tässä toiminnot ovat pidempiaikaisia ja se sisältää useampia tapahtumia peräkkäin. Esimerkkinä reitinetsintä ja suojaan meneminen. Tätä tyyliä Ruvettiin käyttämään 1990-luvun puolivälissä ja sitä käytetään edelleenkin. (AI Game Programmer Guild 2011.)

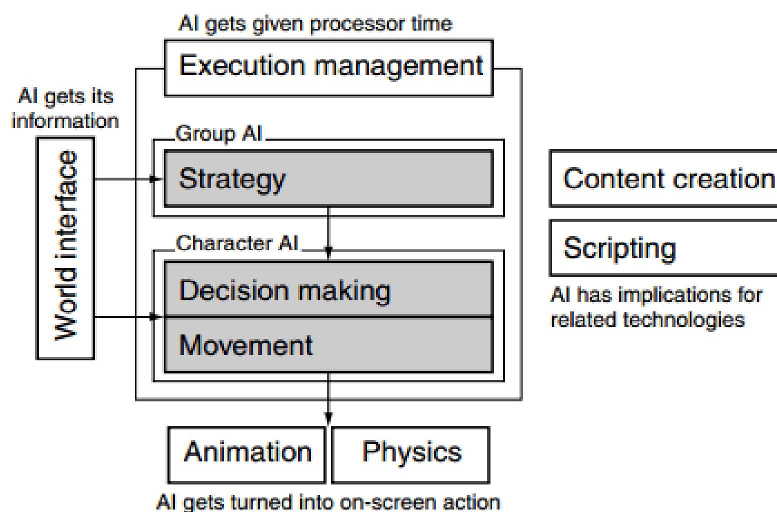
Strategic

Strategicin kehitys alkoi 90-luvun loppupuolella. Tekoälyn ohjaamat yksittäiset hahmot ovat tietoisia toisistaan. Moni tätä tekoälyajattelua käyttävä peli käyttää ”*Overseer”*-tekoälyä koordinoimaan useita yksittäisiä hahmoja, jotta suurempi ryhmä osaisi toimia paremmin yhdessä. Esimerkkejä tämäntasoisista peleistä ovat Halo, StarCraft 2 ja Left 4 Dead. (AI Game Programmer Guild 2011.)

3 TEKOÄLY PELEISSÄ

Nykypäivänä lähes jokaisessa pelissä on mukana tekoälyä, nopeimman reitin etsimistä, tekoälyn ohjaama hahmo tai yksinkertaista asioiden ohjaamista. Suurimmassa osassa moderneissa peleissä tekoälyllä on kolme perustarvetta: kyky liikuttaa hahmoa, kyky tehdä päätös minne liikkua ja kyky ajatella taktisesti tai strategisesti (Millington & Funge 2006, 8).

Millingtonin ja Fungen (2006, 9) mukaan kaikki pelit eivät tarvitse näitä kaikkia perustarpeita. Lautapelit, peleissä kuten tammi tai shakki, käytetään vain strategista ajattelua. Yksittäiset hahmot ja nappulat eivät itse tee päätöksiä mihin liikkuvat. Monissa peleissä taas ei ole strategista ajattelua ollenkaan. Monissa tasohyppelypeleissä viholliset toimivat itsenäisesti reagoiden vain pelaajan tekemisiin ja liikkeisiin. Moni peli sisältää liikkumiseen tarvittavia algoritmeja, mutta niissä ei ole edistynyttä päätöksentekoa. Toista ääripäätä edustavissa peleissä taas ei tarvita algoritmeja liikkumiseen. Kuvassa 3 esitetään tekoälyn mallia, esimerkiksi Real-Time Strategy (RTS) tai vuoropohjaisissa peleissä ei tarvita algoritmeja liikkumiseen, vaan kun hahmolle päätetään minne mennä, se menee lyhintä reittiä sinne.



Kuva 3. Tekoälyn malli peliohjelmoinnissa. (Artificial Intelligence for Games, Second Edition, 2006.)

3.1 Tyylit

Peliohjelmoinnissa käytetään enimmäkseen kolmea eri tekoälytyyliä, *hackingia*, heuristiikkaa ja algoritmeja.

Hacking, on pieniä asioita, joilla saadaan peli tuntumaan ja näyttämään järkevämmältä. ”Se näyttää kalalta, se tuoksuu kalalta; se on luultavasti

kala”. Tällaista ajattelutapaa kutsutaan behaviorismiksi. Yleensä peliohjelmoijat eivät tarvitse hahmoa, joka toimii ja ajattelee kuin ihminen, vaan että se tekee tietyn asian. Yksinkertaista satunnaisnumeron luomista ei luokitella tekoälytekniikaksi, mutta sillä voidaan luoda asioita, jotta peli näyttäisi järkevämmältä kuin se onkaan. (Millington & Funge 2006, 22.)

Millington ja Funge (2006, 22) kertovat hyväksi esimerkiksi luovasta pelitekoälyn kehityksestä *The Sims* -pelisarjan pelit. Vaikka pelin hahmoilla, simeillä, on paljon tekoälyllistä koodia, niin silti suurin osa simien käyttäytymisistä esitetään animaatioilla. Jotta oikeat animaatiot suoritetaan oikeissa paikoissa, ei vaadi paljon koodia taakseen. Pienillä animaatioilla saadaan peli näyttämään paljon älykkäämmältä ja paremmalta.

Kuva 4. The Sims –pelissä näytetään animaatioilla mitä hahmot puhuvat. (The Sims 3, 2009.)

Heuristiikka, on arvioitu ratkaisu, joka toimii useimmissa tapauksissa, mutta luultavasti ei toimi kaikissa. Ihmiset käyttävät heuristiikkaa päivittäin. Ihmiset eivät mieti kaikkia mahdollisia vaihtoehtoja tehdessään jotain, vaan toimivat sillä tavalla minkä ovat aiemmin oppineet tai löytäneet toimivaksi ratkaisuksi. Esimerkiksi ”älä mene vieraan auton kyytiin tai älä ota tuntemattomalta ihmiseltä karkkia”. (Millington & Funge 2006, 23.)

Ensimmäisissä Real-Time Strategy (RTS) -peleissä, esimerkiksi Blizzardin julkaisemassa Warcraft-pelissä käytettiin hahmojen liikuttamiseen heuristiikkaa. Kaukaa ampuvat hahmot liikkuivat niin kauan eteenpäin, kunnes pystyivät ampumaan vastustajan hahmoa. Vaikka hahmojen liikuttaminen tähän tapaan toimi useimmissa tapauksissa, se ei ollut paras vaihtoehto.

Pelaajan hahmot saattoivat kävellä vastustajan suuren tykin kantoalueelle. Monet pelaajat eivät pitäneet tästä tyylistä ja myöhemmin RTS-peleihin tuli vaihtoehtoja, liikkuvatko hahmot tähän tapaan vai eivät. (Millington & Funge 2006, 23.)

Peliohjelmoinnissa käytetään erilaisia heuristiikkoja erilaisiin tilanteisiin. Alla on selitetty kolme yleisintä peleissä käytettyä heuristiikkaa.

Lupaavin asia ensin

Jos tekoälylle on tarjolla useampi vaihtoehto, on usein mahdollista tehdä vaihtoehdoille nopea arviointi ja antaa jokaiselle vaihtoehdolle numero. Vaikka numerointi olisikin epätarkka, kokeilemalla vaihtoehtoja parhaiten arvioidusta huonoimpaan. Tekoäly löytää parhaimman tuloksen, toisin kuin täysin satunnaisessa kokeilujärjestyksessä. (Millington & Funge 2006, 24.)

Vaikein asia ensin

Vaikeimman asian tekemisellä on yleensä seuraamuksia muihin asioihin. On parempi tehdä vaikein asia ensin kuin tehdä helpoimmat ja huomata ettei vaikeinta voikaan enää tehdä. Esimerkiksi, jos pelaajan armeijalla olisi 4 peikkoa ja yksi jättiläinen, ja ne pitäisi jakaa kahteen yhtä vahvaan joukkoon. Tällä tyylillä saadaan jaettua joukot tasaisemmin. Jättiläinen on vaikein sijoittaa joukkoihin, joten se jaetaan ensimmäisenä. Jos peikot jaettaisiin ensiksi kahteen, joukoista ei tulisi yhtä vahvoja. (Millington & Funge 2006, 24.)

Harvinaisin asia ensin

Esimerkiksi joukko pelaajan sotilaita joutuu hyökkäyksen uhriksi. Hyökkääjillä on panssarivaunu ja sen pystyy tuhoamaan vain singolla. Yhdellä pelaajan sotilaista on sinko. Kun sotilaat päättävät keneen ne hyökkäävät, niin ”harvinaisin asia ensimmäisenä” -heuristiikka otetaan käyttöön. On vaikeinta tuhota panssarivaunu, joten tämä täytyy tuhota ensimmäisenä. (Millington & Funge 2006, 24.)

Hackingilla ja heuristiikalla pääsee jo pitkälle pelitekoälyssä, mutta jos halutaan tarkempia tuloksia, pitää ottaa käyttöön algoritmit. Algoritmit ovat tarkempia ja niillä saadaan tehtyä monimutkaisempia käyttäytymisiä, mutta ne ovat yleensä paljon raskaampia. Algoritmeja käytetään peliohjelmoinnissa esimerkiksi reitinetsintään, päätöksen tekoon tai taktiseen ajatteluun.

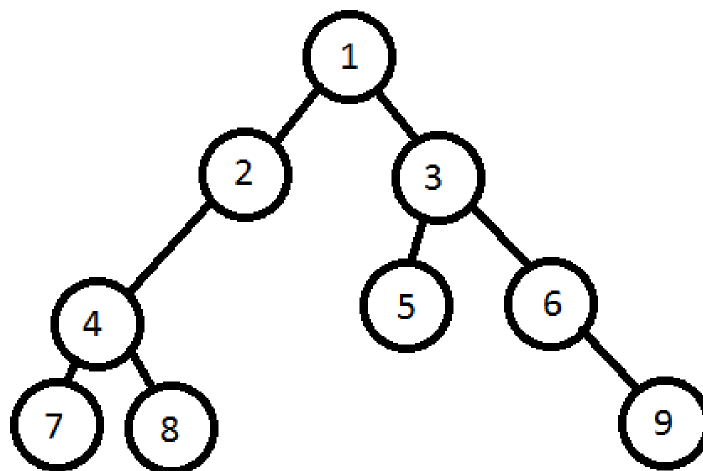
3.2 **Reitinetsintä**

Tässä esitellään kolmea eri algoritmia, joita käytetään peliohjelmoinnissa reittienetsintään. Nämä käydään läpi ensimmäisistä algoritmeista uusimpaan.

Reitinetsintää tehtäessä hakualue yksinkertaistetaan tekemällä hakualueesta neliöverkko. Tällä saadaan yksinkertaistettua hakualue kaksiulotteiseen taulukkoon. Jokainen solu esittää yhtä neliöverkon neliötä ja sillä on tieto, onko neliössä este vai voiko siihen kävellä. Reitinetsintä selvittää neliöt, joita pitkin pääsee paikasta A paikkaan B. Reitti kulkee neliöiden keskipisteestä seuraavan neliön keskipisteeseen, kunnes saavuttaa päätepisteen. Näitä keskipisteitä kutsutaan solmuiksi (node), koska on mahdollista jakaa hakualue muihinkin muotoihin kuin neliöihin. Ne voivat olla vaikkapa kolmioita tai heksagoneja ja solmut voivat olla missä tahansa kuvion kohtaa. (Lester 2005.)

3.2.1 Leveyssuuntainen läpikäynti

Vuonna 1959 Edward F. Moore kehitti leveyssuuntaisen läpikäynnin. Haku aloitetaan aloitussolmusta, josta edetään seuraavaan viereiseen solmuun. Tätä jatketaan niin kauan, kunnes aloitussolmun viereiset solmut loppuvat. Sen jälkeen aloitetaan tutkimaan jokaisen löydetyn solmun viereisiä solmuja. Kuten kuvassa 5 tätä jatketaan niin kauan, kunnes päätesolmu ja reitti on löydetty. (Khodakarami 2015.)



Kuva 5. Solmujen etsintäjärjestys leveyssuuntaisessa läpikäynnissä (Kuvakaappaus).

Khodakarami (2015) kertoo leveyssuuntaisen läpikäynnin olevan algoritmi joka käy ja tutkii jokaisen mahdollisen löytämänsä haaran. Se tutkii perus-

teellisesti jokaisen löytämänsä solmun, vaikka se ei olisi päätesolmun lähettyvillä. Jos reitinetsintään on mahdollinen ratkaisu, tämä algoritmi löytää lyhimmän reitin loppupisteeseen.

3.2.2 Dijkstran algoritmi

Dijkstran algoritmin kehitti Edsger Dijkstra vuonna 1959. Tämä eroaa leveyssuuntaisesta läpikäynnistä siten, että Dijkstran algoritmi löytää parhaimman reitin lyhyimmän reitin sijaan. Dijkstran algoritmi laskee jokaiselle solmulle annetun arvon. Kun algoritmi löytää useamman reitin, se laskee pisteiden perusteella parhaimman reitin, esimerkiksi jos reitin varrella on tiheää metsää ja metsä on hyvin vaikeakulkuista. Kauempaa kiertää polku, josta pääseminen on helpompaa. Metsässä olevilla solmuilla on suurempi arvo kuin polulla olevissa. Algoritmi laskee kaikki mahdolliset reitit, mutta lyhimmän reitin sijaan se valitsee parhaimman reitin pisteiden avulla. Vuonna 1984 Fredman - ja Tarjan-nimiset tutkijat muokkasivat algoritmia siten, että se poistaa reitit etsinnästä joilla on suurempi arvo kuin pienimmällä arvolla oleva reitti. (Khodakarami 2015.)

3.2.3 A*

A* eli A-tähti kehiteltiin vuonna 1968 Peter Hartin, Nils Nilssonin ja Bertram Raphaelin toimesta. A-tähti-algoritmi käyttää liikkumiseen *F = G + H*-kaavaa. Kyseinen algoritmi käyttää Dijkstran algoritmista otettua arvoa (G). Se käyttää myös toista arvoa joka on arvioitu liikkumisen hinta annetusta neliöstä loppuneliöön. Tätä kutsutaan heuristiikaksi, koska se on vain arvaus (H). (Lester 2005.)

Lesterin mukaan (2005) mainittu G on arvo aloitussolmusta siihen solmuun mihin on edetty. Jos neliön koko on 10*10, niin vinottainen etäisyys on d = √x2 + y2. Arvo olisi tällöin 14,14 mutta se pyöristetään lähimpään kokonaislukuun. Arvo on tarpeeksi lähellä tarkkaa vastausta ja tietokoneiden on nopeampaa laskea arvoja. G:n arvoon lisätään siis 10 tai 14, riippuen kulkeeko reitti pystysuoraan, vaakasuoraan vai viistosti.

Algoritmi lisää tutkitut neliöt ”avoimeen listaan”. Avoin lista sisältää ne neliöt jotka ovat mahdollisesti oikealla reitillä. Algoritmi vertailee tutkittujen solmujen viereisiä solmuja ja valitsee aina solmun, jolla on pienin F-arvo. (Lester 2005.)

3.3 **Rajoitukset**

Millingtonin ja Fungen (2006, 25) mukaan yleisin rajoitus pelien tekoälylle on fyysinen laite, jolla peliä pelataan. Tekoäly ei yleensä voi prosessoida mitä tehdä seuraavaksi montakaan päivää tai varata laitteen koko muistia

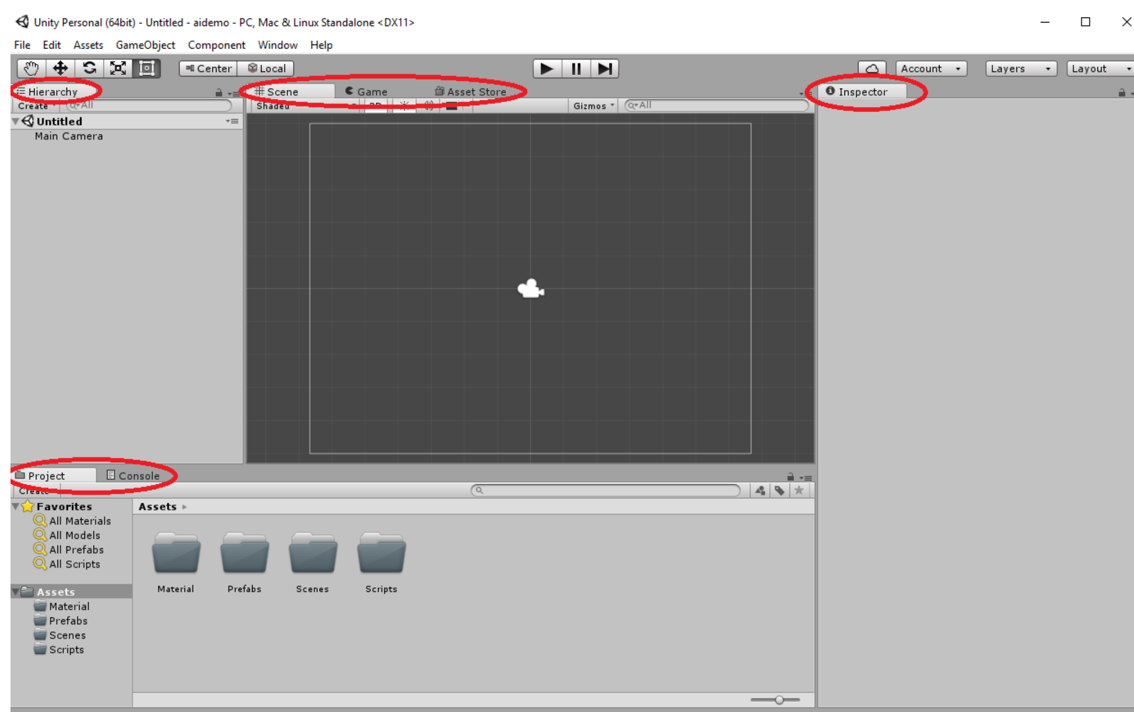
omaan käyttöönsä. Tästä syystä uudet tekoälyn tekniikat eivät saa pelikehittäjien huomiota. Pelien kehittäjät joutuvatkin suunnittelemaan tekoälyn miettien näitä asioita. Aiemmin grafiikka vei suuren osan prosessorin käytöstä, mutta nykyään entistä enemmän animaatioita ja grafiikkaa piirretään grafiikkaprosessorilla. Tämä on mahdollistanut peleissä kehittyneempää tekoälyä, kun prosessorin ei tarvitse enää piirtää grafiikkaa.

Konsolien kanssa työskentely on helpompaa kuin tietokoneiden. Konsolit ovat aina samanlaisia komponenteiltaan, toisin kuin tietokoneiden ja ne vaihtelevat melkein jokaisella pelaajalla. Grafiikka on helppo ottaa huomioon säätämällä grafiikan tarkkuutta tai varjoja heikompaan tai tarkempaan. Toisin kuin grafiikkaa, tekoälyä ei voida skaalata samaan tapaan. Yleisin ratkaisu on tehdä tekoäly heikompien tietokoneiden ehdoilla. Monissa peleissä voidaan skaalata tekoälyä, mutta ne eivät vaikuta pelaamiseen. Esimerkiksi voidaan poistaa käveleviä ihmisiä tai lentäviä lintuja. (Millington & Funge, 2006.)

4 UNITY-PELIMOOTTORI

Tässä luvussa esitellään Unity-pelimoottoria ja sen toimintoja. Käytössäni on Unity-versio 5.4.1f1. Unityn mukana asentuu MonoDevelop-koodauseditori. Itse käytän MonoDevelopin sijaan Visual Studio 2015:ta.

Unityllä tehdyt pelit koostuvat sceneistä. Scenejä voidaan käyttää pelin päävalikoksi, yksittäisiksi pelitasoiksi tai vaikka lopputeksteille. Ne sisältävät GameObjecteja. GameObject on Unity-pelissä oleva rakennuspalikka. Se voi olla esim. äänitiedosto, tekstitiedosto tai 3D-malli. Näihin objekteihin voidaan liittää erilaisia komponentteja.



Kuva 6. Unityn perusnäkymä (Unity 5.4.1f1 2017, kuvakaappaus).

Uuden projektin tehdessään tekijälle aukeaa tyhjä scene. Tässä vaiheessa scene sisältää vain yhden GameObjectin, Main Camera. Tämän objektin kamera kuvaa sitä mitä pelaajalle pelissä näytetään. Unityn yläreunasta löytyy valikoita, joilla voidaan lisätä sceneen uusia GameObjecteja, Asset-teja ja komponentteja.

Kuvassa 6 näkyy neljä eri ikkunaa. Scene-ikkunasta nähdään valitun scenen sisältö. Tämän näkymän voi myös vaihtaa Game-ikkunaan eli siihen josta nähdään, miten nykyinen scene näkyy itse pelissä. Hierarchy-ikkunasta voidaan valita eri scenejä. Valitun scenen sisältä löytyy GameObjectien keskinäinen hierarkia. Inspector-ikkunasta näkyvät valitun GameObjectin tarkemmat tiedot. Alhaalta löytyvät Project-ikkuna, sieltä löytyvät kaikki projektikansiot ja niiden sisältä löytyvät tiedostot. Consoleen eli konsoliin tulostuu kaikki erilaiset varoitukset ja virheilmoitukset. Kaikki ikkunat ovat käyttäjän muokattavissa mieluisaan järjestykseen.

5 TAVOITTEET

Ensiksi asennan Unityn Asset Storesta löytyvän reitinetsintälaajennuksen ja Unityn oman reitinetsintä ohjelman. Lisään laajennuksen itse tekemääni sokkelokarttaan ja koitan saada pallon liikkumaan paikasta A paikkaan B. Lopuksi vertailen näitä kahta eri reitinetsintälaajennusta keskenään, mihin niitä on hyvä käyttää ja miksi. Tämän jälkeen tavoitteenani on tehdä Unity-pelimoottorilla muutamia demonstraatioita tekoälystä peliohjelmoinnissa. Koodikielenä tulen käyttämään C#-kieltä. Työssäni käytän erittäin yksinkertaistettua grafiikkaa, sillä grafiikalla ei ole suurta merkitystä työn toteutuksessa.

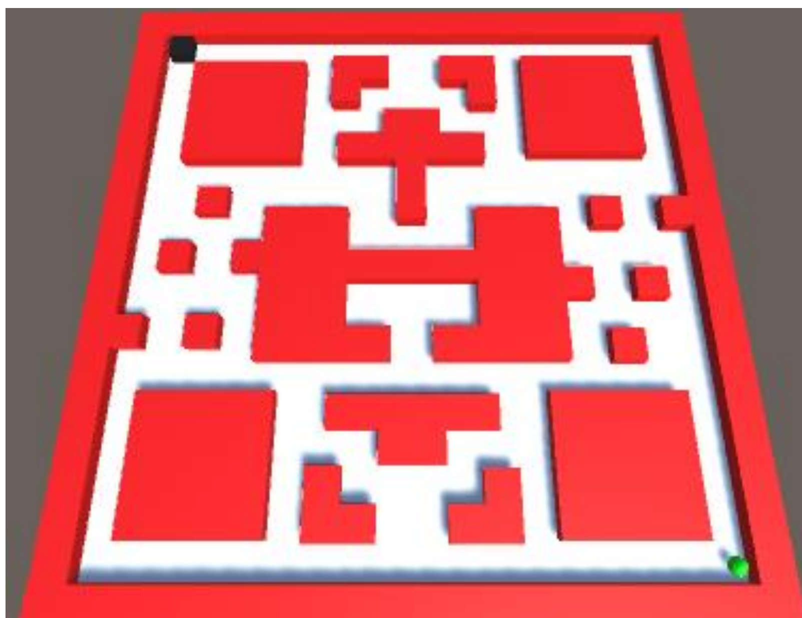
Käytännöllisenä työnä tulen myös toteuttamaan tasohyppely-peliin soveltuvalle hahmolle tekoälyn liikkumista varten. Tähän liittyy hahmon kävelemissuunnan vaihtaminen, jos edessä on seinä tai liian suuri kuilu, jonka yli ei pysty hyppäämään. Tulen myös tekemään hahmolle kyvyn hypätä kuilun ylitse, jos tämän vauhti riittää sen ylitykseen. Viimeisenä tulen tekemään vihollishahmon hyökkäämisen pelaajaa kohden, jos vihollishahmo näkee pelaajan.

6 PELITEKOÄLYN TOTEUTTAMINEN

Tässä kappaleessa esittelen muutaman pelitekoälymenetelmän, joita voidaan käyttää peliohjelmoinnissa. Tekemäni esimerkit ovat grafiikaltaan hyvin yksinkertaisia, sillä niiden tarkoitus on vain esitellä tekoälyn toimintaa.

6.1 Reitinetsintä sokkelossa

Ensimmäinen esimerkki on yksinkertainen sokkelo, joka koostuu sokkelosta, pallosta ja maalialueesta. Tällä demonstroin, kuinka pallo saadaan tekoälyllä liikkumaan paikasta toiseen, väistellen matkalla olevia esteitä ja kulmia. Ideana on saada vihreä pallo eli pelaaja, liikkumaan sokkelon lävitse mustan väriselle maalialueelle.



Kuva 7. Kolmiulotteinen sokkelokartta (kuvakaappaus).

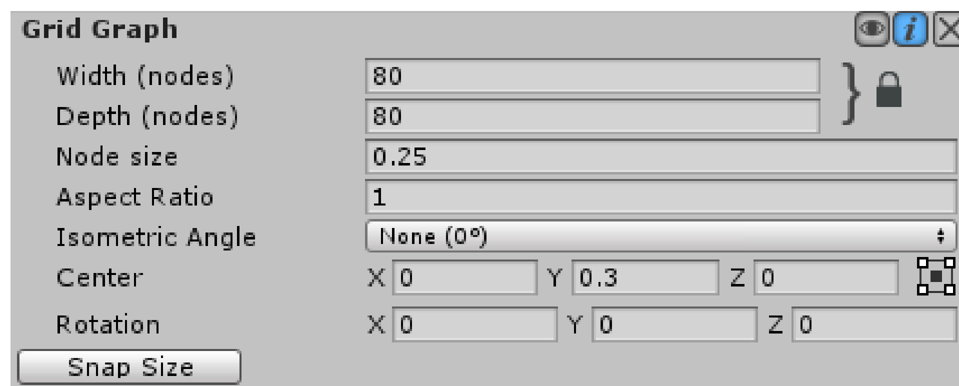
Kuvasta 7 havaitaan, että esimerkissä käytetään radan luontiin vain erikokoisia suorakulmioita. Niitä saa kätevästi luotua ja muokattua suoraan Unityn sisällä, ilman muita 3D-piirto- tai mallinnusohjelmia. Demon grafiikkaa olisi voinut parannella asentamalla Unity Asset Storesta löytyviä 3D-malli laajennuksia, mutta työni ei tarvinnut paremman näköistä grafiikkaa toimiakseen.

6.1.1 Reitinetsinnän toteuttaminen A* Pathfinding projectilla

Ennen työn aloittamista pitää päättää minkälaista reitinetsintäalgoritmia haluaa käyttää. Kaikki nämä algoritmit eivät sovellu kaikkiin ongelmiin. Tässä työssä monimutkaiselle reitinetsinnälle ei ole suurta tarvetta. En myöskään tuota algoritmia itse.

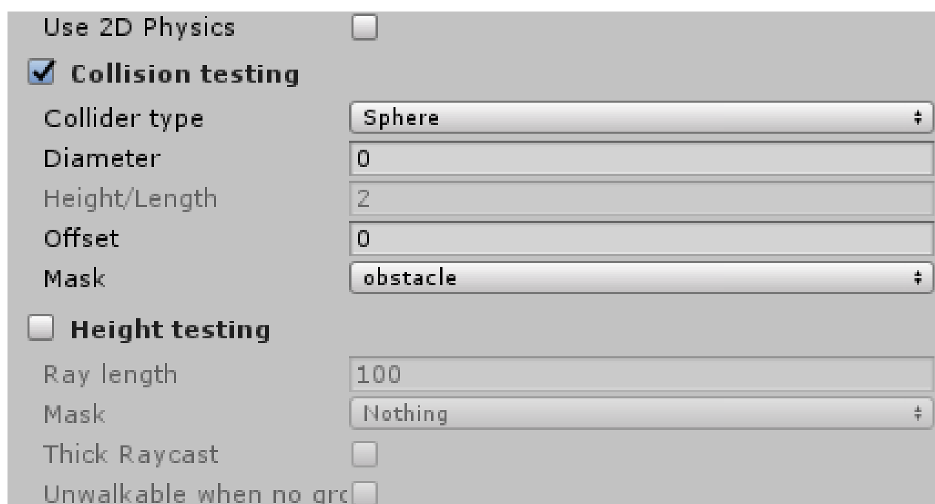
Työssä käytän reitinetsimiseen A*-algoritmia. Algoritmia en tuota itse vaan käytän valmiiksi tehtyä projektia laajennuksena omaan työhöni.
Valitsin Unity Asset Storesta löytyvän Aron Granbergin tuottanut *A* Pathfinding Project* -ilmaisversion. Maksullisessa versiossa olisi ollut enemmän ominaisuuksia ja hienosäätöjä, mutta tähän työhön tarvittavat ominaisuudet löytyivät ilmaisesta versiosta.

Toteuttaminen aloitetaan sillä, että laajennus lisätään (*import*) projektiin ja lisätään Unityyn kaksi uutta layeria: lattialle ja seinille omansa. Layer kertoo tekoälylle mitkä peliobjektit eivät ole esteitä ja mitkä ovat. Tämän jälkeen sceneen luodaan uusi tyhjä peliobjekti, johon lisätään ladatun laajennuksen mukana tullut reitinetsintäkomponentti. Jotta tekoäly rupeaisi löytämään reittejä, täytyy vielä luoda pisteverkosto. Pisteverkosto saadaan luotua valitsemalla juuri tehneen peliobjektin inspector-ruudusta ”*Add New Graph*”, jonka sisältä valitaan yksinkertainen ruudukkoverkko (Grid graph). Inspector-ruutuun aukeaa suuri lista säädeltäviä asetuksia. Niitä muokkaamalla saadaan pelihahmot kulkemaan mahdollisimman tarkasti. Tärkeimpiä asetuksia ovat ruudukon sijainti ja koko sekä *Collision testing* eli törmäyksien asetukset. Peliruudukko koko on 20x20 ruutua, joten sille riittää samankokoinen pisteverkko. Itse muunsin pisteverkon kokoa 80x80een. Jos pisteverkon kokoa muuttaa isommaksi, täytyy myös muuttaa ”N*ode size*” eli solmujen kokoasetusta, jotta pisteverkko olisi samankokoinen kuin pelikenttä. Tällöin hahmon liikkumisesta saadaan sulavampaa ja luonnollisemman näköistä, kun pisteitä on tiheämmin. Kuvassa 8 näkyvät asettamani ruudukon koko ja sijainti. On myös kannattavaa asettaa pisteverkko hieman korkeammalle kuin lattian taso. Tällöin saadaan minimoitua peliobjektien välisiä komplikaatioita.



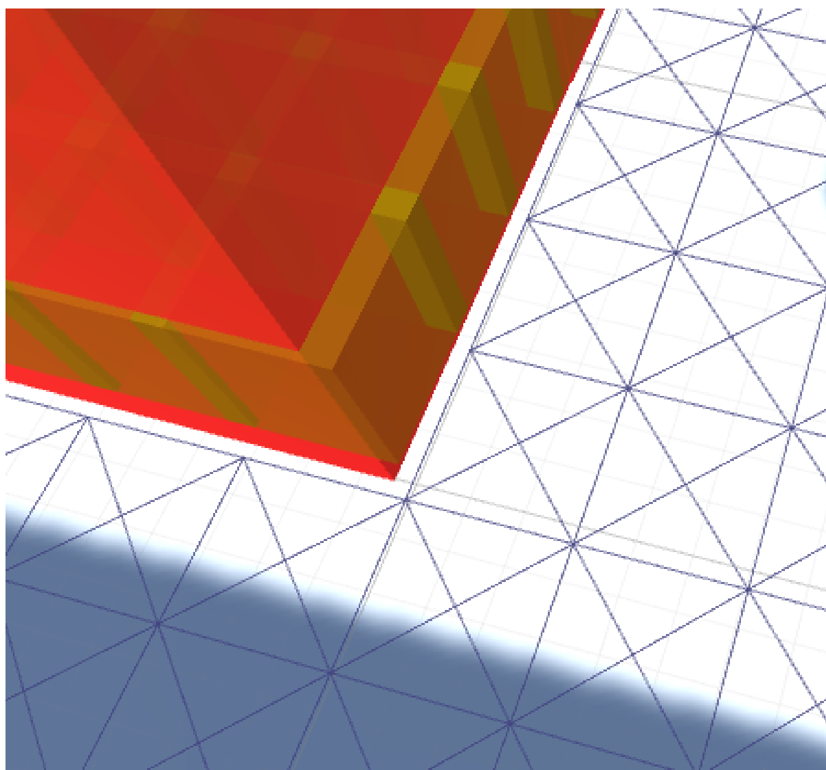
Kuva 8. Pisteverkostoinnin koko- ja sijaintiasetuksia, kuvakaappaus.

Collision testing on reitinetsinnän kannalta tärkeä asetus. Tekoäly testaa sen avulla, mitkä pisteverkoston pisteistä ovat kulkukelvottomia. Työssäni tekoälyn ohjaama peliobjekti on pallo, *collider typeksi* voidaan valita *Sphere*. Kuvassa 9 näkyy Diameter-asetus, joka määrittelee testauksessa käytettävien esteiden halkaisijan. *Height testing* -valikosta voidaan valita eri lattiatasoille asetuksia reitinetsintää varten. Tässä työssä ei ole kuin yksi taso jolla pallo liikkuu, joten sitä asetusta ei tarvitse ottaa käyttöön.



Kuva 9. Pisteverkon törmäys- ja korkeusasetuksia, kuvakaappaus.

Asetuksien säätelyn jälkeen, voidaan reittienluomista testata painamalla *Scan*-nappia, joka sijaitsee *Inspector*-ruudun alareunassa. Unityn pelinäkymään pitäisi nyt ilmestyä pisteverkosto, niihin alueille missä hahmo pystyy liikkumaan. Esteiden sisälle myös ilmestyy kuutioita, jotka merkkaavat mihin hahmon ei pitäisi pystyä kävelemään. Kuvassa 10 näkyvät ohuilla sinisillä viivoilla pisteverkosto, jota pitkin hahmo liikkuu ja seinien sisällä näkyvät keltaisia kuutioita, jotka merkkaavat alueet joilla ei voida kävellä.



Kuva 10. A* reitinetsinnän pisteverkosto, kuvakaappaus.

Jotta hahmo lähtisi liikkumaan pisteverkossa, pitää tälle vielä lisätä ensimmäisenä *Seeker*-skripti, joka tuli laajennuspaketin asennuksen yhteydessä.

Hahmoon tarvitsee lisätä vielä toinen skripti, jossa asetetaan loput hahmon tarvitsemat tiedot ja kutsutaan *Seeker*-skriptiä etsimään reitti määränpäähän. Tätä ennen hahmoon lisätään *Character Controller* -komponentti, jotta hahmoa pystytään ylipäänsä liikuttamaan.

Pelistä löytyy nyt valmis pisteverkosto ja hahmolle on annettu *Seeker*-skripti komponentti, sille pitää vielä tehdä viimeinen skripti toimiakseen. Tätä varten tehdään uusi C#-skripti, jossa hahmo käsketään liikkeelle. Ennen koodin kirjoittamista, skriptin alkuun lisätään *”Using Pathfinding;”.* Tällä lauseella saadaan A* Pathfinding laajennuksen ominaisuudet käyttöön koodissa. Tämän jälkeen koodiin voidaan kirjoittaa tarvitsemansa asiat, riippuen minkälaista itse tarvitsee. Esimerkissä 1 *seeker.startPath* metodilla saadaan hahmon sijainti (transform.position) ja määränpään sijainti (target.position). Reitinlaskenta tapahtuu *Start()*-funktion sisältä. Kun ohjelma käynnistetään, ohjelma laskee reitin ensimmäisenä. *OnPathComplete* metodia kutsutaan lopuksi, jossa tarkistetaan, löytyikö pisteiden välistä kuljettava reitti.

```
void Start(){
seeker = GetComponent<Seeker>();
seeker.StartPath(transform.position, target.position,
             OnPathComplete);
 characterController = GetComponent<CharacterController>();
}

public void OnPathComplete (Path p)
{
             if (!p.error)
             {
             path = p;
             currentWaypoint = 0;
             }
}
```

Esimerkki 1. reitin luominen.

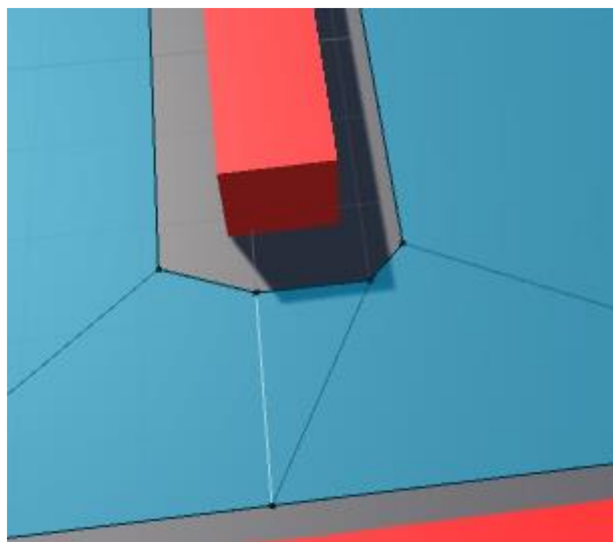
Skripti tallentaa löydetyn reitin *”path”*-muuttujaan. Enää tarvitaan koodi, joka saa hahmon liikkumaan pisteverkon pisteiden välejä. Hahmo siis vain etsii suunnan, missä on seuraava piste ja liikkuu sitä kohti koodissa määrätyllä nopeudella. Esimerkissä 2. näkyy ensiksi koodinpätkä, joka saa pallon liikkumaan. Alempana olevalla *if*-lauseella estetään hahmon liiallinen pisteiden välinen ”hyppeleminen”. Jos pisteverkossa on liian tiheästi pisteitä, hahmo saattaa oikaista kulmien tai seinien läpi, jolloin se jää jumiin.

```
Vector3 dir = (path.vectorPath[currentWaypoint]
-transform.position).normalized * speed;
characterController.SimpleMove(dir);

if (Vector3.Distance(transform.position, path.vectorPath
[currentWaypoint]) < maxWaypointDistance)
             {
             currentWaypoint++;
             }
```

Esimerkki 2. Hahmon liike koodissa.

6.1.2 Toteuttaminen Unity Navigation -ominaisuudella

Reitinetsinnän pystyy myös toteuttamaan Unityn sisäänrakennetulla komponentilla nimeltä ”*Navigation*”. Navigaatio-ikkunan saa auki painamalla ylhäältä Windows ja valitsemalla ”*Navigation”.* Ikkuna aukeaa Inspector-ikkunan päälle. Ensiksi täytyy luoda lattia, esteitä, pelaaja ja määränpää.



Kuva 11. Unity Navigation, kuvakaappaus.

Navigaatio-ikkunasta valitaan *”object*” ja valitaan scene-ikkunasta lattia. Navigaatio-ikkunaan tulee näkyviin lattian peliobjekti, jolle asetetaan ”*Navigation Static”* ja *”Navigation Area”* valikosta valitaan *Walkable* eli käveltävä alue. Tämän jälkeen valitaan kaikki seinät joiden läpi ei haluta kävellä. Jos pelissä on seiniä sellaisilla alueilla missä pelaaja ei pysty kulkemaan, niin silloin niitä seiniä ei kannata valita, sillä ne vievät turhaan muistia ja voivat hidastaa peliä. Navigation ikkunasta valitaan jälleen *Navigation Static,* mutta tällä kertaa *Navigation Areasta* valitaan *Not Walkable*, eli alue joiden läpi ei voida kulkea. Kuvassa 11 sinisellä värjätty alue esittää missä hahmo voi kulkea vapaasti.

```
public Transform targetPoint;
void Update ()
{
transform.GetComponent<NavMeshAgent>().destination =
targetPoint.position;
}
```

Esimerkki 3. Hahmon liikkuminen koodissa.

Jotta hahmo saataisiin liikkeelle, pitää sille asettaa *NavMeshAgent*-komponentti. Tällä komponentilla voidaan asettaa nopeus-, kääntyvyys-, tai esteiden kiertämisasetuksia. Viimeiseksi tehdään uusi C#-skripti. Esimerkissä

3 ensimmäisenä skriptiin alustetaan muuttuja määränpäälle. Tämän jälkeen koodilla määrätään hahmo liikkumaan määränpään nykyiseen sijaintiin.

6.1.3 Vertailu

Tehdessäni molempia esimerkkejä, huomasin molemmissa vaihtoehdoissa olevan hyviä, sekä huonoja puolia. Aron Granbergin tekemässä A* Pathfinding Project -laajennusta asentaessani ensimmäistä kertaa, huomasin että siinä oli ehkä turhankin paljon pieniä säädettäviä asetuksia tähän yksinkertaiseen esimerkkiin, jonka tuotin. Tutkiessani laajennusta tarkemmin, löysin sieltä valmiita skriptejä, joilla voitaisiin esimerkiksi hiljentää vauhtia tullessa mutkaan tai tehdä kaarrevamman käännöksen. Löysin myös skriptin, joka sopisi hyvin Real-Time Strategy-peleihin tai tornipuolustus peleihin. Jos haluttaisiin liikuttaa montaa hahmoa samaan aikaan paikasta toiseen, niin tällä kyseisellä skriptillä saataisiin jokaiselle hahmolle hieman eroava reitti toisistaan, jottei kaikki kulkisi peräkkäin täysin samaa reittiä. Tämä laajennus on erittäin laaja ja monipuolinen, ollakseen ilmaisversio. Tämän laajennuksen käyttö ei ehkä ole suotavaa pieniin projekteihin tai todella yksinkertaisiin reitinetsinnän käyttöön, vaan laajempiin ja monimutkaisempiin tapauksiin.

Unityn Navigationin sai nopeasti käyttöön muutamalla klikkauksella. Tällä tyylillä onnistui vaivattomasti käveltävien alueitten merkkaus, ja Unity osasi asettaa esteille, sekä hieman niiden ympärille kävelemättömät alueet. Lopuksi hahmolle annettiin valmiina oleva komponentti, josta löytyi asetuksia nopeuteen ja kääntyvyyteen. Näitä asetuksia ei tarvinnut ollenkaan muokata, että reitinetsintä lähti toimimaan. Tämä olisi hyvä joihinkin pienempiin projekteihin, sillä se on nopea ja helppo asentaa, mutta selvästi vaikeampi lähteä itse tekemään omia skriptejä, jos haluaa saada hieman monimutkaisempaa reitinetsimistä.

6.2 **Tasohyppely-esimerkki**

Tässä esittelen Unityssä tekemäni hahmon, joka kävelee eteenpäin niin kauan, kunnes vastaan tulee seinä tai kuilu. Ensimmäisenä aloitin tekemällä uuden Unity 2D projektin. Kuten kuvasta 12 huomaa, tämän jälkeen tuotin hahmon ja lattian grafiikan käyttämällä Adoben Photoshop -ohjelmaa. Nämä saadaan Unityyn yksinkertaisimmillaan vain vetämällä halutut muodot (spritet) Unity-editoriin, jossa voidaan nimetä muodot halunsa mukaan esimerkiksi, lattia ja vihollinen.

Kuva 12. Kuvakaappaus tämän hetkisestä tilanteesta. Lattia, seinä ja hahmo.

Tämän jälkeen hahmolle, seinälle ja lattialle annetaan komponentteja. Seinille ja lattioille annetaan Box Collider 2D, joka on neliskanttisenmuotoinen. Collider määrittelee spriteille reunat, jolloin kyseinen muoto on interaktiivinen muille spriteille, joilla on myös collider. Annoin myös työstämälleni hahmolle Box Collider 2D:n, sekä Rigidbody 2D:n. Rigidbody on Unityn sisällä oleva fysiikkamoottori. Sillä saadaan vaivattomasti luotua hahmoille paino, taikka painovoiman vahvuutta.

Seuraavaksi tein hahmolle skriptin nimeltä *Enemy.cs.* Skriptiin olen tehnyt hahmolle liikkumisen. Hahmo liikkuu oikealle niin kauan, kunnes se törmää seinään tai lattia loppuu hahmon edestä. Tämän tapahtuessa hahmo vaihtaa suuntaansa ja samaan tapaan kääntyy taas, kunnes eteen tulee seinä tai kuilu.

```
public float velocityX = 1f;
public Transform sight;
public Transform floorSight;
public bool colliding;
public bool CollidingFloor;

colliding =
Physics2D.Linecast(sight.position, sight.position);
CollidingFloor =
Physics2D.Raycast(floorSight.position, floorSight.position);

if (colliding)
{
transform.localScale =
new Vector2(transform.localScale.x * -1, transform.localScale.y);
velocity *= -1;
}

if (CollidingFloor == false)
{
transform.localScale =
new Vector2(transform.localScale.x * -1, transform.localScale.y);
velocity *= -1;
}
```

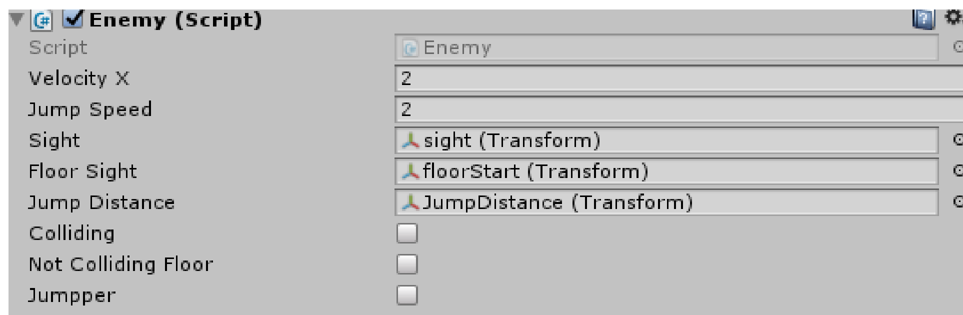
Esimerkki 3. Hahmon suunnanvaihto skriptissä.

Asetin hahmon eteen näkymättömiä peliobjekteja, jotka toimivat sensoreina ja tarkistavat osuuko hahmo lattiaan, seinään tai ei mihinkään. Esimerkissä 3 bool-muuttujalle *"colliding"* on asetettu näkymätön peliobjekti hahmon puoliväliin, jottei se osuisi maahan ja luulisi, että edessä on seinä. Sillä siis tarkistetaan törmääkö hahmo seinään, ja jos törmää hahmo kääntyy ympäri. Toinen bool-muuttuja "CollidingFloor" tarkistaa, loppuuko hahmon edestä lattia ja jos loppuu, hahmo kääntyy ympäri ja jatkaa kävelyä. Tälle muuttujalle on asetettu tyhjä peliobjekti hahmon eteen, osuen lattiaan.

```
public float jumpSpeed = 5f;
public Transform jumpDistance;
public bool jumpper;

Rigidbody2D jump = GetComponent<Rigidbody2D>();
jumpper = Physics2D.Raycast(jumpDistance.position,
             jumpDistance.position);

if (CollidingFloor == false && jumpper)
{
jump.velocity = new Vector2(jump.velocity.x, jumpSpeed);
}

else if (CollidingFloor == false)
{
transform.localScale =
new Vector2(transform.localScale.x * -1, transform.localScale.y);
velocityX *= -1;
}
```

Esimerkki 4. Hahmon hyppäämisen ja kääntymisen tehtyä koodia.

Jotta hahmon saisi vielä hyppäämään kuilujen ylitse, tarvitsee koodiin lisätä hyppäämiselle vauhti, tyhjä peliobjekti ja bool-muuttuja. Tyhjä peliobjekti lisätään hahmolle ja se asetetaan hahmon eteen, niin pitkälle kuin halutaan hahmon hyppäävän. Jotta hahmo osaisi katsoa onko edessä pelkkä kuilu vai voiko sen yli hypätä päästen laskeutumaan lattialle, äskettäin lisätyllä tyhjällä peliobjektilla tarkistetaan, onko hypyn määränpäässä lattiaa mihin laskeutua. Esimerkissä 4 esitettään ettei hahmo hyppisi koko ajan tai vaihda suuntaa aina kuilun tullessa vastaan, tarvitaan *if*-lausetta tarkistamaan, osuuko lattiaa tutkiva sensori lattiaan ja hypyn määränpäätä tutkiva sensori myös lattiaan. Jos lattian sensori ei osu lattiaan ja hypyn määränpää osuu, niin silloin hahmo hyppää. Jos hypyn määränpäässä ei ole lattiaa, silloin hahmo vain kääntyy tulessaan kuilun kohdalle. Kuvassa 13 näkyvät asetetut liikkumis- ja hyppynopeudet, myös sensoreina toimivat peliobjektit on asetettu nimillä, jotteivat ne sekoitu myöhemmin.



Kuva 13. Unityssä asetetut tyhjät peliobjektit, kuvakaappaus.

6.3 Hyökkäys-esimerkki

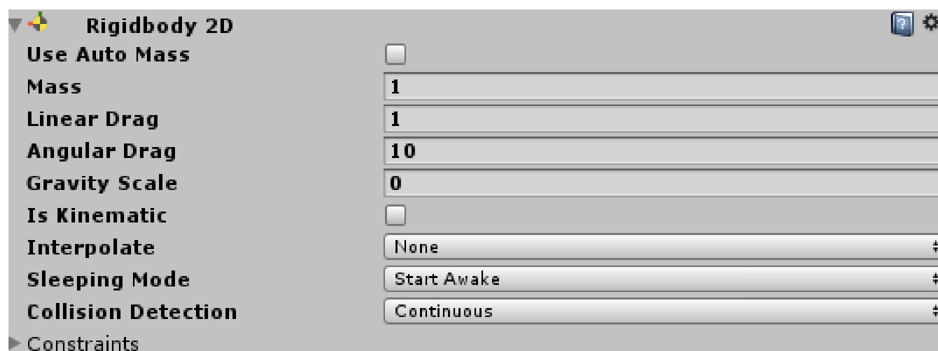
Tässä esimerkissä tuotan ylhäältäpäin kuvatun pelitilanteen, jossa vihollishahmo lähtee jahtaamaan pelaajaa, pelaajan tullessa tarpeeksi lähelle vihollishahmoa. Vihollishahmo jahtaa pelaajaa niin kauan, kunnes pelaaja pääsee tarpeeksi kauas vihollisesta. Kuvassa 14 näkyvät hahmojen spritet, jotka latasin *MillionthVectorilta.* Vihollishahmon väriä muokkasin punaiseksi Adoben Photoshop -ohjelmalla, erottuakseen pelaajan hahmosta. Lisäsin tähän esimerkkiin myös reunat, jotta pelihahmot eivät karkaisi liian pitkälle. Hahmoille lisäsin BoxCollider2D-komponentit, jotta ne eivät karkaisi pelialueen ulkopuolelle.

Kuva 14. Pelihahmot, kuvakaappaus.

```
public float speed;
public float rotateSpeed;
void Update () {
if (Input.GetKey(KeyCode.A))
{
transform.Rotate(Vector3.forward * rotateSpeed * Time.deltaTime);
}
if (Input.GetKey(KeyCode.D))
{
transform.Rotate(-Vector3.forward * rotateSpeed * Time.deltaTime);
}
if (Input.GetKey(KeyCode.W))
{
GetComponent<Rigidbody2D>().AddForce(transform.up * speed *
Time.deltaTime);
}
}
```

Esimerkki 5. Pelaajan liikkuminen koodissa.

Ensimmäisenä pelaajalle tarvitsee tehdä liikkumiseen skripti. Skriptin nimesin *Player.cs*. Esimerkissä 5 skriptiin asetetaan muuttujat liikkumiselle ja hahmon kääntymisnopeudelle. Seuraavaksi hahmolle tehdään kääntyvyys asettamalla näppäimille A ja D. Kun painaa näppäintä A, hahmo kääntyy paikallaan vasempaan ja näppäimellä D, hahmo kääntyy paikallaan oikeaan suuntaan. Jotta hahmo ei vain pyörisi paikallaan, näppäimellä W annetaan hahmolle vauhtia. Unityn *Inspector* näkymästä täytyy muuttaa vielä Rigidbody2D asetuksia, jotta hahmo liukuisi eteenpäin vauhtia hiljentäessä, eikä jatka matkaa ikuisesti eteenpäin.



Kuva 15. Unityn Rigidbody 2D-asetuksi, kuvakaappaus.

Kuten kuvasta 15 näkee, on *"Linear Drag"* eli suoraviivainenjarrutus-asetuksen arvo on hyvä pitää vähintään yhdessä. Jos asetus asetetaan nollaan, niin alus ei pysähdy ennen kuin törmää johonkin esteeseen. *"Angular Drag"* eli kääntyvyysjarrutusta käytetään, jos halutaan hidastaa peliobjektin kääntymisenvauhtia. Mitä suurempi asetuksen arvo on, sitä enemmän peliobjekti jarruttaa kääntyessään. Tässä demonstraatiossa ei ole käyttöä kääntyvyysjarrutukselle, sillä hahmot kääntyvät eri tavalla mihin asetusta tarvittaisiin.

```
public GameObject player;
private Vector3 offset;
void Start()
{
offset = transform.position - player.transform.position;
}
void Update()
{
transform.position = player.transform.position + offset;
}
```

Esimerkki 6. Skripti, jolla kamera seuraa pelaajaa.

Pelialue saattaa olla suurempi mitä kamera voi kerralla näyttää ruudulle. Kamera-peliobjektin voisi joissain tapauksissa liittää suoraan pelaaja-peliobjektiin, mutta esimerkiksi tässä tapauksessa kamera pyörisi ympyrää

pelihahmon liikkumis-algoritmin takia. Esimerkissä 6 asetetaan pelaaja-peliobjekti ja offset-muuttuja. Pelin alkaessa *"void Start()"*-funktio hakee pelaajan sen hetkisen sijainnin ja *"void Update()"*-funktio päivittää pelaajan sijaintia jokaisella ruudunpäivityksellä. Nyt kamera seuraa pelaajaa jatkuvasti kameran pysyen koko ajan suorassa.

```
public Transform Player;
Vector3 dir = Player.position - transform.position;
float angle = Mathf.Atan2(dir.y, dir.x) * Mathf.Rad2Deg;
transform.rotation = Quaternion.AngleAxis(angle - 90f,
Vector3.forward);
```

Esimerkki 7. Vihollishahmon kääntyminen pelaajaa kohti.

Jotta vihollishahmon liikkuminen näyttäisi luontevammalta, voidaan avuksi käyttää *Hackingia.* Tein vihollishahmolle skriptin *Enemy.cs.* Esimerkissä 7 saadaan vihollishahmon keula suuntaamaan aina kohti pelaajaan sijaintia. Skriptissä tarvitsee asettaa *"Quaternion.AngeAxis"*-luokalla, minkä kyljen haluaa suuntaavan pelaajaa kohden. Hahmolle täytyi asettaa -90 asteen kulma, että se näyttäisi menevän suoraan.

```
public float MoveSpeed;
public float distance;
Rigidbody2D rgbd = GetComponent<Rigidbody2D>();
RaycastHit2D hit = Physics2D.Linecast(transform.position,
Player.position);
if (hit.collider != null)
{
distance = Vector2.Distance(transform.position, hit.point);
}
if (distance > 1 && distance < 15)
{
Debug.Log("Hyökkäykseen");
rgbd.AddForce(transform.up * MoveSpeed * Time.deltaTime);
}
```

Esimerkki 8. Vihollishahmon liikkuminen pelaajaa kohti.

Vihollishahmo ei vielä liiku pelaajaa kohden, joten esimerkissä 8 Enemy.cs-skriptiin lisätään muutama muuttuja. Toiseen asetetaan liikkumiselle nopeus ja toiseen lasketaan pelaajan ja vihollisen välinen matka. Jos *hit.collider* ei osu pelaajaan, silloin *distance*-muuttujaan lasketaan vihollisen ja pelaajan välinen matka. Tämän jälkeen tarkistetaan, onko pelaaja jo kiinni vihollisessa tai liian kaukana. Jos pelaaja saapuu 15 yksikön päähän vihollisesta, vihollinen lähtee hakeutumaan pelaajaa kohti jatkuvasti niin kauan, kunnes pelaaja pääsee yli 15 yksikön päähän vihollisesta. Viholliselle asetin nopeus ja *"Linear Drag"*-asetukset hieman erilaiseksi kuin pelaajalle. Vihollisen nopeus on hieman hitaampi kuin pelaajan, mutta *"Linear Drag"*-asetuksen asetin 0.5, jolla saadaan hitaampi jarrutus aikaiseksi.

7 YHTEENVETO

Unityllä pelien tekeminen on ollut pitkään mielessä ja tämä opinnäytetyö antoi nyt mahdollisuuden tutustua syvemmin Unityyn. Opin itse paljon tekoälyn historiasta ja synnystä. Videopeleistä ja peliohjelmoinnista opin myös paljon asioita, joista en ollut ennen kuullutkaan. Teoriaosuuden työstämisen aikana kiinnostuin entistä enemmän aiheesta.

Käytännönosuutta tehdessäni huomasin sen olevan osittain vaikeaa, mutta tutkimalla ja miettimällä ratkaisuvaihtoehtoja, sain vastauksia aikaiseksi. Unityn käyttöliittymäkin tuli paljon tutummaksi, sillä se on aluksi aloittelijoille hieman sekava. Järjestelmällisyys auttoi projektin hallinnassa ja asioiden etsimisessä projektin sisällä.

Omat tavoitteeni olivat saada tuotettua jonkinlaista tekoälyä, jota voitaisiin käyttää peliohjelmoinnissa. Esimerkkini ovat kuitenkin melko suppeita, verrattuna siihen mihin tekoäly oikeasti pystyisi. Mutta ne kyllä ajavat asiansa tässä työssä. Omasta mielestäni pääsin hyvin tavoitteeseeni. Tutkimuskysymykseni olivat: kuinka tekoälyä käytetään peliohjelmoinnissa, kuinka reitinetsintää käytetään ja tuotetaan Unity-pelimoottorissa? Reitinetsintää tutkin hieman enemmän koko opinnäytetyön aikana, sillä reitinetsintä on suuressa osassa nykypäivän peleissä. Tutkin kolmea eri reitinetsintäalgoritmia. Parhaaksi reitinetsintäalgoritmiksi löysin A*-algoritmin. Opinnäytetyöhön tuotinkin kaksi erilaista reitinetsintään perustuvaa esimerkkiä. Molempiin tutkimuskysymyksiini sain vastauksen teoria- ja käytännönosuuden aikana.

Ennen työn alkua, ajattelin tekoälyä peliohjelmoinnissa paljon pienemmäksi aiheeksi mitä se oikeasti on. Tekoälyä käytetään nykypäivänä kaikenlaisissa peligenressä, tavalla tai toisella. Tekoäly kehittyy nykypäivänä hurjalla vauhdilla. Kehitystä on ennen rajoittanut tietokoneiden tehot.

Tavoitteeni opinnäytetyössä onnistui ja tulen varmasti jatkamaan peliohjelmointia harrastuksena vapaa-aikana.

LÄHTEET

AI Game Programmers Guild. 2011. History Of Game AI. gameai.com.
http://gameai.com/wiki/index.php?title=Game_AI_Eras
Luettu: 18.1.2017

Anna-Leena Heino. 2003. Tekoäly.
http://docplayer.fi/14138685-1-johdanto-i-1-taustaa-tekoalyn-historia-2-logiikka-tietamys-ja-paattely.html
Luettu: 16.1.2017

Khodakarami A. (2015). AI Theory - 001 - Basics - Path Finding - BFS, Dijkstra, A*. Haettu 3.2.2017 http://www.youtube.com

Kurenkov, Andrey. 2016. A ”Brief” History of game AI up to AlphaGo, Part 1. 18.4.2016
http://www.andreykurenkov.com/writing/a-brief-history-of-game-ai/
Luettu: 22.1.2017

Millington, I. & Funge, J. 2006. *Artificial intelligence for games*. Morgan Kaufmann Publishers.

Patrick Lester. 2005. A* Pathfinding for Beginners.
http://www.policyalmanac.org/games/aStarTutorial.htm
Luettu: 3.2.2017

The Turing Test. 1999. Alan Turing and the Imitation Game
http://www.psych.utoronto.ca/users/reingold/courses/ai/turing.html
Luettu: 18.1.2017

William Stewart. 7.1.2000. Dartmouth Artificial Intelligence (AI) Conference.
http://www.livinginternet.com/i/ii_ai.htm
Luettu: 18.1.2017

Xu, Siyuan. 2008. History of AI design in video games and its development in RTS games. WWW-dokumentti.
https://sites.google.com/site/myangelcafe/articles/history_ai
Luettu: 22.1.2017