TEKOÄLY UNITYSSÄ

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Ammattikorkeakoulututkinnon opinnäytetyö

Hämeenlinnan korkeakoulukeskus, tietojenkäsittelyn koulutusohjelma

hyväksymislukukausi, 2020

Juho Luoto

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Tietojenkäsittelyn koulutusohjelma
Hämeenlinnan korkeakoulukeskus

Tekijä Juho Luoto **Vuosi** 2020

Työn nimi Tekoäly Unityssä

Työn ohjaaja/t Tommi Lahti

TIIVISTELMÄ

Opinnäytetyön aiheena oli tekoälyn käyttäminen Unityssä. Tavoitteina opinnäytetyössä oli tutustua tekoälyyn ja reitinhakualgoritmeihin. Tämän jälkeen piti Unity-pelimoottorilla luoda demonstraatiosovellus, joka käyttäisi Unityn NavMesh-työkaluja kahdella pelikentällä olevalla hahmolla, jotka leikkivät hippaa keskenään.

Opinnäytetyön teoriaosuudessa käydään lyhyesti läpi tekoälyä ja sen käyttökohteita, jonka jälkeen kerrotaan enemmän suosituimmista reitinhakualgoritmeista ja niiden käyttökohteista. Viimeisenä teoriaosuudessa käsitellään Unity-pelimoottoria ja sen ominaisuuksia.

Opinnäytetyötä valmistaessa todettiin, että Unityn valmiita NavMesh-ominaisuuksia käyttämällä on yksinkertaista ja helppoa saada tekoälyhahmo liikkumaan pelikentällä antamalla sille vain pisteitä kartalla, joihin se pyrkii liikkumaan. Todettiin myös, että Unityssä olisi huomattavasti hankalampaa lähteä rakentamaan tekoälyjärjestelmää, joka käyttäisi jotakin muuta tapaa reitinhaussa, eikä syitä olla käyttämättä NavMeshiä tullut mieleen.

Alkuperäisiin tavoitteisiin päästiin ja demonstraatiosovellus toimii halutulla tavalla. Kumpikin tekoälyllisistä hahmoista käyttäytyi roolinsa mukaan, eikä ylitsepääsemättömiin esteisiin törmätty.

Avainsanat tekoäly, reitinhaku, Unity, NavMesh

Sivut 23 sivua.

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Programme in Business Information Technology
HAMK Visamäki

Author Juho Luoto **Year** 2020

Subject Artificial Intelligence in Unity

Supervisors Tommi Lahti

ABSTRACT

The subject of the thesis was to use artificial intelligence with the Unity game engine. The goal was to do research on artificial intelligence and pathfinding algorithms used with it. After the research, the goal was to create a demonstration software in Unity, that uses its NavMesh tools for controlling two unique artificial intelligence characters.

The theoretical background consists of artificial intelligence and its uses, some of the most common pathfinding algorithms and where they are used. Finally, about the Unity-engine itself.

The thesis process revealed that getting two characters to move using the NavMesh features of Unity is simple and straightforward. It became also clear that using something else than NavMesh is not very smart, as it is much more complicated and does not have the prebuilt components ready for use in the engine.

The initial goals were reached, and the demonstration software is working as intended. Both artificial intelligence characters are behaving according to their roles and no major obstacles were faced during the thesis.

Keywords artificial intelligence, pathfinding, Unity, NavMesh

Pages 23 pages.

SISÄLLYS

1 JOHDANTO

Tekoäly on tärkeä osa videopelejä ja sitä käytetäänkin tavalla tai toisella suurimmassa osassa suositummista peleistä. Tekoälynä pidetään tietokoneohjelmaa, joka osaa tehdä älykkäänä pidettyjä valintoja. (Heikkinen, 2017) Tekoäly saattaa esimerkiksi olla pelikentällä liikkuva NPC-hahmo (*Non-Player-Character*), joka tarkoittaa pelihahmoa, jota pelaaja ei itse ohjaa. Kyseiset NPC-hahmot ovatkin usein vihollisia, joita vastaan pelaaja taistelee. Tämä ei kuitenkaan sääntö jokaisessa pelissä, vaan tekoälyn ohjaamat hahmot voivat olla myös olla pelaajan liittolaisia, jotka esimerkiksi seuraavat pelaajaa ja auttavat tätä vihollista vastaan taistelemisessa.

Tekoäly on aina kehittyvä osa videopelejä ja siitä pyritään tekemään jatkuvasti vieläkin älykkäämpi luoden vielä syvempi illuusio, että pelaaja on vuorovaikutuksessa älyllisen hahmon kanssa. Peleissä, joissa pelaaja joutuu suojelemaan NPC-hahmoa, onkin usein ongelmana tekoälyn tekemät huonot valinnat joihin pelaaja itse ei välttämättä pysty vaikuttamaan. Esimerkiksi pelissä *The Last Of Us* tekoälyn ohjaama hahmo saattoi epähuomiossa paljastaa pelaajan sijainnin, kun pelaaja oli yrittämässä hiipiä vihollisten ohi huomaamattomasti.

Opinnäytetyössäni tutustun erilaisiin tekoälyn käyttämiin reitinhakualgoritmeihin. Näitä reitinhakualgoritmejä käyttäen NPC-hahmot pystyvät laskemaan reittejä, joita pitkin ne voivat pelikentällä kulkea. Tutkimuksen jälkeen, käyttäen näitä reitinhakualgoritmeja luon pienen demosovelluksen Unity-pelimoottorissa, jossa kaksi erillistä tekoälyllistä hahmoa pelaavat keskenään piilosta. Toisen tekoälyhahmon tarkoitus on pyrkiä pääsemään mahdollisimman kauaksi toisesta tekoälyhahmosta. Vastaavasti taas toinen tekoäly-hahmoista yrittää pelikenttää tutkien löytää itseltään piiloutuvaa tekoälyhahmoa.

Tutkimuskysymykseni ovat:
Miten Unityn valmiita reitinhakualgoritmejä voidaan hyödyntää?
Millaisia muutoksia reitinhakualgoritmeihin demosovellukseni vaatii?
Millä tavalla erilaiset reitinhakualgoritmit eroavat toisistaan demosovelluksessa?

2 TEKOÄLY

Tekoäly on hyvin laaja käsite, joka jatkaa kehittymistään hurjaa vauhtia. Alan johtavien tutkijoiden Nils J. Nilssonin, Stuart Russelin ja Peter Norvigin mukaan ”tekoälyn avulla koneet, laitteet, ohjelmat, järjestelmät ja palvelut voivat toimia tehtävän ja tilanteen mukaisesti järkevällä tavalla.” (Heikkinen, 2017) Tekoälyä onkin määritelty myös tietokoneohjelmana, joka kykenee tekemään ihmisen mukaan älykkääksi kuvailtuja valintoja. Tekoälyllä ei kuitenkaan ole vain yhtä määritelmää, vaan tekoälyksi voidaan luokitella mikä tahansa tilanteeseen adaptoituva autonominen ohjelma. (Elements of AI, n.d.)

Tekoälyjärjestelmät on luotu siten, että ne pystyvät oppimaan ja adaptoitumaan erilaisiin työtehtäviin, joka mahdollistaa niiden hyödyntämisen moniin eri tilanteisiin. (Heikkinen, 2017) Tekoälyssä on kuitenkin huomioitava, että yhteen tehtävään luotu tekoäly ei tule pystymään toimimaan tehtävänsä ulkopuolella.

2.1 Tekoälyn käyttökohteet

Robotteja alettiin kehittää teollisuuteen 1960-luvulla ja niitä onkin ollut jo pitkään melkein jokaisen tehtaan palkkalistoilla. Robotit ovat tarkkoja ja väsymättömiä ja pystyvät tekemään samaa tehtävää pitkiä aikoja vain pienien huoltokatkojen voimittamana.

Näitä robotteja kuitenkin jatkokehitetään edelleen ja robotiikkaan yhdistetään tekoälyä päivä päivältä enemmän. Jo nykypäivänä tekoäly pystyy ajamaan autoa turvallisemmin kuin ihminen, koska sen tarkkaavaisuus ei häiriinny. Tekoälyä käytetään aktiivisesti myös esimerkiksi röntgenkuvien tulkinnassa, kasvojen tunnistuksessa ja puheen muuttamisessa tekstiksi. Oxfordin tutkijat Frey & Osborne ja MIT:n Brynjolfsson arvioivatkin, että ”20-50 % työtehtävistä korvautuu tekoälyn, automaation ja robotiikan avulla 10-15 vuoden kuluessa.” (Heikkinen, 2017)

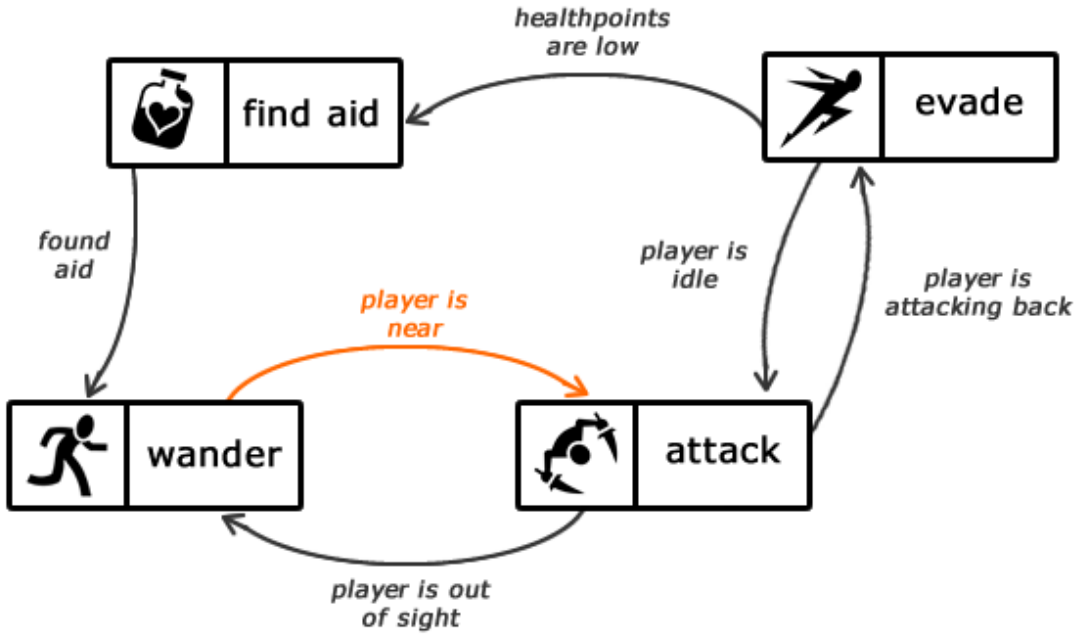
2.2 Tekoäly videopeleissä

Tekoälyä on käytetty videopeleissä aina jo 1950-luvulta lähtien, ensimmäiset versiot tekoälystä olivat kuitenkin yksinkertaisia. Tästä huolimatta kyseiset tekoälyt kykenivät silti voittamaan ihmisen vastapelaajana. (Nield, 2019)

Vuonna 1989 luotu ohjelma Chinook pystyi päihittämään maailman mestarin Tammipelissä. Kyseiselle tekoälylle tallennettiin maailman mestareiden tekemiä aloitussiirtoja, ja niitä käytettiin myöhemmin itse tekoälyllä. Tekoäly käytti siirroissaan ”Deep-Search”-algoritmiä, joka pystyi arvioimaan kaikki mahdolliset lopputulokset tietyn vuoromäärän päähän. Tätä algoritmiä pystyttiin viemään pidemmälle teknologian kehittymisen mukana, kun tietokoneisiin saatiin lisää prosessointi tehoa. Lopulta vuonna 2007, kyseisen ohjelman kehittäjä Jonathan Schaeffer onnistui tiiminsä kanssa niin sanotusti ”ratkaisemaan Tammen”, eli tekoäly pystyi jokaisessa tilanteessa joko voittamaan tai päätymään tasapeliin minkä tahansa pelaajan siirron kanssa. (Sreedhar, 2007)

Toinen tunnettu tekoäly ”Deep Blue” oli IBM:n kehittämä shakkitietokone, joka oli ensimmäinen tietokone, joka voitti silloin hallitsevan Shakin maailman mestarin Garri Kasparovin 10. helmikuuta 1996 virallisessa ottelupelissä.

Vaikka tekoäly videopeleissä on todella laaja käsite, sen ei aina tarvitse tarkoittaa tekoälyä, joka oppisi pelaajan toimista. Monet peleistä käyttävätkin ”Finite State Machine”-algoritmiä, jossa pelinkehittäjät luovat listan kaikista mahdollisista tapahtumista joihin tekoäly saattaa joutua reagoimaan. Näille tapahtumille tämän jälkeen ohjelmoidaan tietynlaiset toiminnot, joiden mukaisesti tekoäly toimii. (kuva 1)



Kuva 1. Finite State Machine-malli videopelistä. (Fernando Bevilacqua, 2013)

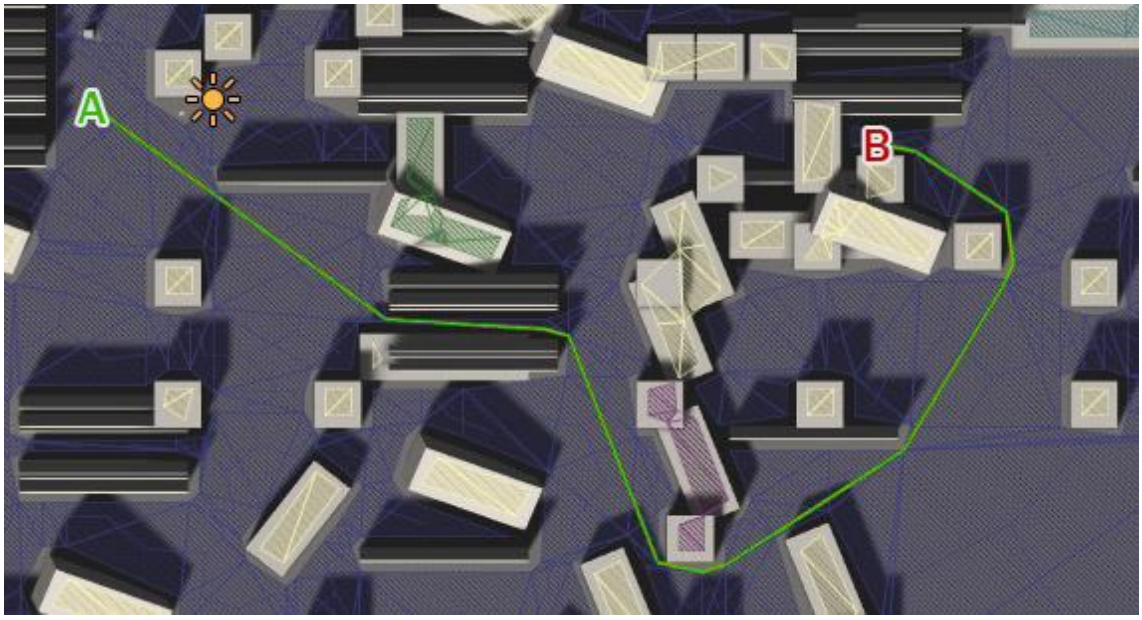
”Finite State Machine”-algoritmi ei kuitenkaan ole optimaalinen esimerkiksi strategiapeleissä, sillä pelaajan olisi helppoa tulkita tekoälyn tekevän samat siirrot joka kerta tiettyjen tapahtumien kohdalla ja oppia pelaamaan juuri tätä pelityyliä vastaan, tehden pelistä nopeasti tylsän ja helpon, rikkoen samalla sen todentuntuisuutta.

Tekoälyn tarkoitus videopeleissä ei kuitenkaan ole luoda myöskään voittamatonta vastusta, vaan luoda peliympäristöstä mielenkiintoisempi ja antaa pelaajalle suurin piirtein oman tasoisensa vastustaja, jota vastaan pelaamalla pelaaja pystyy kehittymään ja saavuttamaan onnistumisen tunteita. On tärkeää huomioida myös, että jos pelissä oleva tekoäly on liian helppo päihittää, saattaa pelaajan mielenkiinto loppua paljon aiemmin. Myös liian vaikea vaikeustaso saattaa ajaa pelaajan pois pelin parista.

Luonnollisesti pelaajien taitotasoilla tulee aina olemaan eroja, ja tämän takia onkin hyvä idea sisällyttää peleihin monia eri vaikeusasteita, joiden avulla pelaaja pystyy määrittelemän itselleen mieluisan vaikeusasteen.

2.3 **Reitinhakualgoritmit videopeleissä**

Reitinhakualgoritmien yleisin käyttötarkoitus on löytää lyhin mahdollinen reitti kahden pisteen välillä tietokoneohjelmassa. (kuva 2) Vaikka reitinhakualgoritmeja käytetäänkin useimmiten videopeleissä ohjaamassa tekoälyllisiä NPC-hahmoja, käytetään niitä myös GPS-laitteissa, robotiikassa ja esimerkiksi liikennesimulaatioissa. (Algfoor, 2015)



Kuva 2. Esimerkki A*-algoritmistä käytössä NavMesh ympäristössä. (A* Pathfinding Project)

Vaikka reitinhakualgoritmeja on kehitetty jo vuosikymmeniä ja niiden tehokuutta on parannettu huomattavasti, riittää niissä edelleen ratkottavaa ja tutkittavaa. Nykypäivänä tärkein osa reitinhaun kehittämisestä liittyy algoritmien tehokkuuden jatkuvaan parantamiseen ja realistisempien reittien luomiseen.

Videopeleissä reitinhaku on hyvinkin keskeinen käsite, ja sitä käytetäänkin lähes jokaisella tekoälyllisellä pelikentällä liikkuvalla hahmolla. Reitinhaku on tärkeä ominaisuus, kun yritetään luoda pelissä liikkuvasta tekoälystä uskottava. Lopullinen tekoälyn käyttämä reitti ei välttämättä ole kuitenkaan lyhin mahdollinen, sillä lisää realistisuutta saadaan, kun käytetyllä reitillä liikkuminen tehdään realistisen näköiseksi, välttäen esimerkiksi liian jyrkkiä tai yhtäkkisiä käännöksiä. Reitinhakualgoritmia pitää myös päivittää nopeaan tahtiin, jos halutaan reagoida esimerkiksi toisiin pelikentällä liikkuviin NPC-hahmoihin ja väistää niitä.

Reitinhaun tärkeys tulee helposti esille, kun se ei toimi kunnolla. Esimerkkejä huonosti toteutetuista reitinhakualgoritmeista on varmasti hyvin monella videopelejä pelanneella. Yleisempänä ongelmana on kuitenkin NPC-hahmon jumiin jääminen, jossa reitinhakualgoritmi jostain syystä jää jumiin eikä löydä reittiä pois esimerkiksi pöydän päältä, johon sen ei ollut alun perin tarkoitus päästä. Tämä mahdollistaa pelaajalle äärimmäisen helpon vastustajan ja tapahtuessaan tarpeeksi usein voi viedä myös mielenkiinnon pelistä.

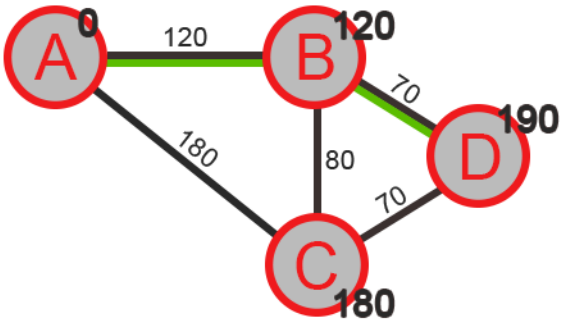
Reitinhakua saatetaan käyttää myös pelaajan itse ohjaamilla hahmoilla. Esimerkkinä RTS-peleissä (Real-Time-Strategy), ohjaa pelaaja usein monia kymmeniä hahmoja taistelukentällä. Tämä on usein toteutettu menetelmällä, jossa pelaaja valitsee haluamansa yksiköt ja klikkaa pelikartalla mihin haluaa ne siirrettävän. Tämän jälkeen reitinhakualgoritmi ottaa ohjat käsiin ja pyrkii lyhintä reittiä pitkin liikuttamaan yksiköt pelikartalla pelaajan haluamaan sijaintiin. Mikäli tässä vaiheessa reitinhaku- algoritmi ei osaisi kiertää matkalla olevia esteitä vaan törmäisi niihin, olisi pelikokemus huomattavasti huonompi.

2.3.1 Dijkstra

Dijkstran algoritmi on Edsger Dijkstran vuonna 1959 kehittämä reitinhakualgoritmi, joka on edelleen yksi kuuluisimmista lyhyimmän reitin löytämiseen käytettävistä algoritmeista. Algoritmia käytetään esimerkiksi navigointilaitteissa, tietoliikenneverkkojen reitityksessä, sekä joissakin videopeleissä.

Algoritmin toiminta alkaa tutkimalla etäisyyttä graafilla olevasta yhdestä pisteestä sen ympärillä oleviin pisteisiin, kun kaikki etäisyydet on mitattu, siirtyy algoritmi seuraavaan pisteeseen.

Tämän jälkeen algoritmi mittaa uudet etäisyydet seuraaviin pisteisiin lisäten uusiin etäisyyksiin myös edelliseltä pisteeltä siirrytyn etäisyyden. Algoritmi jatkaa samaan kaavaan, kunnes kaikki graafilla olevat pisteet on käyty läpi. Mikäli kesken prosessin algoritmi huomaa, että johonkin pisteelle tullut etäisyys on lyhempi, kuin sille aiemmin laskettu etäisyys, päivittää se etäisyyden uuden reitin mukaiseksi. Tämä mahdollistaa lopullisen lyhimmän reitin löytämisen. (kuva 3)



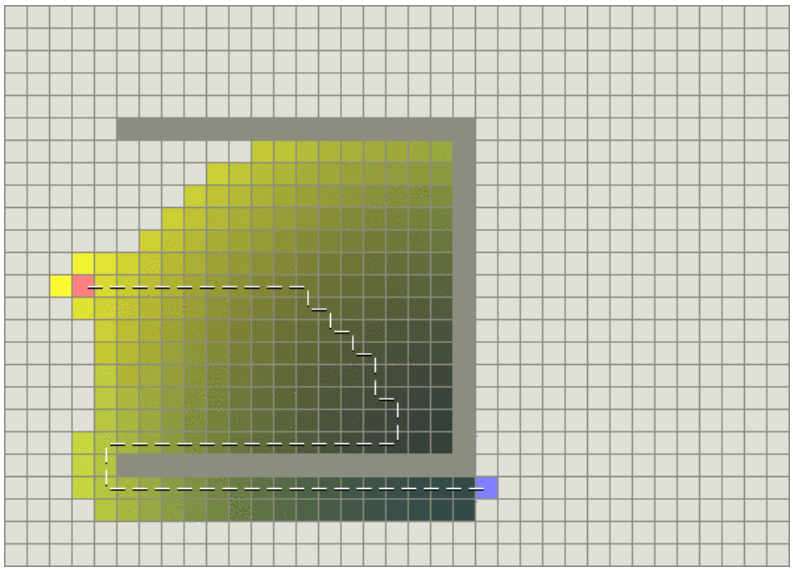
Kuva 1. Esimerkki graafi pisteiden A ja D välillä. Lyhimmäksi reitiksi osoittautuu siis A>B>D etäisyydellä 190.

Algoritmin alkuperäinen muoto vaati kaksi erillistä pistettä, joiden välinen etäisyys tämän jälkeen mitattaisiin. Nykyään kuitenkin useammin käytetty algoritmi tarvitsee vain yhden pisteen graafilla, joka määritetään lähteeksi. Tämän jälkeen algoritmi mittaa lyhimmän etäisyyden lähteeseen jokaiselta muulta pisteeltä.

2.3.2 Greedy best-first-search

”Greedy best-first-search” eli ”ahne paras-ensin”-reitinhaku on hyvin samankaltainen, kuin Dijkstran algoritmi, mutta se käyttää myös heuristiikkaa siitä, kuinka kaukana maalista jokainen piste graafilla on. Tämä tekee algoritmista myös huomattavasti nopeamman kuin Dijkstran-algoritmi, koska ”ahne paras-ensin”-algoritmi käyttää koko ajan tietoa maalin etäisyydestä, joka ohjaa sen valitsemaa polkua koko ajan kohti maalia.

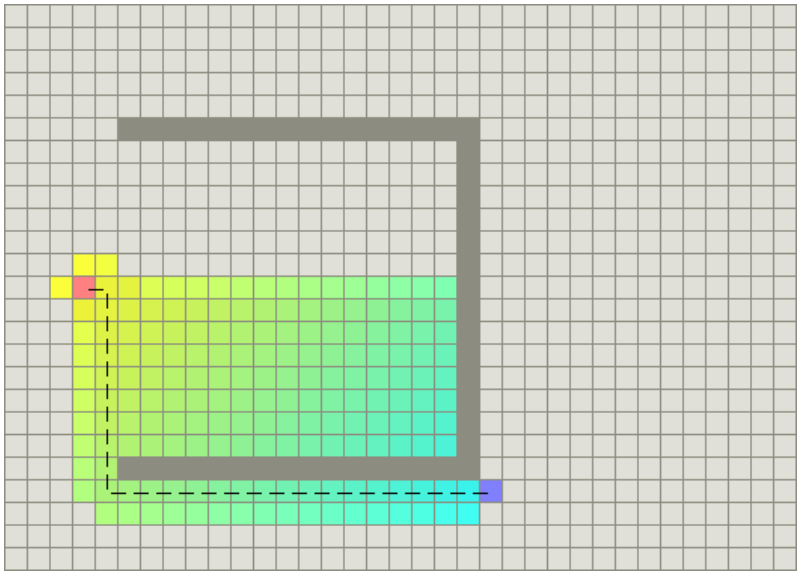
”Ahne paras-ensin”-algoritmi ei kuitenkaan luo varmuutta löytää lyhintä reittiä, koska se yrittää koko ajan liikkua maalia kohti välittämättä siitä kasvaako reitin pituus prosessin mukana. Tämä saattaakin johtaa algoritmin ansaan, jossa se huomaa olevansa umpikujassa, ja joutuukin palaamaan taaksepäin päästäkseen jälleen lähemmäksi maalia. Havainnollistus tästä kuvassa 4.



Kuva 4. Esimerkki ahne paras-ensin-algoritmin ansasta, joka pidentää huomattavasti reitin pituutta. (Amit)

2.3.3 A* (A-star)

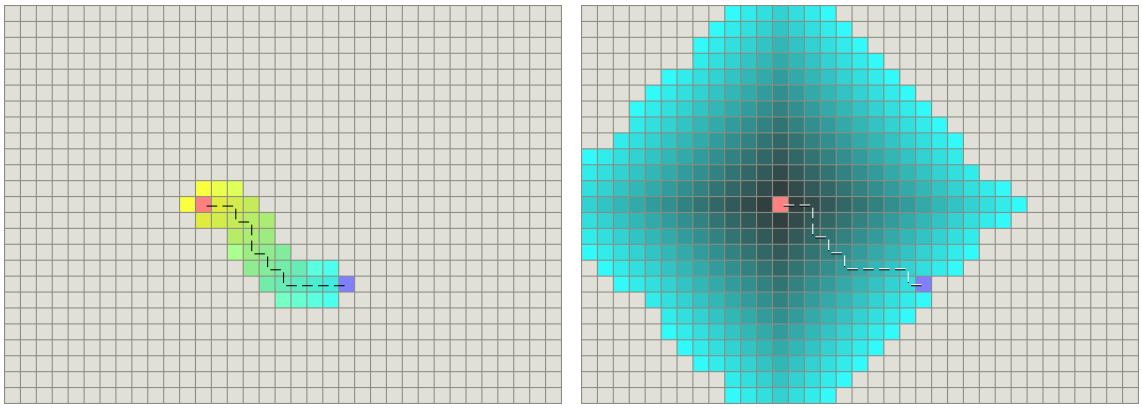
A*-algoritmi on *paras-ensin* -haun tunnetuin algoritmi, joka kehitettiin vuonna 1968 yhdistämään Dijkstran-algoritmin lyhimpään reittiin pyrkiminen *ahne paras-ensin*-haun tapaan käyttää tietoa maalin etäisyydestä. Vaikka pelkällä tiedolla maalin etäisyydestä ei pysty kertomaan varmaa lyhintä reittiä, on A*-algoritmi rakennettu sen päälle siten, että se pystyy takaamaan lyhimmän reitin joka kerta. (kuva 5)



Kuva 5. Esimerkki A* -algoritmin tavasta selviytyä aiemmin käytetystä ”algoritmi-ansasta”. (Amit)

A*-algoritmi käyttää Dijkstran-algoritmin tapaa tutkia lähellä lähdettä olevia pisteitä ja *ahne paras-ensin* -algoritmin tapaa tutkia maalin lähistöllä olevia pisteitä. Algoritmi laskee jokaisen pisteen *F*-costin, joka muodostuu *G*-costista, sekä *H*-costista.(F = G + H) Nämä arvot muodostuvat pisteiden etäisyyksistä toisistaan graafilla. *G*-cost tarkoittaa etäisyyttä annetulta pisteeltä aloituspisteelle, kun taas *H*-cost tarkoittaa arvioitua etäisyyttä annetulta pisteeltä maaliin. Tätä tietoa käyttäen pystyy algoritmi vertailemaan jokaisen pisteen etäisyyttä alkuun, sekä maaliin, löytäen lyhimmän reitin joka kerta. (Lester, 2005)

A* -algoritmi on huomattavasti nopeampi kuin esimerkiksi Dijkstran -algoritmi (kuva 6), koska se pystyy rajaamaan pois graafilla olevat turhat pisteet, joita ei tarvita kyseisellä reitillä. (kuva 6) Tämä mahdollistaa algoritmille huomattavasti pienemmän määrän pisteitä, joita algoritmin pitää tutkia, joka taas nopeuttaa algoritmin suoritusnopeutta.



Kuva 6. Esimerkki A*-algoritmin tutkimista pisteistä verrattuna Dijkstran-algoritmin tutkimiin pisteisiin.

3 UNITY

Unity on monelle eri alustalle luotu pelimoottori, joka julkaistiin Applen vuosittain pidettävässä Worldwide Developers Conferencessa vuonna 2005. Aluksi Unity -pelimoottori tuki vain Applen Mac-alustaa, joka ei itsessään ollut kovin merkittävä pelialusta. Pian tämän jälkeen pelimoottoriin kuitenkin lisättiin tuki Windowssille, sekä selaimille. Ensimmäiset mahdollisuudet mobiilipelien kehitykseen Unity-pelimoottorilla syntyivät vuonna 2008, kun pelimoottoriin lisättiin tuki Applen Iphone-puhelimille. (Junttila, 2015)

Unityn kehittänyt peliyhtiö Unity Technologies perustettiin vuonna 2004. Yhtiön perustajiin kuuluivat David Helgason, Nicholas Francis ja Joachim Ante. Yhtiö tunnettiin aluksi nimellä ”Over The Edge Entertainment”, joka vaihdettiin vuonna 2007 nimeen Unity Technologies. Idea Unityn kehittämiseen lähti liikkeelle, kun yhtiön ensimmäinen peli epäonnistui taloudellisesti. Peliä kehittäessään he huomasivat pelimoottoreiden ja editorien todellisen arvon alalla, ja kuinka vaikeaa pienempien peliyhtiöiden oli päästä käsiksi niihin. Unityn lisätavoitteeksi tulikin luoda edullinen pelimoottori pienemmille peliyhtiöille ja muille peliohjelmointi harrastelijoille. Rahoitusta Unityn kehittämiseen yhtiö sai isommilta yhteistyökumppaneilta. Projekti sai paljon suosiota tavoitteestaan luoda edullinen pelimoottori pienille peliyhtiöille. (Krupp, 2014)

Unityn suosio kasvoi valtavasti tuettujen laitteiden määrän kasvaessa. Nykypäivänä tuettuja alustoja on valtavasti. Tuettuihin alustoihin kuuluvat tätä kirjoittaessa esim. seuraavat: iOS, Android, Windows, Universal Windows Platform, Mac, Linux/Steam OS, WebGL, PlayStation 4, Xbox One, Nintendo 3DS, Oculus Rift, Google Cardboard Android & iOs, Steam VR PC & Mac, Playstation VR, Gear VR, Windows Mixed Reality, Daydream, Android TV, tvOS, Nintendo Switch, Apple ARKit, Google ARCore ja Vuforia. (Unity3D, 2019)

Unity on tukenut pelien scriptien luomiseen *Boo*-ohjelmointikieltä, joka kuitenkin poistettiin alustalta version 5.0-tullessa vuonna 2015. Tämän lähtiessä Unityn tärkeimmäksi ohjelmointikieleksi jäi C# (C sharp), jonka lisäksi vaihtoehtoisena ohjelmointikielenä oli myös JavaScript. JavaScriptin tukemisen lakkauttaminen Unitystä ilmoitettiin vuonna 2017, jättäen jäljelle enää vain C#-ohjelmointikielen. Tätä perusteltiin JavaScriptin pienien käyttäjämäärien takia. Paljon JavaScriptiä käyttäviä projekteja oli Unityssä vain 3,6%. Tämän lisäksi todettiin, ettei JavaScriptillä pystytty tekemään mitään, mitä ei pystyisi ohjelmoimaan myös C# -ohjelmointikielellä ja JavaScript-tuen jatkuva kehittäminen söi paljon kehitystiimin resursseista. (Fine, 2017)

Unity-pelimoottori sisältää editorin kaksi- ja kolmiulotteisiin peleihin. Editori kaksiulotteisille peleille lisättiin kuitenkin vasta vuonna 2013. Se sisältää myös oman fysiikkamoottorinsa. Pelimoottorin tärkeimpiä ominaisuuksia ovat sen sisältämä ”Project Browser”, jonka avulla käyttäjän on helppo nähdä kaikki tällä hetkellä käytössä olevat objektit. Toisena tärkeänä pidettävänä ominaisuutena on ”Inspector”, jonka avulla pystyy helposti muokkaamaan jokaisen objektin ominaisuuksia. Unity tarjoaa myös ”Game View”-ikkunan, jonka avulla on helppo testata miltä peli tulee näyttämään lopullisella laitteella ja kuinka se toimii.

Unityä pystyy käyttämään täysin ilmaiseksi, mikäli vuoden kokonaistulot omista projekteista pysyvät alle 100 000 dollarissa. Tämän rajan rikkoutuessa Unity tarjoaa useita erilaisia maksullisia paketteja. Unityn tarjoamat maksulliset palvelut suuremmille kehittäjille kuuluvat kahteen eri pakettiin. *Plus*-pakettia tarjotaan 40 dollarin kuukausihinnalla, ja sitä voidaan käyttää, mikäli vuoden kokonaistulot projekteista pysyvät alle 200 000 dollarissa. Arvokkaampana pakettina tulee ”Pro”-paketti, jota vaaditaan, mikäli vuoden kokonaistulot ovat yli 200 000$. Unity tarjoaa myös ”Learn Premium”-pakettia, joka tarjoaa käyttäjälle live-aikaisia opetustunteja, sekä ilmaista versiota enemmän valmiita resursseja. (Unity3D, 2020)

3.1 **Reitinhaku Unityssä**

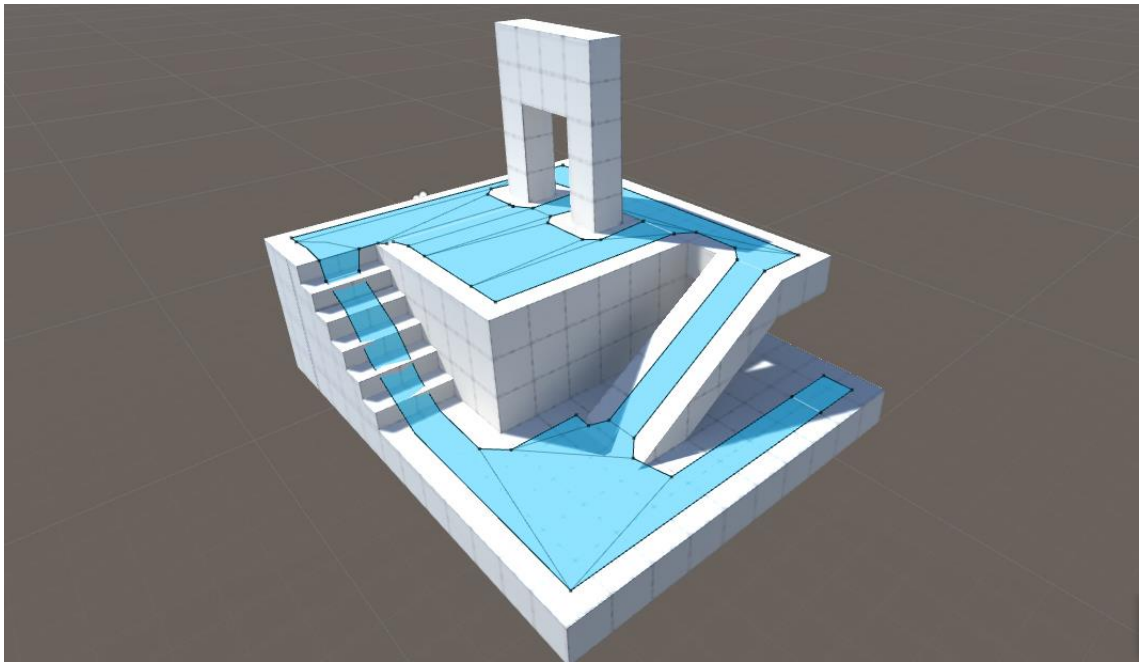
Reitinhaun Unityssä voisi toteuttaa niin sanotusti huijatulla tekoälyllä, joka toteutettaisiin luomalla pelikentälle useita sijaintipisteitä, ja luomalla funktio, joka laskisi aina lyhimmän mahdollisen reitin kulkemalla näiden sijaintien välillä. Toinen vaihtoehto olisi käyttää raycasteja ympäristön tutkimiseen ja antaa sitä kautta tekoälylle ohjeita. Helpoin vaihtoehdoista on kuitenkin käyttää Unityyn sisältyvää NavMesh-ominaisuutta.

Unityssä on sisäänrakennettu reitinhakualgoritmi, joka hyödyntää suoraan sitä muuttamatta Unityn NavMesh-toimintoa, jossa valitut objektit kartoitetaan tekoälyä varten käytettäväksi. Tämä tekee pelialueen kartoittamisesta todella nopeaa ja helppoa. Alueen kartoittamisen jälkeen reitinhakualgoritmi toimii käyttämällä valmiiksi asennettua A*-algoritmia.

Projektiin tarvitsee kuitenkin ladata Unityn omilta sivuilta paketti, joka sisältää joitakin tarkempia NavMesh-ominaisuuksia, joita käyttämällä saadaan käytettyä Unityn reitinhakualgoritmiä. Tässä paketissa on mukana esimerkiksi NavMeshSurface-komponentti, jota käytetään itse NavMeshin luomisessa. (Unity, 2020)

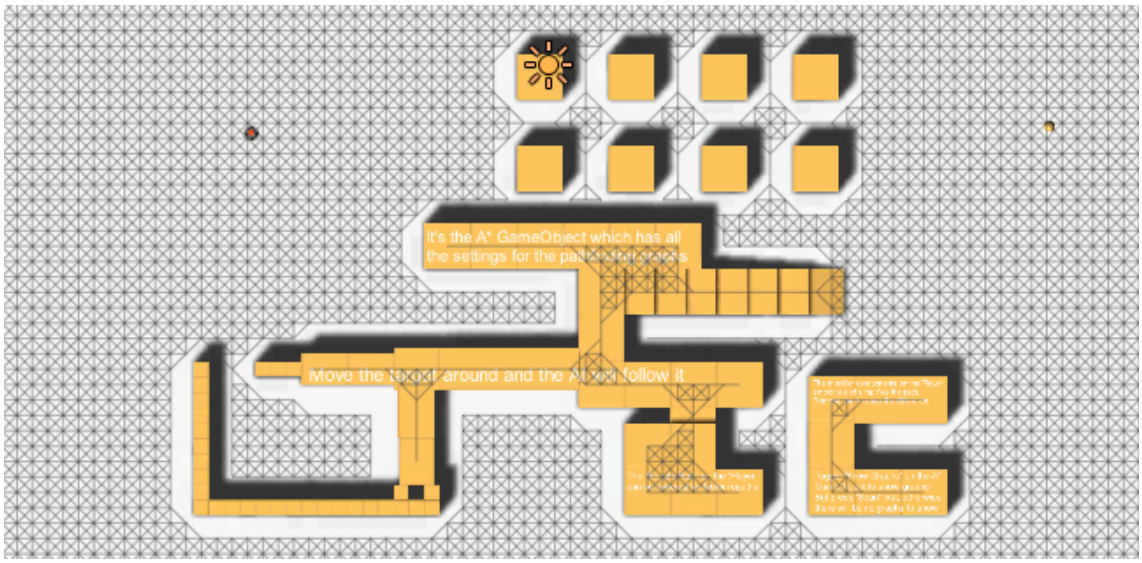
3.2 **NavMesh**

NavMesh eli Navigation Mesh on tekoälyllä käytetty tietorakenne, jolla autetaan tekoälyllisiä objekteja liikkumaan monimutkaisessa ympäristössä. NavMesh muodostuu eri kokoisista ja -mallisista polygoneista mukautuen ympäristön objekteihin. Näillä polygoneilla määritetään tekoälylle alue, jolla se pystyy vapaasti liikkumaan törmäämättä kulkua estäviin esteisiin. (kuva 7) Navigation Meshillä reitinhaku pystytään toteuttamaan esimerkiksi A* -algoritmilla. Unityssä on sisäänrakennettu ominaisuus, jolla pystytään luomaan automaattisesti Navigation Mesh valitulle alueelle, joka käyttää valmiiksi A*-reitinhaku algoritmia.



Kuva 7. NavMesh-näkymä Unity editorissa. (Unity Learn)

Navigation Mesh:iä käytetään yleisesti sen joustavuudesta kartoittaa isojakin alueita käyttämättä ylimääräisiä polygoneja, vaan pystyy niiden koko skaalautumaan ympäristön objektien mukaan. Vastaavia tietorakenteita on monia erilaisia, esimerkkinä Kuva 8:ssä näkyvä Grid Graph, joka soveltuu ruudukkopohjaisiin peleihin paremmin kuin NavMesh. Tässä opinnäytetyössä keskitymme kuitenkin Navigation Mesh:iin, joka on helppo luoda Unityn omilla työkaluilla. NavMesh on ollut videopeleissä suuressa suosiossa sen mahdollisuudesta kartoittaa suuriakin alueita käyttämällä paljon pienempää määrää muistia verrattuna esimerkiksi Grid Graph:iin, jonka täytyy luoda huomattavasti enemmän pisteitä isoille avonaisille alueille. (kuva 8)



Kuva 8. Grid Graph esimerkki Unity editorissa. (A* Pathfinding Project)

4 REITINHAUN KÄYTTÄMINEN UNITYSSÄ

4.1 Demosovellus ja sen tavoitteet

Opinnäytetyön tavoitteena on luoda Unity-pelimoottorissa pieni testiympäristö, johon sijoitetaan kaksi tekoälyllistä hahmoa pelaamaan ”hippaa” keskenään. Piiloutuvan tekoälyn tarkoitus on pyrkiä jatkuvasti kauemmaksi etsivästä tekoälystä. Ympäristön kartoittaminen tekoälylle aiotaan toteuttaa käyttäen Unityn NavMesh-ominaisuutta.

Tekoälyhahmoille on tarkoituksena luoda jonkin näköinen näkökenttä, jotta piiloutuva tekoälyhahmo pystyisi helpommin välttelemään etsivää tekoälyhahmoa. Lisäksi demoon olisi kätevää saada jonkin näköinen ajastin, joka mittaisi kuinka kauan etsivällä hahmolla kestää löytää karkaava hahmo ja saada se kiinni. Tämän avulla pystyisi helposti arvostelemaan tekoälyn nopeutta löytää karkuriagentti ja saada tämä kiinni.

Sovellukseen voisi lisätä myös useita karkuun juoksevia agentteja, jolloin etsivän agentin pitäisi saada niistä jokainen kiinni ennen kuin demo pysäytettäisiin. Toinen lisäominaisuus voisi olla agenttien sijainnin päättäminen demon suorittajalle.

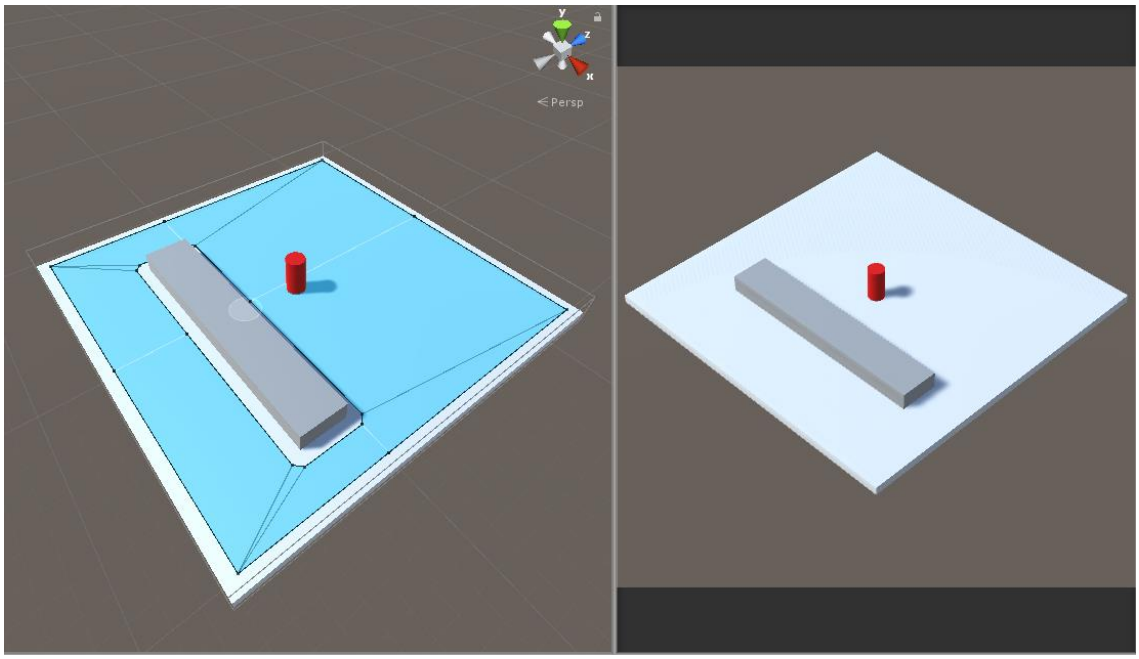
4.2 NavMeshin luominen

Projekti aloitettiin luomalla pieni demoympäristö, jossa pystyttäisiin helposti testaamaan, mikäli reitinhaku toimii odotettavalla tavalla käyttämättä siihen liikaa aikaa vielä tässä vaiheessa. Ympäristö koostui yhdestä lattiaobjektista, yhdestä seinäobjektista, sekä yhdestä hahmo-objektista.

Aloittaessa työstämään reitinhakua Unityssä, tajuttiin hyvin pian, ettei suurta osaa NavMesh ominaisuuksista ole sisälletty Unityn normaaliin asennukseen, vaan piti ne ladata erikseen scripti- ja editoritiedostoina ja liittää olemassa olevaan projektiin. (Unity, 2020)

Näillä tiedostoilla saatiin esimerkiksi komponentti ”NavMeshSurface”, jota käyttämällä pystyttiin automaattisesti luomaan NavMesh valituille objekteille. NavMeshSurfacea luodessa pystyy myös poislukemaan tiettyjä objekteja asettamalla niille komponentin ”NavMeshModifier”. Tätä komponenttia käyttämällä pystyy esimerkiksi asettamaan, että objekti jätettäisiin kokonaan huomiotta, jolloin NavMesh muodostuu vain sen läpi. Tämän kaltaista voisi käyttää esimerkiksi vesiobjektilla, jonka läpi haluat agenttien edelleen liikkuvan.

Projektissa käytettiin kuitenkin ”override” toimintoa, jolla pystyttiin asettamaan seinä-objekti muotoon ”not walkable”, joka estää NavMeshiä luomasta käveltävää pintaa seinien päälle. (kuva 9)

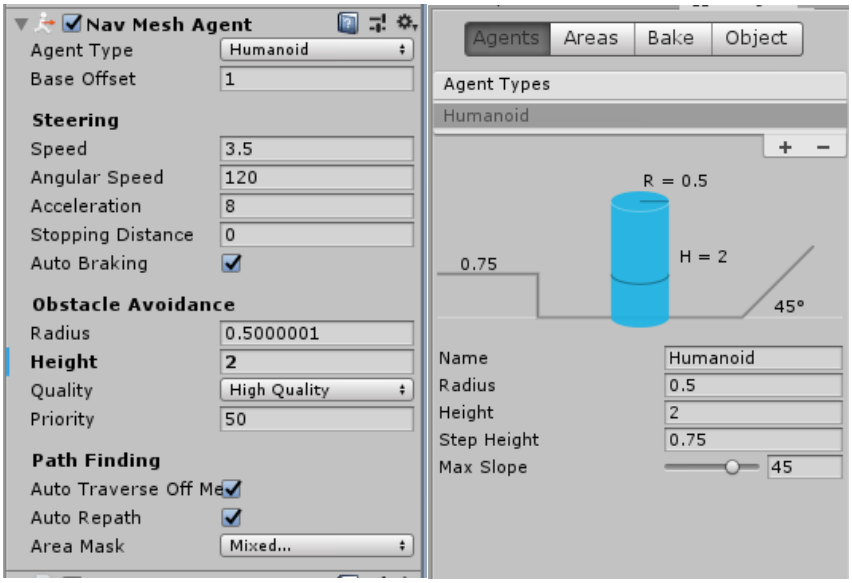


Kuva 9. Esimerkki ensimmäisestä NavMesh:istä.

4.3 **Reitinhaun käyttöönotto NavMesh-agentille**

Kun NavMesh saatiin kuntoon, päästiin käyttöönottamaan itse reitinhakua. Tämä tarkoitti *NavMeshAgent*-komponentin(kuva 10) lisäämistä hahmo-objektille, jossa määritetään agentin reitinhaku-ominaisuudet. Näitä ovat esimerkiksi hahmon liikkumisnopeus, kääntymisnopeus, leveys, korkeus ja kuinka jyrkkää mäkeä ne voivat kiivetä.

Agenteille voi luoda myös useita erilaisia pohjia, joille tallennetaan omat käyttötarkoitusta mukailevat ominaisuutensa. Esimerkiksi peliin voisi kehittää agentin jättiläiselle, joka olisi hitaampi, ja huomattavasti isompi, eikä mahtuisi yhtä pienistä väleistä NavMeshillä.



Kuva 10. Humanoid-agentin ominaisuus-näkymät.

Tämän lisäksi hahmo-objektille piti lisätä myös scripti-tiedosto, jolla hahmoa pystyttäisiin ohjaamaan. Yksinkertainen tapa toteuttaa tämä oli kirjoittaa scriptiin ”Raycast”-toiminto, joka lähettäisi säteen kamerasta hiiren klikkaamaan sijaintiin, ja testaisi mikäli tämä säde osuu johonkin objektiin. Mikäli osuma objektiin todettaisiin todeksi, annettaisiin agentille tämän osuman koordinaatit tavoitesijainniksi. Tämän jälkeen Unityn oma reitinhakualgoritmi A*-algoritmiä käyttäen ohjaa hahmon niin lähelle tavoitesijaintia, kuin sille annettu NavMeshAgent pystyy siirtymään. (kuva 11)

```
if (Input.GetMouseButton(0))
{
    Ray ray = cam.ScreenPointToRay(Input.mousePosition);
    RaycastHit hit;

    if (Physics.Raycast(ray, out hit))
    {
        agent.SetDestination(hit.point);
    }
```

Kuva 11. Raycastin, eli säteen lähettäminen hiiren klikkaamaan sijaintiin.

4.4 NavMesh-agenttien satunnainen vaeltelu

Kun agentti oli saatu onnistuneesti liikkumaan käyttäjän haluamiin sijainteihin, haluttiin projektiin ruveta työstämään satunnaista reitinvalinta toimintoa. Lähdin lähestymään tätä kahdella eri tavalla. Toisessa tavassa määritettiin ympäristöön muutama eri tavoitesijainti ja kirjoitettiin scripti, jossa agentille satunnaisesti syötetään yksi näistä tavoitesijainneista. Aina, kun agentti saavuttaa tavoitesijaintinsa, määrätään sille jälleen uusi satunnainen tavoitesijainti. Näitä pisteitä satunnaisessa järjestyksessä kulkeva agentti liikkui satunnaisesti, mutta siinä oli edelleen havaittavissa pientä järjestelmällisyyttä ennalta määrättyjen pisteiden pohjalta. (kuva 12)

```
void Update()
{
    speed = agent.velocity.magnitude;


    waypoints = GameObject.FindGameObjectsWithTag("waypoint");
    waypointInd = Random.Range(0, waypoints.Length);

    if (speed < 0.1)
    {
        agent.SetDestination(waypoints[waypointInd].transform.position);
    }
```

Kuva 12. Scriptin kohta tavoitesijainti vaeltelusta.

Toisenlaista vaeltelua varten kopioitiin hahmo-objekti, ja tehtiin siitä samalla myöhemmin käytettävä niin sanottu karkuri-hahmo, jonka tarkoituksena lopullisessa projektissa olisi paeta toisen hahmo-objektin luota.

Tälle hahmolle annettiin täysin satunnainen vaelteluscripti, jonka toiminta perustuu ympyrään hahmon ympärillä. Tästä ympyrästä saadaan arvottua satunnainen sijainti ”Random.insideUnitSphere”-toiminnolla, joka palauttaa kolmiulotteisen sijainnin ympyrän sisäpuolelta. Kolmiulotteisuutta ei kuitenkaan tarvittaisi, sillä hahmon tulee pysyä koko ajan samalla korkeudella, joten korkeusakseli jätetään satunnaisesta sijainnista kokonaan käyttämättä.

Tämän lisäksi funktioon piti lisätä myös kohta, joka vertaa satunnaista pistettä *NavMeshSurfaceen,* ja etsii siitä lähimmän pisteen NavMeshin sisäpuolelta. Tätä tarvitaan estämään hahmon tavoitesijainnin joutumista pelialueen ulkopuolelle. Toiminnosta huolimatta huomattiin, että joutuessaan kiinni pelialueen laitaan, saattoi objekti jäädä joskus jumiin antaen tavoitesijainniksi ”infinite”-koordinaatteja. Tälle ongelmalle luotiin laastariksi pieni ajastin, joka testaa, mikäli edellisestä tavoitesijainnista on liian kauan aikaa ja määrää karkuri-hahmolle uuden tavoitesijainnin. (kuva 13)

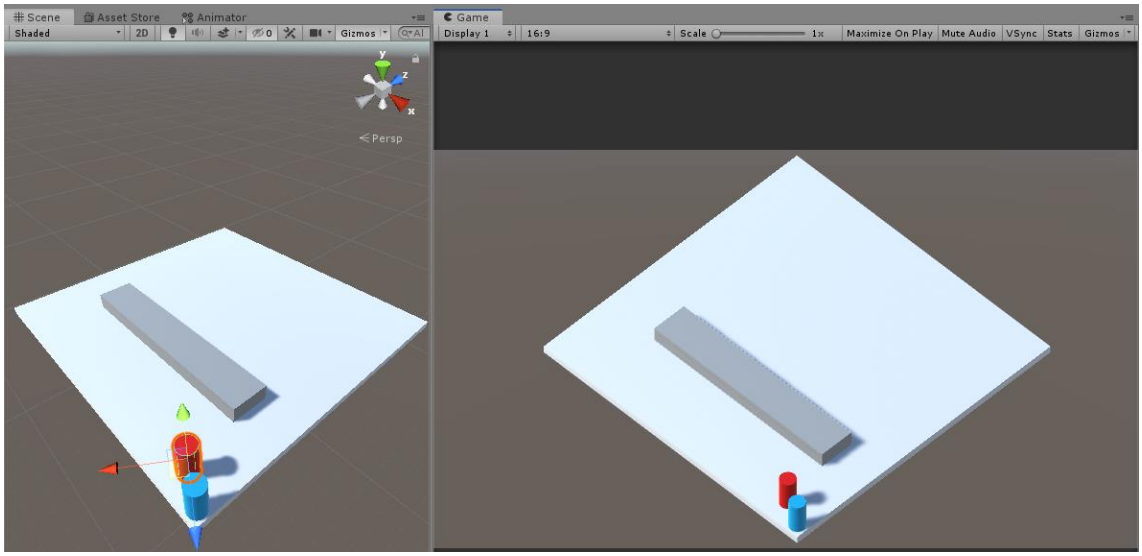
```
public void Wander()
{
    if (Vector3.Distance(transform.position, wanderPoint) < 2f)
    {
        wanderPoint = RandomWanderPoint();
        timeoutTimer = 0;
    }
    else
    {
        if (timeoutTimer < 250)
        {
            evader.SetDestination(wanderPoint);
        }
        else
        {
            wanderPoint = RandomWanderPoint();
            timeoutTimer = 0;
        }

    }
}

3 references
public Vector3 RandomWanderPoint()
{
    Vector3 randomPoint = (Random.insideUnitSphere * wanderRadius) + transform.position;
    NavMeshHit navHit;
    NavMesh.SamplePosition(randomPoint, out navHit, wanderRadius, -1);
    return new Vector3(navHit.position.x, transform.position.y, navHit.position.z);
}
```

Kuva 13. Scriptin kohta täysin satunnaisesta vaeltelusta.

4.5 **NavMesh-agenttien toisiinsa reagoiminen**

Kun ympäristöön oli saatu onnistuneesti kehitettyä kaksi eri periaatteilla vaeltelevaa agenttia, oli aika ruveta työstämään näille agenteille erilaisia tapoja reagoida toisiinsa. Etsivälle agentille tehtiin ensimmäiseksi versioksi hyvin yksinkertainen jahtaamistoiminto, jossa testattiin etsivä- ja karkuriagenttien etäisyyttä toisiinsa. Mikäli etäisyys oli pienempi kuin määritetty jahtaamisetäisyys, annettaisiin etsivälle agentille tavoitesijainniksi karkuriagentin sijainti. Tämä johti versioon, jossa etsiväagentti huomattuaan karkurin seuraisi tätä ympäri pelialuetta ikuisesti. (kuva 14)

Tämän version jälkeen oli aika ruveta työstämään omaa reagointitapaa karkuriagentille. Karkurille haluttiin antaa mahdollisuus karata etsivän jahtaamiselta, mikäli ei törmättäisi esimerkiksi umpikujaan. Tämä toteutettiin antamalla karkurille pieni lisä liikkumisnopeuteen, kun tämä oli tarpeeksi lähellä etsivää agenttia. Karkurin tavoitesijainniksi ei annettu tiettyä sijaintia ympäristössä, vaan verrattiin koko ajan etäisyyttä etsiväagenttiin, ja karkuriagenttia pyrittiin liikuttamaan vain poispäin etsivästä.



Kuva 14. Esimerkki nurkkaan ahdetusta karkuriagentista.

Tämän muutoksen jälkeen käytössä oli versio, jossa kaksi agenttia liikkuivat ympäri peliympäristöä törmäten välillä toistensa lähialueelle, joka johtaisi hetken takaa-ajotilanteeseen. Nämä takaa-ajot ratkesivat yleisesti jonkin ajan kuluttua, kun karkuri loppujen lopuksi pääsisi tarpeeksi kauaksi etsivästä agentista lopettaen samalla etsivän agentin jahtaamistoiminnon.

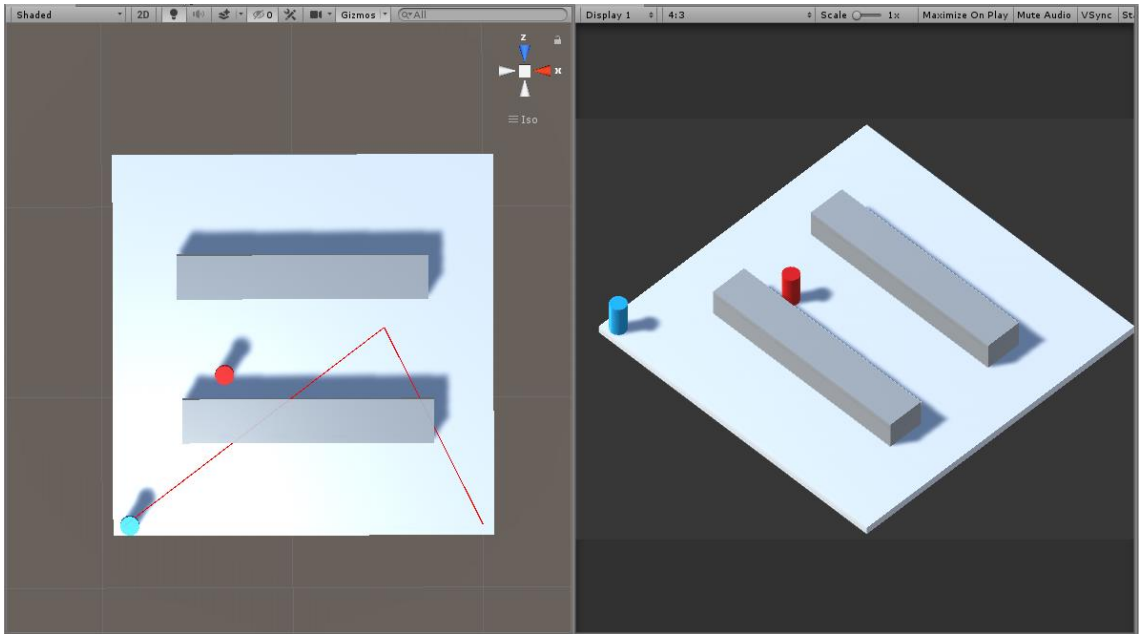
4.6 **Raycasting**

4.6.1 Karkuriagentin piiloutumislogiikka

Seuraavaksi demoon haluttiin järjestelmä, jolla pystyttäisiin testaamaan, mikäli agentit näkevät toisiaan ja ohjelmoimaan niille tämän tiedon mukaisia toimintoja. Ensimmäisenä ideana tuli käyttää näkökentän testaamiseen ”Physics.Raycast”-toimintoa, joka lähettää säteen sille valitusta koordinaatista ennalta määrättyyn suuntaan. Tätä sädettä käyttämällä on mahdollista vastaanottaa paljon informaatiota säteeseen liittyen, jota pystyy käyttämään hyödyksi näkökenttien luomisessa. Säteiden tallettama tieto sisältää esimerkiksi säteen pituuden ja mihin objektiin säde on osunut.

Tavoitteena olisi saada luotua näkökenttä kummallekin agentille, jolloin vältyttäisiin agenttien seinien läpi tunnistamiselta. Tämä helpottaisi huomattavasti piiloutuvan agentin tehtävää, kun sen liikkeitä ei enää havaittaisi seinien toiselta puolelta.

Ensimmäisenä tehtävänä aloitettiin raycastin toteuttaminen karkuriagentin piilopaikan valinta-funktiossa, jotta agentti ei yrittäisi jatkuvasti piiloutua etsivän agentin näkökentälle. Karkuriagentin scriptin piiloutumis-funktioon lisättiin raycast-testi, joka lähetti säteen satunnaisesti sijoitetusta piiloutumissijainnista etsiväagenttiin. Mikäli raycast osui etsivään agenttiin, voitiin päätellä, ettei välissä ollut minkäänlaisia esteitä ja käskettiin scriptin arpoa uusi satunnainen piilopaikka. Näitä piilopaikkoja arvottiin niin kauan, kunnes näkökentän ulkopuolella oleva piilo löytyi. (kuva 15)



Kuva 15. Raycastit punaisilla viivoilla kuvaavat näköyhteyttä vanhaan etsiväagentin sijaintiin. (punainen objekti)

Tässä tehtävässä oli hieman haasteita saada raycast suunnattua oikeaan suuntaan, ja jouduttiin tekemään jonkin verran tutkimustyötä, kunnes löydettiin lyhyt ja ytimekäs vastaus säteen suuntaamisesta. Suunta tulisi määrittää luomalla laskutoimitus, jossa kohdesijainnin sijainti vähennetään säteen lähdesijainnista. (kuva 16)

```
//Määritetään säteen suunta
Vector3 raycastDir = player.transform.position - hidePoint;
//Piirretään editoriin säde punaisella, jotta sitä on helpompi tulkita.
Debug.DrawRay(hidePoint, raycastDir, Color.red, 5f);
```

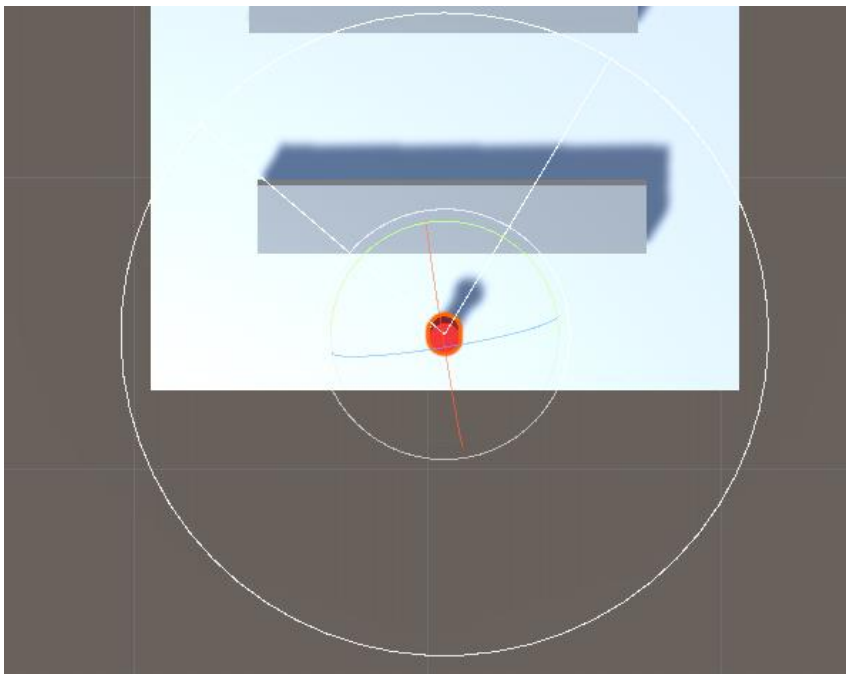
Kuva 16. Määritetään säteen suunta, ja piirretään säde editoriin.

4.6.2 Etsiväagentin näkökentän luominen

Kun karkurin piiloutumislogiikka oli saatu kuntoon, aloitettiin työskentely etsivän agentin näkökentän parissa. Näkökentän on tarkoituksena olla havainnollinen ja helposti nähtävä myös demon suorittajalle. Tämä näkökenttä korvaisi tämänhetkisen jahtaamistoiminnon, joka määrittää jahtaamiseen lähtemisen ainoastaan tutkimalla etäisyyttä etsiväagentista karkuriagenttiin. Vanhan funktion suurin ongelma on etsiväagentin mahdollisuus havaita karkuriagentti myös seinäobjektien läpi, jotka muuten estäisivät näkyvyyden agenttien välillä.

Tehtävää lähdettiin työstämään kahden uuden scriptin kanssa. Ensimmäisenä avattiin näkökenttää käsittelevä scripti ”FieldOfView.cs”. Tähän scriptiin luotiin ”float”-tyyppiset muuttujat näkökentän pituudelle(viewRadius) ja sen leveydelle(viewAngle). Näiden lisäksi luotiin funktio, jonka avulla pystytään kätevästi laskemaan näkökentän laajuuden luoman kulman suunta.

Toinen scripti ”FieldOfViewEditor.cs” luotiin, sillä olisi kätevää pystyä havainnollistamaan kulmaa editorissa. Tämä scripti toimii ”custom editorina”, joka auttaa näkökentän määrittelyjen kanssa. Tätä scriptiä käyttäen pystytään havainnollistamaan esimerkiksi näkökentän pituus ympyränä agentin ympärillä, jota pystytään helposti muuttamaan Unityn omassa editorissa. Kuten kuvassa 17 näkyy, scriptiä käyttäen piirretään myös näkökentän määritetty leveys kulmana agenttia ympäröivässä ympyrässä. Tästä muodostuu agentin näkökenttä, eli FOV (Field Of View), jotta käyttäjän on helppo nähdä, minkälaisen näkökentän kanssa hän työskentelee.



Kuva 17. Custom Editorin luoma näkymä Scene-näkymässä.

Tämän jälkeen palattiin näkökentän omaan scriptiin, ja aloitettiin työskentelemään karkuriagentin tunnistavan funktion parissa. Tätä funktiota varten luotiin kaksi uutta ”LayerMask”-muuttujaa, joille määritettäisiin seinäobjektien layer, sekä karkuriagentin layer.

Näitä layereitä käyttäen tutkitaan, mikäli näkökentän ympyrän sisällä on objekti, jolla on karkuriagentin layer. Mikäli objekti löytyy, testataan, onko se näkökenttämme kulman sisäpuolella laskemalla kulma etsivän ja karkurin välillä, ja vertaamalla sitä etsivälle määritettyyn näkökentän kulmaan. Tämän jälkeen tehdään viimeinen testi lähettämällä raycast etsiväagentista kohti tunnistettua karkuriagenttia ja mikäli raycast ei osu objektiin, jolla on seinäobjektin layer, voidaan todeta, että etsivällä agentilla on suora näköyhteys karkuriagenttiin.

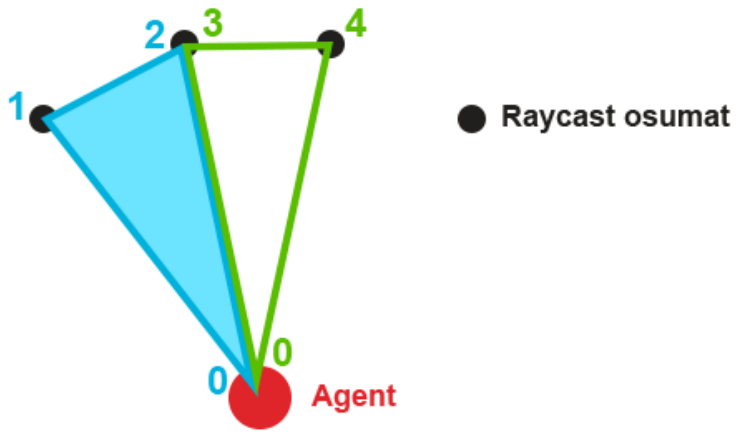
Tätä tietoa käyttämällä ei toistaiseksi tehdä vielä mitään, ja sen sijaan löydetty karkuriagentti lisätään vain ”visibleTargets”-listalle, jossa pystytään pitämään muistissa näkyvillä olevat agentit karkuriagentin ”layerillä”.

Seuraavaksi aloitettiin työskentely näkökentän visualisoimisen parissa, joka auttaisi havainnollistamaan näkökentän näkyvyyttä myös demon suorittajalle. Tätä varten luotiin uusi ”float”-tyyppinen muuttuja ”meshResolution”, jolle annettaisiin arvoksi

piirrettävän näkökentän resoluutio. Tämä resoluutio käytännössä tarkoittaa kolmioiden määrää, joista näkökenttä muodostetaan.

Näkökentän piirtämistä varten luotiin uusi funktio ”DrawFieldOfView”, jolle annettiin muuttujat ”stepCount” ja ”stepAngleSize”. Muuttuja ”stepCount” vastaa raycastien määrää, joista näkökenttä muodostuu. Muuttujalle ”stepAngleSize” lasketaan jokaisen raycastin välinen kulma jakamalla näkökentän leveys raycastien määrällä. Näin saadaan jaettua raycastit näkökentän leveydelle tasaisin välimatkoin.

Tämän jälkeen aloitetaan näkökentän piirtäminen käyttämällä Unityn ”mesh”-ominaisuutta. Kyseinen ”mesh” muodostuu kolmioista, jotka muodostetaan aloituspisteestä alkaen ensimmäiseen raycastin osumaan, siitä toiseen raycast osumaan ja takaisin aloituspisteeseen. Seuraava kolmio muodostetaan samalla tavalla, mutta alkaen toisesta raycastistä liikkuen jälleen kolmanteen raycastiin ja takaisin aloituspisteeseen. Tätä kaavaa seuraten käydään läpi jokainen raycast kellonsuuntaisessa järjestyksessä muodostaen kolmiot jokaisen raycastin välille. (kuva 18)

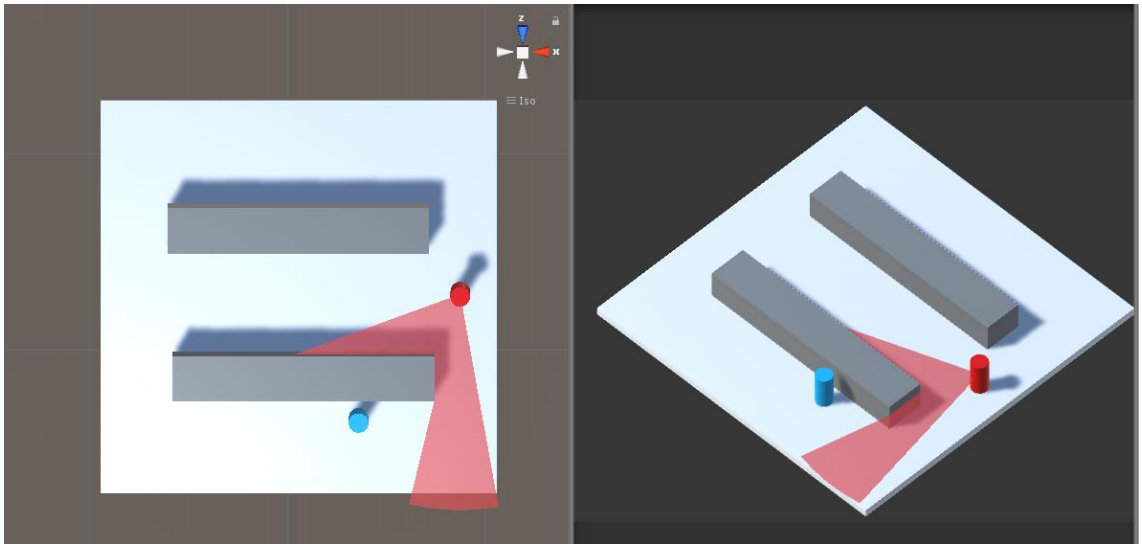


Kuva 18. Havainnollistus näkökentän kolmioiden muodostamisesta kellon suunnassa.

Näille muodostetuille kolmioille tehtiin oma lista, joka ottaa ylös jokaisen kolmion muodostavan pisteen. Tätä listaa käyttämällä pystytään myöhemmin piirtämään näkökentän muodostavat kolmiot.

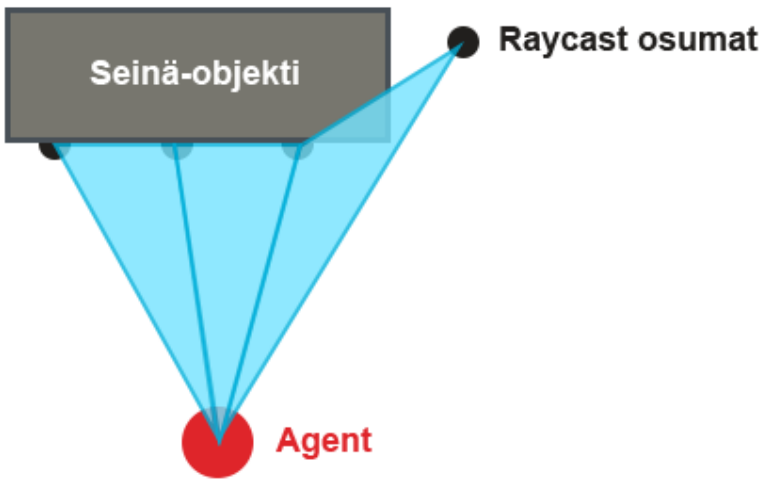
Meshiä varten tarvittiin etsivälle hahmolle tyhjä ”gameObject”, jolle lisättiin komponentit ”Mesh Filter” ja ”Mesh Renderer”. ”Mesh Renderer”-komponentista otettiin pois päältä varjojen luominen ja niiden vastaanottaminen, jotta näkökenttämme ei vaikuttuisi peliympäristön valaistuksesta.

Tämän objektin luomisen jälkeen voitiin aloittaa itse näkökentän piirtäminen. Ensimmäisenä toimena scriptiin lisättiin kohta, joka tyhjentäisi koko meshin, jotta edelliset piirretyt meshit eivät jäisi jälkeen peliympäristöön jokaisen liikkeen jälkeen. Tämän jälkeen scriptiin lisättiin toiminto ”viewMesh.vertices”, jolla määritetään jokaisen meshillä olevan pisteen sijainti. Nämä pisteet voidaan ottaa suoraan aiemmin luodusta listasta, johon kolmioiden pisteet lisättiin. Lisäksi tarvitaan myös toiminto ”viewMesh.triangles”, jolle annetaan aiemmin luodusta piste-listasta luodut kolmiot. Näiden toimintojen avulla saadaan piirrettyä mesh, josta näkökenttämme muodostuu. (kuva 19)



Kuva 19. Toimivan näkökentän havainnollistus.

Näkökentän havainnollistamisessa oli enää pieni ongelma seinäobjektien reunojen kanssa, joissa näkökenttä vilkkuisi kääntyessään objektin reunan kohdalla. Tämä johtui näkökentän mesh-resoluutiosta, joka aiheuttaisi pienen vilkkumisefektin, joka kerta kun yksi raycasteista siirtyisi pois seinäobjektilta. (kuva 20)



Kuva 20. Havainnollistus ongelman luovasta tapauksesta.

Ongelman pystyisi korjaamaan vain nostamalla resoluutiota riittävän korkealle, jotta vilkkuminen muuttuisi niin pieneksi, ettei sitä enää huomaa. Resoluution nostaminen tosin nostaisi raycastien määrää, joka taas tekee ohjelmasta turhan raskaan, siihen nähden mitä se tekee. Oppimismielessä ajateltiinkin, että olisi tärkeää opetella parempi tapa ongelman ratkaisemiseen nostamatta resoluutiota turhan suuriin lukuihin.

Tätä varten luotiin uusi ”FindEdge”-funktio, joka käyttäisi ”binary search”-algoritmin kaltaista tapaa reunan etsimiseen, jossa verrataan viimeistä raycastiä, joka osui objektiin ja ensimmäistä raycastiä, joka ei osunut objektiin. Näiden raycastien väliin lähetetään uusi raycast, ja mikäli se osuu objektiin, tulee siitä uusi viimeinen objektiin osunut raycast ja tätä kaavaa jatkamalla siirryttäisiin koko ajan lähemmäksi objektin reunaa.

Scriptiin luotiin uusi ”integer”-tyyppinen muuttuja ”edgeResolveIterations”, jolle pystyttiin antamaan arvona, kuinka monta kertaa uusi raycast lähetettäisiin kyseisten raycastien väliin. Tällä arvolla pystyttiin siten määrittämään, miten tarkasti reuna haluttiin löytää. Arvoksi muuttujalle jätettiin viisi, jolloin seinän reunaa etsittäisiin viidellä uudella raycastillä. Tämän funktion todettiin toimivan hyvin, ja näkökentän reunojen vilkkuminen saatiin poistettua kokonaan.

4.6.3 Näkökentän mukainen jahtaaminen

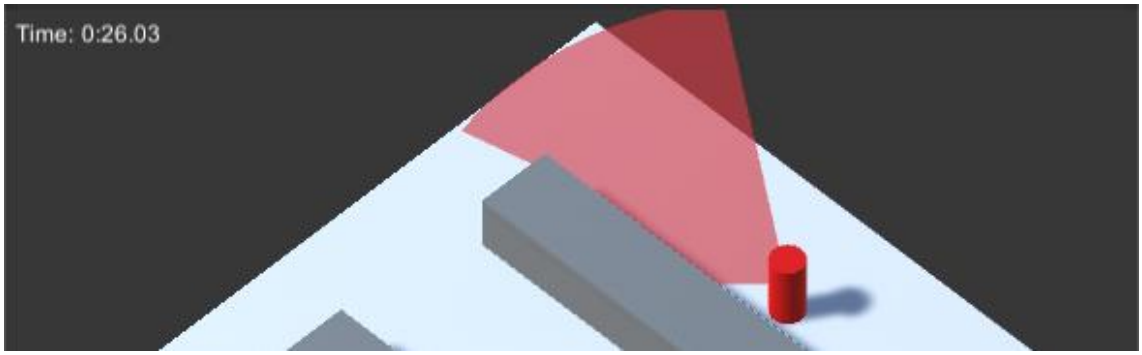
Näkökentän toimiessa mainiosti, päästiin työstämään uutta jahtaamislogiikkaa. Muutos vanhasta oli hyvin yksinkertainen, kun näkökenttää luodessa oli luotu ”visibleTargets”-lista, jossa pidettiin muistissa kaikkia ”evader”-tyyppisiä objekteja. Enää piti vain luoda referenssi ensimmäiseen listalla olevaan karkuriobjektiin ja asettaa sen sijainti uudeksi tavoitesijainniksi. Listan sijaan olisi voitu käyttää myös yksittäistä muuttujaa, mutta listan ansiosta olisi demosovellukseen helppo lisätä useampiakin karkureita.

Uusi jahtaamislogiikka toimi mainiosti, ja karkuriagentti pystyi nyt ensimmäisiä kertoja kadottamaan etsiväagentin vain juoksemalla tämän selän taakse. Mikäli etsiväagenttia haluaisi parannella, olisi sille hyvä lisätä toiminto, jonka avulla etsivä katselisi hetken ympärilleen joka kerta kun se kadottaa karkuriagentin. Tämä estäisi karkurin kadottamisen, kun agentit liikkuvat toistensa ohi, jättäen karkuriagentin etsivän taakse.

Nyt demon käyttäjän oli hyvin helppo nähdä mikäli etsiväagentti näkisi karkurin, mutta oli edelleen vaikeaa sanoa milloin karkuriagentti havaitsi etsiväagentin. Tätä varten luotiin vielä yksi yksinkertainen raycast, joka testaa mikäli karkuriagentti on tietyn etäisyyden päässä etsiväagentissa ja mikäli niiden välissä ei ole mitään esteitä. Raycastiin siirrettiin myös karkurin käyttämä karkaamislogiikka, joka oli ennen toiminut vain etäisyyttä etsivään mittaamalla. Karkurin liikkeiden tulkitsemista helpottamaan kehitettiin hyvin yksinkertainen värikoodaus, jossa karkuriagentin havaitessa etsiväagentin, muuttuu karkurin väri pinkiksi ja päästessään riittävän kauaksi etsivästä, muuttuu se takaisin siniseksi. Tämä teki demosovelluksen seuraamisesta huomattavasti helpompaa, kun oli helposti nähtävissä, milloin karkuriagentti yritti piiloutua, ja milloin se vain vaelteli satunnaisesti.

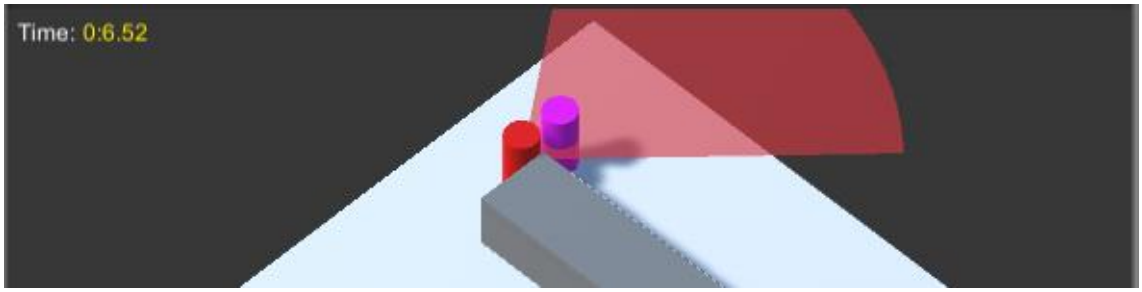
4.7 **Ajastin**

Sovellukseen haluttiin vielä ajastin kertomaan, kuinka kauan hippaleikki oli kestänyt. Projektiin lisättiin tekstikenttänä toimiva UI-elementti (User Interface), johon saatiin pienellä scriptillä näkyviin, kuinka kauan sovellus oli ollut käynnissä. (kuva 21) Tähän käytettiin Unityn ”Time.time”-toimintoa, jota käyttämällä saadaan sekunneissa, kuinka kauan sovellus on ollut käynnissä.



Kuva 21. Ajastinelementti sovelluksen vasemmassa yläkulmassa.

Ajastimeen haluttiin lisätä vielä pysähtymistoiminto. Tämä luotiin lisäämällä toiselle agenteista Unityn rigidbody-komponentti, jolloin pystyttiin tutkimaan agentin kontaktia muihin objekteihin. Kun todettiin että etsiväagentti törmäsi karkuriagenttiin, muutettaisiin ajastimelle luotu ”finished”-booleanarvo todeksi, joka pysäyttäisi ajastimen ja muuttaisi samalla tämän värin keltaiseksi. (kuva 22)



Kuva 22. Ajastinelementti sen pysähdyttyä.

Ajastimen formaattia varten käytettiin pientä scriptiä, jotta voitiin erotella minuutit omaan lokeroonsa, estäen sekunteja laskemasta yli kuudenkymmenen. Tämä helpottaisi ajastimen lukemista myös pidemmillä suorituskerroilla. (kuva 23) Tämä luotiin yksinkertaisesti jakamalla sekuntit kuudellakymmenellä saaden näkyviin minuutit ja lisäämällä sekunteille oma osio, johon sisällytettiin myös kaksi desimaalia, saaden ajastimen näyttämään paljon tarkemmalta.

```
float t = Time.time - startTime;

string minutes = ((int)t / 60).ToString();
string seconds = (t % 60).ToString("f2");

timerText.text = minutes + ":" + seconds;
```

Kuva 23. Ajastimen minuuttien ja sekuntien erottelu.

5 YHTEENVETO

Opinnäytetyötä työstäessä opittiin, että Unityn valmiilla työkaluilla pystyy helposti heidän omia ohjeitaan seuraten valmistamaan peliin tekoälyn, joka pystyy navigoimaan pelialueella täysin itsenäisesti käyttämällä Unityn NavMesh-ominaisuuksia. Nämä ominaisuudet ovat kuitenkin vain reitinhakuratkaisuja ja kaikki lisäominaisuudet tekoälylle on rakennettava itse.

Vaihtoehtoisia ratkaisuja NavMeshin käyttämiselle Unityssä on huomattavasti vaikeampi hyödyntää, eikä todettu mitään syytä olla käyttämättä helposti saatavilla olevia NavMesh-ominaisuuksia. Esimerkkejä vaihtoehtoisista reitinhakujärjestelmistä olisi esimerkiksi järjestelmä, johon määriteltäisiin lukuisia sijaintipisteitä, jonka jälkeen ohjelmoitaisiin funktio, joka laskisi lyhimmän reitin tavoitesijainnin luokse näiden pisteiden kautta.

Opinnäytetyössä saatiin valmiiksi kaikki ominaisuudet, joita alkuperäisissä ideoissa oli pohdittu. Näitä olivat esimerkiksi reitinhaun käyttöönottaminen, toimiva interaktio agenttien välillä, visuaalinen näkökenttä, sekä ajastin, joka mittaa miten kauan etsivällä agentilla kestää saada karkuriagentti kiinni.

LÄHTEET

Algfoor, Z. (2015). A Comprehensive Study on Pathfinding Techniques for Robotics and Video Games. Haettu 21.01.2020 osoitteesta https://www.hindawi.com/journals/ijcgt/2015/736138/

Amit, (2019). Introduction to A*. Haettu 28.01.2020 osoitteesta http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html

Arongranberg. (n.d.) A* Pathfinding Project. Haettu 28.01.2020 osoitteesta https://arongranberg.com/astar/features

Bevilacqua, F. (2013). Finite-State Machines: Theory and Implementation. Haettu 21.01.2020 osoitteesta https://gamedevelopment.tutsplus.com/tutorials/finite-state-machines-theory-and-implementation--gamedev-11867

Elements of AI. (n.d.) Miten tekoäly määritellään? Haettu 14.01.2020 osoitteesta https://course.elementsofai.com/fi/1/1

Fine, R. (2017). UnityScript's long ride off into the sunset. Haettu 04.02.2020 osoitteesta https://blogs.unity3d.com/2017/08/11/unityscripts-long-ride-off-into-the-sunset/

Heikkinen, S. (2017). Tekoäly muuttaa maailman – pian se tekee jopa lääkärin ja juristin töitä. Haettu 14.01.2020 osoitteesta https://yle.fi/aihe/artikkeli/2017/06/04/tekoaly-muuttaa-maailman-pian-se-tekee-jopa-laakarin-ja-juristin-toita

Junttila, H. (2015). *Unity3D:n käyttö pelikehityksessä.* Opinnäytetyö. Tietotekniikan koulutusohjelma. Oulun seudun ammattikorkeakoulu. Haettu 04.02.2020 osoitteesta https://www.theseus.fi/bitstream/handle/10024/87939/Junttila_Henri.pdf?sequence=1&isAllowed=y

Krupp, H. (2014). *Game Mechanics of Unity 2D Game.* Thesis. Degree Bachelor of Engineering. Helsinki Metropolia University of Applied Sciences. Haettu 04.02.2020 osoitteesta https://www.theseus.fi/bitstream/handle/10024/81393/Bachelor%20Thesis%20final%20Hans%20Jurgen%20Krupp%20%2017.10.pdf?sequence=1&isAllowed=y

Kukkonen, A. (2010). *Reitinhaku ja tekoälyn* päätöksenteko kaksiulotteisessa videopelissä. Opinnäytetyö. Ohjelmistotekniikan koulutusohjelma. Rovaniemen Ammattikorkeakoulu. Haettu 21.01.2020 osoitteesta https://www.theseus.fi/bitstream/handle/10024/21342/opinnaytetyo_reitinhaku_ja_tekoalyn_paatoksenteko_kaksiulotteisessa_videopelissa.pdf?sequence=1&isAllowed=y

Lester, P. (2005). A* Pathfinding for Beginners. Haettu 28.01.2020 osoitteesta http://csis.pace.edu/~benjamin/teaching/cs627/webfiles/Astar.pdf

Maass, L. (2019). Artificial Intelligence in Video Games. Haettu 14.01.2020 osoitteesta https://towardsdatascience.com/artificial-intelligence-in-video-games-3e2566d59c22

Nield, T. (2019) The practical value of game AI. Haettu 14.01.2020 osoitteesta https://towardsdatascience.com/ai-research-and-the-video-game-fetish-71cb62ffd6b3

Sreedhar, S. (2007). Checkers, Solved! Haettu 16.01.2020 osoitteesta https://spectrum.ieee.org/computing/software/checkers-solved

Unity3D. (2020) Unity Store. Haettu 04.02.2020 osoitteesta https://store.unity.com/compare-plans

Unity3D. (2020) Unparalleled platform support. Haettu 04.02.2020 osoitteesta https://unity3d.com/unity/features/multiplatform

Unity. (n.d.) NavMesh Baking. Haettu 04.02.2020 osoitteesta https://learn.unity.com/tutorial/navmesh-baking#5ce58ae8edbc2a0ff9332558

Unity. (n.d.) NavMesh building components. Haettu 04.02.2020 osoitteesta https://docs.unity3d.com/Manual/NavMesh-BuildingComponents.html