Tampereen ammattikorkeakoulu

Tekoälyn tekeminen 2D-peliin Unityn Entity Component Systemiä hyödyntäen

Valtteri Ojanen

OPINNÄYTETYÖ
Marraskuu 2020

Tietojenkäsittely
Pelituotanto

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietojenkäsittely
Pelituotanto

OJANEN, VALTTERI:
Tekoälyn tekeminen 2D-peliin Unityn Entity Component Systemiä hyödyntäen

Opinnäytetyö 29 sivua
Marraskuu 2020

Opinnäytetyössä tutustuttiin Unity-pelinkehitysalustan uuteen teknologiapakettiin, joka koostuu kolmesta eri osasta. Opinnäytetyössä tutkittiin, miten teknologiapaketti vaikuttaa tekoälyn tekemiseen ja minkälaisia hyötyjä ja haittoja siinä ilmenee. Lisäksi selvitettiin, miten se vaikuttaa työskentelyyn. Opinnäytetyössä käytettiin Unityn 2019.1.10f1 -versiota ja koodin kirjoittamiseen käytettiin Microsoft Visual Studiota sekä C#-ohjelmointikieltä.

Opinnäytetyön aikana tehtiin 2D-pelin prototyyppi, jossa hyödynnettiin Unityn uuden teknologiapaketin osia. Teknologiapakettia hyödyntämällä pystyttiin kirjoittamaan monisäikeistä koodia, joka paransi pelin suorituskykyä. Lisäksi se auttoi rakentamaan koodia selkeämmin, mikä teki koodista helppolukuista. Myös ohjelmointivirheiden havaitseminen oli helpompaa.

Teknologiapaketti on vielä keskeneräinen, siitä puuttuu monia tärkeitä ominaisuuksia, joita pelisuunnittelussa tarvitaan. Teknologiapaketissa on myös hyötyjä, ja jos sitä kehitetään eteenpäin, se on tulevaisuudessa pelisuunnittelun toimiva työkalu.

Asiasanat: unity, unity dots, tekoäly, pelinkehitys, c#

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Business Information Systems
Game Production

OJANEN, VALTTERI:
Making AI for a 2D Game Using Unity's Entity Component System

Bachelor's thesis 29 pages
November 2020

This thesis focuses on Unity's new technology stack. This new technology stack consists of three parts. This thesis studies how the technology stack affects making a videogame AI, what are the pros and cons of it, and how it affects workflow. The practical part of this thesis was made using Unity 2019.1.10f1 along with Visual Studio and C#.

A 2D game prototype was made during the thesis using parts of Unity's new technology stack. By utilizing these new technologies, the game's performance increased, since parts of the code used multithreading. The technologies helped structuring the code, which made the code more readable and finding bugs was easier.

In conclusion, although the new technology stack was used in the project, but it is still in early stages, so it is still missing important parts for game development. If the new technology stack was developed more, it would surely be used in the future. It showed great benefits in the game protype, even though it was still under development.

Key words: unity, unity dots, ai, game development, c#

SISÄLLYS

ERITYISSANASTO tai LYHENTEET JA TERMIT (valitse jompikumpi)

DOTS	Data-oriented technology stack
ECS	Entity Component System
VR	Virtuaalitodellisuus
Algoritmi	Joukko ohjeita tehtävän suorittamiseen
Agentti	Videopelissä oleva hahmo

1 JOHDANTO

Tämän Opinnäytetyön tavoitteena oli tutustua Unityn uuteen teknologiapakettiin, Data-Oriented Technology Stack:iin (DOTS), sekä sen hyödyntämiseen tekoälyn tekemisessä 2D-peliin. DOTSin hyödyllisyyttä tutkittiin tekemällä 2D-pelin prototyyppi, jossa DOTSia käytettiin tekoälyn tekemisessä. Pelissä pelimoottorina toimi Unity.

Opinnäytetyön teoriaosuudessa käydään läpi Olio-ohjelmonnin perusteet, jota käytetään Unityn perinteisessä järjestelmässä. Tämän jälkeen siirrytään DOTSiin, jossa puhutaan sen kolmesta eri osasta. Lopuksi perehdytään hieman videopelien tekoälyyn ja reitinhakuun. Toteutusosuudessa käydään läpi, kuinka pelin tekoäly toteutettiin ja miten DOTS vaikutti toteutukseen. Viimeisenä pohdintaosuudessa tarkastellaan DOTSin hyödyllisyyttä, sekä DOTSin sen hetkistä tilannetta.

Työllä ei ollut toimeksiantajaa, vaan se tehtiin omaksi eduksi. Opinnäytetyötä saa hyödyntää kuka tahansa aiheesta kiinnostunut.

2 Unity

2.1 Unity pelinkehitysalustana

Unity on maailman suosituin pelimoottori. Siitä löytyy tuhansia eri toimintoja ja sillä pystyy tehdä melkein minkä vain pelin. Unity on suosittu sekä aloittelijoiden, että isojen pelistudioiden keskuudessa. Sitä on käytetty monen tunnetun pelin tekemiseen kuten HearthStone, Pokemon Go, Rimworld ja Cuphead (Petty n.d.). Unity on suosittu myös Virtual Reality (VR) kehityksen parissa, sillä se tarjoaa useita työkaluja ja valmiita toimintoja siihen.

Unityssä käytetään skriptejä kehittämään melkein kaikki osat pelistä tai muista vuorovaikutteisista sisällöistä. Unity käyttää C#-koodikieltä. Kaksi yleisintä tyyliä suunnitella koodirakennetta on olio-ohjelmointi malli, joka on tavanomainen lähestymistapa ja sitä käytetään paljon. Toinen tapa on dataorientoitunut malli, jota käytetään huippu tehokkaan monisäikeisen Data-Oriented Technology Stackin (DOTS) kautta (Unity Technologies 2020a.). Unity tarjoaa ohjelmointiin natiivin Visual Studio -integraation.

2.2 Olio-ohjelmointi

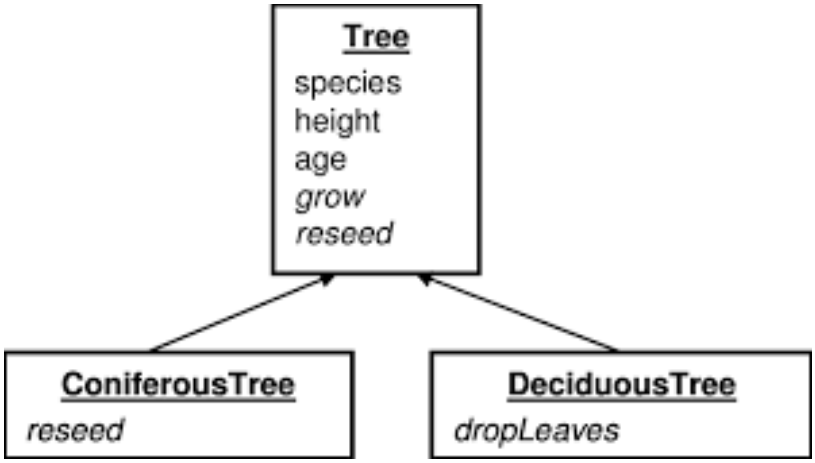
Kenneth Kuvailee Olio-ohjelmointia ajatusmallina, jossa ratkaisu ohjelmointiongelmaan on kokoelma yhdessä toimivia objekteja. Objektit toimivat yhdessä lähettämällä viestejä toisilleen. Sen paras käyttötarkoitus on isot ja monimutkaiset ongelmat (Kenneth 2017.)

Objekti on kokonaisuus, joka sisältää dataa ja funktioita. Data on yleensä piilotettu muilta objekteilta, jotta ainut tapa muokata objektin dataa on funktioiden kautta (Kenneth 2017.)

2.2.1 Periytyminen

Periytyminen on yksi Olio-ohjelmoinnin tärkeimpiä konsepteja. Se mahdollistaa yhteyksien ja hierarkioiden luomisen luokkien välillä. Periytymisessä on kaksi eri

luokkatyyppiä: Alaluokka ja Pääluokka. Alaluokat perivät pääluokan attribuutit ja metodit. Periyttäminen siis vähentää tarvetta kirjoittaa samoja koodin osia useita kertoja (kuva 1). Jos pääluokkaa muokataan, niin myös kaikki siitä periytetyt alaluokat perivät muutokset. Sen sijaan alaluokkaa muokatessa pääluokkaan ei tule mitään muutoksia. Periyttämistä voi jatkaa vielä alaluokasta eteenpäin, jolloin se perii pääluokan ja sen alaluokan ominaisuudet. (Educba n.d.)



KUVA 1. Periytyminen (Janssen n.d.).

2.2.2 Kapselointi

Kapseloinnissa poistetaan suora pääsy dataan ja data on sen sijaan piilotettu. Dataa ei muokata suoraan, vaan sitä muokataan objektin kautta, jolle data kuuluu. Objektille määritellään kuinka dataa voi muokata. Kun dataa halutaan muokata, objektille lähetetään pyyntö. Tämän jälkeen objekti määrittää, onko pyyntö validi. Pyynnön ollessa validi, muokkaa objekti dataa pyynnön mukaisesti ja lähettää lopputuloksen takaisin viestinä. Datan kapseloidessa siitä tulee paljon luotettavampaa. Silloin tiedetään, miten dataan pääsee käsiksi ja miten sitä muokataan. Tämä helpottaa ohjelmiston ylläpitoa, kun dataa ei käsitellä monesta paikasta, joten päivitysten teko koodiin hoituu helpommin. (Clark 2011.)

2.2.3 Abstraktio

Abstraktio on tärkeä osa olio-ohjelmointia. Sen avulla pystytään piilottamaan tarpeetonta tietoa käyttäjältä, täten vähentäen koodin monimutkaisuutta. Sen ympärille voidaan rakentaa toiminnallisuutta, tietämättä miten abstraktin luokan metodit toimivat. Käyttäjän tarvitsee ainoastaan tietää, mitä metodeja siinä on käytettävissä, minkälaista dataa ne ottavat vastaan ja minkälainen lopputulos on odotettavissa. (Janssen 2017.)

2.2.4 Polymorfismi

Polymorfismilla tarkoitetaan objektin kykyä ottaa monta eri muotoa. Polymorfismia on kahta erilaista. Yleensä polymorfismista puhuttaessa tarkoitetaan dynaamista polymorfismia, jossa yläluokan referenssiä käytetään alaluokan objektiin. Kun luokka hierarkiassa useammalla luokalla on sama metodi ja sitä kutsutaan, ajetaan objektin luokan metodi, eikä luokka referenssin metodi. Staattinen polymorfismi tarkoittaa, että luokalla on monta saman nimistä metodia, jotka käyttävät eri argumentteja. Tässä tilanteessa kääntäjä päättää kääntäessä mitä metodia tulee käyttää, riippuen siitä mitä parametrejä annetaan metodia kutsuessa. (Ediriweera 2017.)

3 DOTS

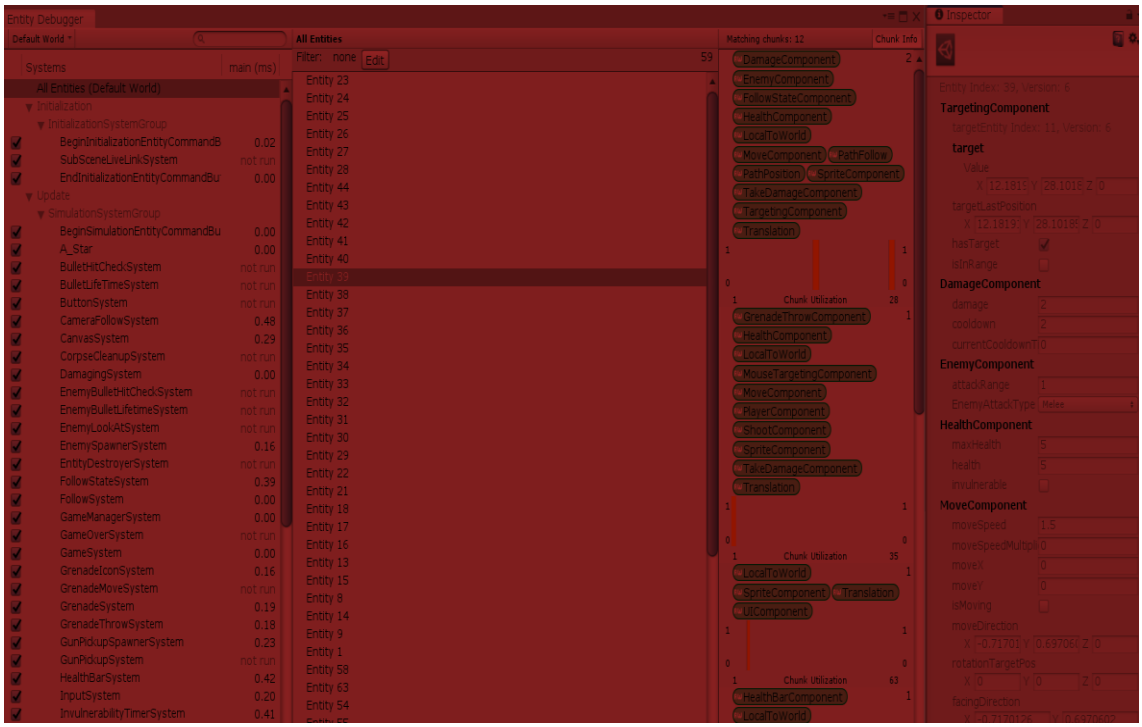
Moni asia on muuttunut pelituotannossa viimeisen vuosikymmenen aikana, mutta yksi isoimmista muutoksista on, että ennen 90 % koodista toimi vain yhdessä säikeessä ja yhdessä ytimessä. Nykyään kaikki kantavat taskussaan satoja prosessorin ytimiä, ja pelituotannon pitää mukautua tähän ja käyttää monisäikeisyyttä hyväkseen. Data-Oriented Technology Stack (DOTS) on Unityn yritys uudelleenrakentaa sen sisäistä arkkitehtuuria tavalla, joka on nopeampi, kevyempi ja tärkeimpänä: se on optimoitu monisäikeisyydelle. DOTS sisältää kolme osaa: Entity Component System (ECS), Job System ja Burst compiler. (Aversa 2019.)

3.1 Entity Component System

Entity Component System (ECS) on Data-Oriented Technology Stack:in (DOTS) ydin. Niin kuin nimestä voi päätellä, ECS pitää sisällään kolme eri osaa: entiteetit, komponentit ja systeemit. ECS-arkkitehtuuri erottelee identiteetin (entiteetit), datan (komponentit) ja käyttäytymisen (systeemit). Tässä arkkitehtuurissa keskitytään dataan. Systeemit työskentelevät komponenttien kanssa, ne muokkaavat niiden dataa, entiteetit puolestaan indeksoivat komponentit. (Unity Technologies 2020b.)

ECS tuo mukanaan oman debuggerin (kuva 2). Tämä debuggeri havainnollistaa entiteetit, systeemit ja komponentit. Debuggerin saa auki Unityn sisällä painamalla Window > Analysis > Entity Debugger. Systeemit-lista näyttää kaikki projektin systeemit ja kuinka kauan systeemien ajamiseen käytetään aikaa. Systeemit-listasta yksittäisiä systeemejä pystyy käynnistämään tai sammuttamaan. Systeemin yksityiskohdat -lista näyttää komponenttiryhmät, joita kyseinen systeemi käsittelee ja listan entiteeteistä, joilta kyseinen komponentti löytyy. Jos EntityManager on valittuna systeemit-listasta, näyttää systeemin yksityiskohdat -lista kaikki entiteetit. Valitsemalla systeemin yksityiskohdat listasta jonkin enti-

teetin, näyttää Unityn Inspector-ikkuna kyseisen entiteetin komponentit ja komponenttien arvoja pystyy tarkastelemaan. (Unity Technologies 2020b.)



KUVA 2. Entity Debuggeri.

3.1.1 Entiteetit

Entiteetit ovat yksi kolmesta periaatteesta ECS-arkkitehtuurissa. Ne kuvastavat yksilöllisiä asioita pelissä tai ohjelmassa. Entiteetti ei sisällä dataa tai toiminnallisuutta. Se sisältää datan yhteenkuuluvat osat. Entiteetti on käytännössä tunniste, joka on ainut vakaa tapa referoida komponenttia tai toista entiteettiä. EntityManager pitää listaa kaikista entiteeteistä ja organisoi niihin liittyvää dataa optimaalisen performanssin saavuttamiseksi. (Unity Technologies 2020b.)

Entiteeteillä ei ole tyyppiä, mutta entiteettijoukko voidaan luokitella niille kuuluvien komponenttien perusteella. Kun entiteettejä luodaan ja niihin lisätään komponentteja, pitää EntityManager kirjaa kaikista uniikeista komponenttiyhdistelmistä. Tällaisia yhdistelmiä kutsutaan arkkityypeiksi. Olemassa olevia arkkityyppejä voi käyttää uusien entiteettien luomiseen, jolloin siinä on kaikki komponentit valmiiksi. Arkkityyppejä voi luoda myös etukäteen, ja käyttää niitä entiteettien luomiseen. (Unity Technologies 2020b.)

3.1.2 Komponentit

Komponentit ovat tietueita, jotka pitävät sisällään dataa ja mahdolliset apumetodit datan hakuun, muuta toiminnallisuutta niissä ei ole. Komponentin tulee toteuttaa jotain vaadituista rajapinnoista. Rajapintoja on useita ja kaikilla on oma käyttötarkoitus. Komponentit tallennetaan Chunk-muistiin, jossa ei ole roskien keruuta, joten komponentit eivät voi pitää sisällään viittauksia hallittuihin objekteihin. (Unity Technologies 2020b.)

Yleisimmin käytetty komponentti on yleiskomponentti (General Purpose Component). Yleiskomponentti sisältää ainoastaan instanssidataa entiteeteille. Jaettu komponentti (Shared Component) on erityinen komponentti, jota käytetään, jos entiteeteillä on jotain yhteistä. Jaettu komponentti pystyy arkkityyppien lisäksi jakamaan entiteettejä alaluokkiin, perustuen niiden yhteisiin arvoihin. EntityManager asettaa kaikki entiteetit, joilla on yhteiset arvot, samaan Chunk-muistiin. EntityManager liikuttaa entiteettejä eri Chunk-lohkoon, mikäli jaetun komponentin arvoja muutetaan tai entiteetiltä lisätään tai poistetaan komponentti. Tarvittaessa EntityManager luo uuden Chunk-lohkon. Systeemintilakomponentti (System State Component) ja jaettu systeemintilakomponentti (System State Shared Component) ovat kuten yleiskomponentti ja jaettu komponentti, mutta niillä pystyy seuraamaan systeemin sisäisiä resursseja sekä lisäämään ja tuhoamaan kyseisiä resursseja turvautumatta takaisinkutsuihin. Mikäli entiteetillä on systeemintilakomponentti tai jaettu systeemintilakomponentti ja entiteetti poistetaan, tavallisesti kaikki komponentit, jotka viittaavat kyseiseen entiteettiin poistettaisiin ja entiteetin tunniste käytettäisiin uudelleen. Sen sijaan vain tavalliset komponentit poistetaan, mutta tunnistetta ei uudelleen käytetä, ennen kuin systeemintilakomponentit poistetaan manuaalisesti. Dynaaminen puskurikomponentti (Dynamic Buffer Component) mahdollistaa erikokoisten muistipuskurien lisäämisen entiteeteille. Dynaamisella puskurikomponentilla on sisäinen kapasiteetti tietylle määrää elementtejä. Sisäisen kapasiteetin täyttyessä se pystyy varaamaan itselleen heap-muisti lohkon. Muistinhallinta on täysin automatisoitu Entity Managerin toimesta: kun dynaaminen puskurikomponentti poistetaan, myös kaikki sille varattu heap-muisti vapautetaan automaattisesti. Lohkokomponentti (Chunk component) sisältää dataa, joka koskee kaikki entiteettejä tietyssä

chunk-lohkossa. Jos lohkokomponentin arvoa muutetaan entiteetin kautta, muuttaa se kaikkien lohkokomponenttien arvoa kyseisessä lohkossa. (Unity Technologies 2020b.)

3.1.3 Systeemit

Kaikki pelin toiminnallisuus tapahtuu systeemeissä, niissä käytetään ja muutetaan komponenttien dataa. Unityn Entity Component System tarjoaa muutamia eri systeemejä, mutta kaksi niistä on yleisesti käytettyjä: ComponentSystem ja JobComponentSystem. Molemmat näistä systeemeistä helpottavat entiteettien valitsemista ja iteroimista, riippuen mitä komponentteja näihin entiteetteihin kuuluu. Systeemeissä käsitellään entiteettejä luomalla EntityQuery-tietue, missä määritellään mitä komponentteja entiteetillä pitää kuulua. Kun entiteetit EntityQueryssä määrittelemillä komponenteilla on löydetty, niitä iteroidaan ja niille kuuluvien komponenttien dataa käsitellään. (Unity Technologies 2020b.)

Systeemit tarjoavat useita takaisinkutsu-funktioita, kuten OnCreate ja OnUpdate. Nämä funktiot ajavat koodia tiettyihin aikoihin. Näitä funktioita voidaan hyödyntää implementointeja tehdessä. OnCreate ja OnUpdate ajetaan pääsäikeessä. JobComponentSysteemissä tyypillisesti aikataulutetaan tehtäviä OnUpdate-funktiossa, tämän jälkeen tehtävät ajetaan työläissäikeissä. JobComponentSystem mahdollistaa parhaan suorityskyvyn, koska se hyödyntää monia prosessorin ytimiä. Suorituskykyä voidaan parantamaan vielä entisestään, kun tehtävät on käännetty käyttämällä Burst Compileriä. Entity Component System havaitsee kaikki systeemiluokat automaattisesti ja luo ne ajon aikana. Automaattisen luomisen pystyy kieltämään antamalla systeemille Systeemi-attribuutin. (Unity Technologies 2020b.)

3.2 C# Job System

Monisäikeisyys on ohjelmointityyppi, joka hyödyntää prosessorin kapasiteettia prosessoida useaa säiettä samanaikaisesti monessa ytimessä. Monisäikeistä koodia voi kirjoittaa itse, mutta se on vaikeaa, ja siinä tulee helposti virheitä, kuten kilpailutilanteet. Kilpailutilanne tarkoittaa, että koodi toimii eri lailla, riippuen

missä järjestyksessä se ajetaan. Säikeitä voi aloittaa missä järjestyksessä tahansa, mutta se ei ole varmaa, että ne ajetaan oikeassa järjestyksessä. Tämän seurauksena voi tapahtua odottamattomia bugeja, varsinkin jos kaksi säiettä tai enemmän käsittelee ja muokkaa samaa dataa. (Amat 2018.)

Job Systemin avulla pystyy kirjoittamaan monisäikeistä koodia helposti ja turvallisesti. Se tapahtuu luomalla tehtäviä säikeiden sijaan. Tehtävä kuvastaa työmäärää, jonka systeemi voi prosessoida osissa. Kun tehtävä aikataulutetaan, Job System asettaa sen jonoon. Ajon aikana työläissäie ottaa tehtävän jonosta ja suorittaa sen. Työläissäikeet ovat yksittäisiä säikeitä, joita Job System hallitsee. Jotta pääsäiettä ei keskeytetä, tehtävät suoritetaan taustalla. (Amat 2018.)

3.3 Burst Compiler

Burst Compiler on koodigeneraattori, joka kääntää C#-kielen alaluokan, jota kutsutaan High-Perforamance C# tai HPC#-konekieleksi, joka on usein lyhyempää ja nopeampaa kuin C++ kirjoitettu koodi. (Aversa 2019.) Burst Compiler toteuttaa tämän käyttämällä hienostunutta teknologiaa, joka hyödyntää avoimen lähdekoodinprojektia nimeltä LLVM (Amat 2019.)

4 Tekoäly peleissä

Gameranx:n mukaan videopelien tekoälyllä ei välttämättä tarkoiteta todellista tekoälyä, kuten itseoppivat robotit tai Googlen DeepMind. Videopelien tekoäly toimii vain siinä ympäristössä, missä sen on tarkoitus toimia. Esimerkiksi rallipeliin tehty tekoäly ei toimisi ensimmäisen persoonan ammuntapelissä. Videopelien tekoäly koostuu ehtolauseista, joiden mukaan tekoäly toimii. Tietyn ehdon tai ehtojen täytyttyä tekoäly toimii sen mukaisesti. Hyvin tehty videopelin tekoäly saattaa vaikuttaa todelliselta tekoälyltä, koska se reagoi pelaajan toimintoihin niin kuin se ajattelisi ja reagoisi sen mukaan. Hyvin alkeellista tekoälyä käytetään joskus videopeleissä esimerkiksi pelaajan toimintojen seuraamiseen ja sen seurauksena tekoälyn taktiikoiden vaihtamiseen. (Gameranx 2016.)

4.1 Rajallinen tilakone

Rajallinen tilakone on laskentamalli, jossa on useampi tila sekä tieto mahdollisista syötteistä. Ainoastaan yksi tila voi olla aktiivinen samanaikaisesti. Rajallinen tilakone aloittaa laskennan aina aloitus tilasta ja siirtyy seuraavaan tilaan hyväksyttävän syötteen saadessa. Rajallisen tilakoneen toiminnallisuus voi olla joko tilan sisällä, joka tunnetaan nimellä Moore machine, tai siirryttäessä tilasta toiseen, joka tunnetaan nimellä Mealy machine. Pelinkehityksessä käytetään yleensä näiden kahden yhdistelmää, jolloin toiminnallisuutta on tilojen sisällä ja siirtyessä tilasta toiseen. (Rabin 2006.)

5 Reitinhaku

Reitinhaun tarkoitus on löytää lyhin reitti kahden pisteen välillä. Reitinhakua hyödynnetään monella eri alalla kuten videopeleissä, robotiikassa ja karttapalveluissa. Videopeleissä agentit liikkuvat usein käyttäen reitinhakua, jolla ne löytävät reitin sen hetkisestä paikasta haluttuun päämäärään. Ongelmana on sopivan ratkaisun löytäminen reitinhakuun. Reitin löytämiseen käytettäviä tekniikoita kutsutaan reitinhakualgoritmeiksi. Algoritmeja käytetään lyhimmän ja optimaalisimman reitin löytämiseen. Reitinhakualgoritmeja on useita, mutta yleisimmät niistä on Dijkstra ja A*. (Rafiq 2020.)

5.1 A* reitinhaku algoritmi

A* on variaatio paras ensin hakualgoritmista. A* on optimaalinen ja valmis hakualgoritmi. Tämä saavutetaan hyödyntämällä heuristisia metodeja. Kun hakualgoritmia sanotaan optimaaliseksi, sillä tarkoitetaan, että algoritmi löytää aina optimaalisimman reitin. Videopeleissä tämä tarkoittaa yleensä nopeinta reittiä. Valmis hakualgoritmi tarkoittaa, että algoritmi löytää aina reitin, jos sellainen on olemassa. (Batoćanin n.d.)

A*:n toiminta perustuu halvimman hintaisen reitin löytämiseen hakupuusta. A* etsii reittiä käyttämällä funktiota:

$$f(n) = g(n) + h(n),$$

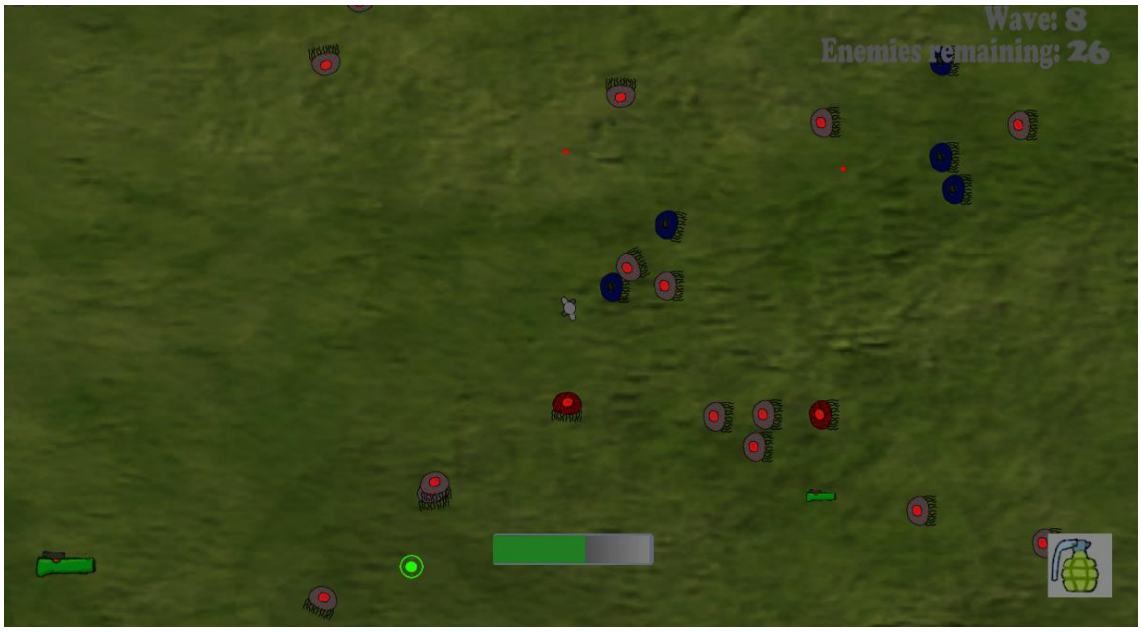
jossa f(n) on reitin yhteenlaskettu hinta solmua n käytettäessä. G(n) on hinta solmuun n mentäessä ja h(n) on arvio reitin hinnasta solmusta n määränpäähän. A* aloittaa alkusolmusta ja listaa kaikki ympärillä olevat solmut. Listan valmistuttua, sieltä poistetaan kaikki saavuttamattomat solmut, kuten seinät ja muut esteet. Tämän jälkeen valitaan solmu, jossa on halvin f(n) arvio. Tätä prosessia toistetaan rekursiivisesti, kunnes saavutetaan määränpää. (Brilliat.org n.d.)

6 Pelin Toteutus

Opinnäytetyön yhteydessä tehtiin peliprototyyppi. Peli tehtiin Unityn 2019.1.10f1 -versiolla käyttäen C# koodikieltä. Peli tehtiin käyttäen Entity Component Systemiä muutamia pieniä osia luukuunottamatta, jotka tehtiin käyttämällä Monobehaviour-luokkaa. Peli tehtiin Microsoft Windows -alustalle ja se on genreltään ylhäältä päin kuvattu räiskintäpeli. Pelissä ei ole moninpeli mahdollisuutta.

6.1 Pelitila

Pelissä pelaaja ohjaa yhtä hahmo. Pelaajan on tarkoitus selvitä vihollisaalloista niin pitkään kuin mahdollista. Pelaajan hahmo pystyy liikkumaan, ampumaan ja heittämään kranaatteja. Pelin alkaessa pelaajalla on aseena pistooli, mutta pelin aikana kartalle ilmestyy erilaisia aseita. Pelaajan poimiessa aseen, vaihtuu sen hetkinen ase poimittuun aseeseen. Pelikentälle ilmestyy satunnaisesti parannuksia. Parannuksia on monta erilaista kuten hetkellinen kuolemattomuus ja elämäpisteiden parannus. Viholliset luodaan pelikentän reunoille ja ne liikkuvat sieltä pelaajaa kohti. Vihollisten tarkoitus on vahingoittaa pelaajaa. Pelin edetessä vihollisten määrä kasvaa ja vastaan tulee uusia vihollistyyppejä (kuva 3). Pelaajan elämäpisteiden loputtua pelaajan hahmo kuolee ja peli loppuu.



KUVA 3. Pelitila.

6.2 Vihollisten toteutus

Pelin vihollisentiteetit luodaan vihollisarkkityypistä. Niissä on useita komponentteja, jotka sisältävät tarpeellisen datan vihollisista, kuten elämäpisteet ja sijainti (kuva 4). Vihollisentiteettien luominen tapahtuu EnemySpawnerSystem-systeemissä, EnemySpawnerSystem-systeemi listaa kaikki luodut vihollisentiteetit. Uuden vihollisaallon alkaessa listassa olevat vihollisentiteetit poistetaan, lista tyhjätään ja uusi lista alustetaan.

```
enemyArchetype = EntityManager.CreateArchetype
    (
    typeof(Translation),
    typeof(LocalToWorld),
    typeof(MoveComponent),
    typeof(EnemyComponent),
    typeof(HealthComponent),
    typeof(DamageComponent),
    typeof(TargetingComponent),
    typeof(TakeDamageComponent),
    typeof(SpriteComponent),
    typeof(FollowStateComponent),
    typeof(PathPosition),
    typeof(PathFollow)
    );
```

KUVA 4. Vihollisarkkityyppi.

Vihollisaallossa on aina tietty määrä vihollisia. Vihollisten määrä kasvaa, mitä pidemmälle pelissä edetään. Vihollisia luodaan kerrallaan satunnainen määrä (kuva 5).

```
Entities.ForEach((ref EnemySpawnerComponent enemySpawnerComponent) =>
{
    if (enemySpawnerComponent.currentWaveKillCount < enemySpawnerComponent.currentWaveEnemyCount)
    {
        if (enemySpawnerComponent.currentWaveEnemiesSpawned < enemySpawnerComponent.currentWaveEnemyCount)
        {
            if (enemySpawnerComponent.enemySpawnTimer <= 0)
            {
                int spawnAmount =
                UnityEngine.Random.Range(1, math.max(1, enemySpawnerComponent.currentWaveEnemyCount - enemySpawnerComponent.currentWaveEnemiesSpawned));

                SpawnEnemies(ref enemySpawnerComponent, spawnAmount);

                enemySpawnerComponent.enemySpawnTimer = UnityEngine.Random.Range(3, 10);

            }
            else
            {
                enemySpawnerComponent.enemySpawnTimer -= Time.deltaTime;
            }
        }
    }
    else
    {
        if (enemySpawnerComponent.waveIntervalTimer <= 0)
        {
            InitNewWave(ref enemySpawnerComponent);
        }
        else
        {
            enemySpawnerComponent.waveIntervalTimer -= Time.deltaTime;
        }
    }
});
```

KUVA 5. EnemySpawnerSystem-systeemin OnUpdate-metodi.

SpawnEnemies-metodi luo annetun määrän vihollisentiteettejä, lisää vihollisentiteeteille tarvittavat komponentit ja asettaa komponenttien arvot (kuva 6). Aluksi pelissä on ainoastaan yhdenlaisia vihollisia, mutta pelin edetessä vihollistyyppejä on useampia. Kun vihollinen luodaan, sille arvotaan vihollistyyppi SpawnEnemies-metodin alussa (kuva 7).

```
for (int i = 0; i < count; i++)
{

    var enemyData = EnemyHelper.GetConfig(RandomizeEnemyType());
    var enemyAnimationData = AnimationHelper.GetConfigByEnemyType(enemyData.enemyType);

    enemyEntity = EntityManager.CreateEntity(enemyArchetype);
    EntityManager.SetComponentData(enemyEntity, new EnemyComponent { attackRange = enemyData.attackRange, attackType = enemyData.enemyAttackType });
    EntityManager.SetComponentData(enemyEntity, new Translation { Value = GetSpawnPosition() });
    EntityManager.SetComponentData(enemyEntity, new MoveComponent { moveSpeed = enemyData.movementSpeed });
    EntityManager.SetComponentData(enemyEntity, new HealthComponent { maxHealth = enemyData.health, health = enemyData.health });
    EntityManager.SetComponentData(enemyEntity, new DamageComponent { damage = enemyData.damage, cooldown = enemyData.attackCooldown });
    EntityManager.SetComponentData(enemyEntity, new TakeDamageComponent { leaveCorpse = true });
    EntityManager.SetComponentData(enemyEntity, new PathFollow { pathIndex = -1 });
    entityManager.SetComponentData(enemyEntity, new SpriteComponent
    {
        currentAnimation = AnimationConfig.AnimationType.Idle,
        spriteType = AnimationConfig.SpriteType.Floater,
        frameCount = enemyAnimationData.idleAnimationFrameCount,
        frameTimerMax = enemyAnimationData.idleAnimationFrameTime,
        scale = enemyAnimationData.scale,
        rotation = Quaternion.identity,
        LoopingAnimation = true,
        shouldAnimate = true,
        color = enemyAnimationData.color
    });

    if(enemyData.enemyAttackType == EnemyConfig.EnemyAttackType.Ranged)
    {
        entityManager.AddComponent<EnemyShootComponent>(enemyEntity);
        EntityManager.SetComponentData(enemyEntity, new EnemyShootComponent {
        gunType = enemyData.gunType,
        audioType = enemyData.audioType,
        projectileCount = enemyData.projectileCount,
        projectileLifeTime = enemyData.projectileLifeTime,
        projectileSpeed = enemyData.projectileSpeed});
    }

    enemyList.Add(enemyEntity);
```

KUVA 6. Entiteettien luominen SpawnEnemies-metodissa.

```
private EnemyConfig.EnemyType RandomizeEnemyType()
{
    int typeCount = System.Enum.GetValues(typeof(EnemyConfig.EnemyType)).Length;
    typeCount = math.min(typeCount, wave / 2);
    return (EnemyConfig.EnemyType)math.min(UnityEngine.Random.Range(0, typeCount - 1), UnityEngine.Random.Range(0, typeCount - 1));
}
```

KUVA 7. Vihollistyypin arpominen RandomizeEnemyType-metodissa.

6.2.1 Vihollisten tekoäly

Pelin tekoäly tehtiin hyödyntäen rajallista tilakonetta. Pelin vihollisilla on kaksi eri tilaa, jossa ne voivat olla. Seuraamistilassa vihollinen hakeutuu tarpeeksi lähelle pelaajaa, jotta se voi aloittaa hyökkäyksen. Päästyään tarpeeksi lähelle, vihollinen vaihtaa hyökkäystilaan, jolloin se yrittää vahingoittaa pelaajaa hyökkäyksillään. Jos pelaaja liikkuu pois vihollisen hyökkäysetäisyydeltä, vaihtaa vihollinen takaisin seuraamistilaan. Rajallinen tilakone ja ECS ovat monin tavoin samankaltaisia. Niissä kooditoteutus pyritään tekemään erillisiin osiin, jolloin yhden skriptin koko ei kasva hallitsemattomasti.

6.2.2 Seuraamistila

Pelissä vihollisten liikkumisesta vastaa FollowStateSystem-systeemi, joka etsii kaikki viholliset, joilta löytyy tietyt komponentit. Tärkeimpänä FollowStateComponent-komponentti (kuva 8). Jos hakukriteerejä vastaava entiteetti löytyy, sille tehdään ensin tarkistus, onko se tarpeeksi lähellä pelaajaa aloittaakseen hyökkäämisen. Tämä tapahtuu TargetReachedCheck-metodissa (kuva 9), jossa tarkistetaan, onko pelaaja vihollisentiteetin hyökkäysetäisyydellä.

```
Entities.ForEach((
    Entity entity,
    ref EnemyComponent enemyComponent,
    ref Translation translation,
    ref MoveComponent moveComponent,
    ref TargetingComponent targetingComponent,
    ref FollowStateComponent followStateComponent) =>
{
```

KUVA 8. vihollisentiteettien etsiminen, joilta löytyy FollowStateComponent-komponentti.

```
private bool TargetReachedCheck(Translation translation, TargetingComponent targetingComponent, EnemyComponent enemyComponent)
{
    bool result = false;
    if(math.distance(targetingComponent.target.Value, translation.Value) < enemyComponent.attackRange)
    {
        result = true;
    }

    return result;
}
```

KUVA 9. TargetReachedCheck-metodi.

Vihollisentiteetin ollessa liian kaukana pelaajasta aloittaakseen hyökkäämisen, varmistetaan että entiteetillä on reitti pelaajan luokse. Mikäli reittiä ei ole tai reitti on pelaajan vanhaan sijaintiin, lähetetään reitinhaulle pyyntö uudesta reitistä (kuva 10). Vihollisentiteetit saavat reitit listoissa, jotka sisältävät etäisyydestä riippuen useampia etappeja. Vihollisentiteetit liikkuvat etappeja pitkin pelaajaa päin. Mikäli vihollisentiteetti on hyökkäysetäisyydellä pelaajasta, poistetaan siltä FollowStateComponent-komponentti. Tämän jälkeen sille lisätään joko MeleeAttackStateComponent-komponentti tai RangedAttackStateComponent-komponentti, riippuen siitä onko vihollisentiteetin hyökkäystyyppi lähitaistelu vai ampuminen.

```
DynamicBuffer<PathPosition> waypoints = EntityManager.GetBuffer<PathPosition>(entity);
PathFollow pathFollow = EntityManager.GetComponentData<PathFollow>(entity);

if(!targetingComponent.target.Value.Equals(targetingComponent.targetLastPosition) || waypoints.Length <= 0)
{
    targetingComponent.targetLastPosition = targetingComponent.target.Value;

    if(!EntityManager.HasComponent<PathfindingParams>(entity))
    {
        PostUpdateCommands.AddComponent(entity, new PathfindingParams
        {
            startPosition = new int2((int)Mathf.Clamp(Mathf.Round(translation.Value.x),0,MapBounds.width), (int)Mathf.Clamp(Mathf.Round(translation.Value.y),0,MapBounds.height)),
            endPosition = new int2((int)targetingComponent.target.Value.x, (int)targetingComponent.target.Value.y)
        });
    }
```

KUVA 10. Vihollisentiteetin reitin varmistaminen FollowStateSystem-systeemissä.

6.2.3 Hyökkäystila

Hyökkäystilaa ohjaa MeleeAttackStateSystem-systeemi ja RangedAttackStateSystem-systeemi. Niiden toteutus on hyvin samankaltainen. Suurimpana erona on, että RangedAttackStateSystem-systeemissä luodaan hyökkäyslogiikan päätteeksi yksi tai useampi ammus. Systeemien alussa tarkastetaan pelaajan etäisyys vihollisentiteetistä. Mikäli vihollisentiteetti on tarpeeksi lähellä pelaajaa hyökätäkseen, aloitetaan hyökkääminen (kuva 11). Vihollistyyppien hyökkäyksillä on eripituiset aikavälit.

```
if(damageComponent.currentCooldownTimer <= 0 && targetingComponent.hasTarget)
{
    Entity targetEntity = targetingComponent.targetEntity;
    int damage = damageComponent.damage;

    Entities.ForEach(( Entity playerEntity,
        ref PlayerComponent playerComponent,
        ref HealthComponent healthComponent,
        ref TakeDamageComponent takeDamageComponent,
        ref Translation playerTranslation ) =>
    {
        if(playerEntity == targetEntity && !takeDamageComponent.invulnerable)
        {
            takeDamageComponent.damageAmount += damage;
            takeDamageComponent.invulnerable = true;
        }
    });


    damageComponent.currentCooldownTimer = damageComponent.cooldown;
    var aSource = Pools.Instance.AudioSourcePool.GetPooledObject();
    aSource.gameObject.SetActive(true);
    AudioHelper.PlayOneShot(AudioConfig.AudioType.EnemyMelee, aSource);
}
else if(damageComponent.currentCooldownTimer > 0)
{
    damageComponent.currentCooldownTimer -= Time.deltaTime;
```

KUVA 11. MeleeAttackStateSystem-luokan OnUpdate-metodi.

RangeAttackStateSystem-systeemissä ei tehdä vahinkoa suoraan, vaan sen sijaan luodaan ammuksia, jotka tekevät vahinkoa osuessaan pelaajaan (kuva12). Ammukset luodaan arkkityypistä, joka määritetään OnCreate-metodissa (kuva13).

```
if(damageComponent.currentCooldownTimer <= 0 && targetingComponent.hasTarget)
{
    float3 direction = targetingComponent.target.Value - translation.Value;
    damageComponent.currentCooldownTimer = damageComponent.cooldown;

    Shoot(enemyShootComponent, translation.Value, new float2(direction.x,direction.y), damageComponent);

    var aSource = Pools.Instance.AudioSourcePool.GetPooledObject();
    aSource.gameObject.SetActive(true);
    AudioHelper.PlayOneShot(enemyShootComponent.audioType, aSource);

}
```

KUVA 12. RangedAttackStateSystem-luokan OnUpdate-metodi.

```
entityManager = World.Active.EntityManager;

bulletArchetype = entityManager.CreateArchetype
    (
    typeof(Translation),
    typeof(LocalToWorld),
    typeof(SpriteComponent),
    typeof(MoveComponent),
    typeof(EnemyBulletComponent)
    );
```

KUVA 13. RangedAttackStateSystem-luokan OnCreate-metodi.

Ammusten luominen tapahtuu Shoot-metodissa, johon annetaan parametrinä tiedot ampuvasta vihollisentiteetistä. Shoot-metodi hakee AnimationConfig-scriptableObjectista tarvittavat tiedot ammuksesta. Ennen ammuksen luomista sille asetetaan vielä oletusarvot komponentteihin (kuva 14).

```
AnimationConfig.AnimationData bulletAnimationData = AnimationHelper.GetConfig(AnimationConfig.SpriteType.EnemyBullet);

NativeArray<Entity> bulletEntities = new NativeArray<Entity>(enemyShootComponent.projectileCount, Allocator.Temp);
for(int i = 0; i < enemyShootComponent.projectileCount; i++)
{
    bulletEntities[i] = entityManager.CreateEntity(bulletArchetype);
    entityManager.SetComponentData(bulletEntities[i], new Translation { Value = spawnPoint + (new float3(aimDirection.x, aimDirection.y, 0) * 0.2f) });
    entityManager.SetComponentData(bulletEntities[i], new SpriteComponent
    {
        currentAnimation = AnimationConfig.AnimationType.Idle,
        spriteType = AnimationConfig.SpriteType.PlayerBuller,
        frameCount = bulletAnimationData.idleAnimationFrameCount,
        frameTimerMax = bulletAnimationData.idleAnimationFrameTime,
        rotation = Quaternion.identity,
        scale = bulletAnimationData.scale,
        color = bulletAnimationData.color
    });

    entityManager.SetComponentData(bulletEntities[i], new EnemyBulletComponent
    {
        lifeTime = enemyShootComponent.projectileLifeTime,
        damage = damageComponent.damage,
    });
```

KUVA 14. Ammusten luominen Shoot-metodissa.

6.3 Reitinhaku toteutus

Reitinhaku toteutettiin käyttämällä A*-algoritmia. Toteutuksen sisällä kartta muutetaan ruudukoksi, jossa jokainen ruutu on oma tietueensa. Jokainen tietue sisältää tiedon sijainnista ruudukossa, indeksin, g hinnan, f hinnan, h hinnan, pystyykö solmuun liikkumaan sekä indeksin solmusta, josta liikuttiin tämänhetkiseen solmuun (kuva 15).

```
private struct PathNode
{
    public int x;
    public int y;

    public int index;

    public int gCost;
    public int hCost;
    public int fCost;

    public bool isWalkable;

    public int cameFromNodeIndex;

    public void CalculateFCost()
    {
        fCost = gCost + hCost;
    }

    public void SetIsWalkable( bool isWalkable )
    {
        this.isWalkable = isWalkable;
    }
}
```

KUVA 15. PathNode-tietue.

Entiteetin pyytäessä uutta reittiä, sille lisätään PathfindingParams-komponentti. Komponentti sisältää tiedon entiteetin sen hetkisestä sijainnista ja määränpään. Reitinhaun alussa etsitään kaikki entiteetit, joilta löytyy PathfindingParams-komponentti. Niille tehdään reitinhaku-tehtävä ja PathfindingParams-komponentti poistetaan. Kun kaikki entiteetit on käyty läpi, tehtävät aikataulutetaan, jolloin Job System laittaa ne jonoon (kuva 16).

```
Entities.ForEach(( Entity entity, ref PathfindingParams pathfindingParams ) =>
{
    FindPathJob findPathJob = new FindPathJob
    {
        gridSize = gridSize,
        pathNodeArray = new NativeArray<PathNode>(gridSize.x * gridSize.y, Allocator.TempJob),
        startPos = pathfindingParams.startPosition,
        endPos = pathfindingParams.endPosition,
        entity = entity,
        pathFollowComponentFromEntity = GetComponentDataFromEntity<PathFollow>(),
    };
    findPathJobList.Add(findPathJob);
    jobHandleList.Add(findPathJob.Schedule());
    PostUpdateCommands.RemoveComponent<PathfindingParams>(entity);
});

JobHandle.CompleteAll(jobHandleList);
```

KUVA 16. PathfindingJob-luokan OnUpdate-metodi.

Reitinhaku tapahtuu FindPathJob-tehtävässä, joka on toteutettu käyttämällä Job Systemiä ja Burst Compileria. Koska se toimii useammassa säikeessä, siinä on useita rajoitteita. Aluksi luodaan uusi ruudukko, joka täytetään oletusarvoissa olevilla solmuilla (kuva 17).

```
for(int x = 0; x < gridSize.x; x++)
{
    for(int y = 0; y < gridSize.y; y++)
    {
        PathNode pathNode = new PathNode();
        pathNode.x = x;
        pathNode.y = y;
        pathNode.index = CalculateIndex(x, y, gridSize.x);

        pathNode.gCost = int.MaxValue;
        pathNode.hCost = CalculateDistanceCost(new int2(x, y), endPos);
        pathNode.CalculateFCost();
        pathNode.isWalkable = true;
        pathNode.cameFromNodeIndex = -1;
        pathNodeArray[pathNode.index] = pathNode;
    }
}
```

KUVA 17. Ruudukon luominen reitinhaku tehtävässä.

Tämän jälkeen käydään läpi solmuja, etsien optimaalisinta reittiä. Haku alkaa aloitus solmusta ja sen ympärillä olevat solmut tarkistetaan. Halvimmat ympärillä olevat solmut laitetaan listaan (kuva 18). Tämän jälkeen haetaan listasta pienimmän f hinnan omaava solmu ja etsitään sen ympäriltä taas halvin solmu. Tätä

jatketaan, kunnes reitti määränpäähän on löytynyt tai kaikki mahdolliset solmut on käyty läpi ja voidaan todeta, että reittiä määränpäähän ei ole.

```
int2 currentNodePosition = new int2(currentNode.x, currentNode.y);
int tentativeGCost = currentNode.gCost + CalculateDistanceCost(currentNodePosition, neighbourPos);
if(tentativeGCost < neighbourNode.gCost)
{
    neighbourNode.cameFromNodeIndex = currentNodeIndex;
    neighbourNode.gCost = tentativeGCost;
    neighbourNode.CalculateFCost();
    pathNodeArray[neighbourNodeIndex] = neighbourNode;

    if(!openList.Contains(neighbourNode.index))
    {
        openList.Add(neighbourNode.index);
    }
}
```

KUVA 18. Ympärillä olevien solmujen tarkastaminen FindPathJob-tehtävässä.

Monisäikeisyyden käytöstä tulevien rajoitteiden takia reitin antaminen entiteeteille ei onnistunut FindPathJob-tehtävässä, joten sitä varten luotiin SetBufferPathJob-tehtävä, joka toimii pääsäikeessä ja se ajetaan FindPathJob-tehtävän jälkeen. SetBufferPathJob-tehtävä saa reitin tiedot FindPathJob-tehtävältä. Tiedot annetaan entiteeteille CalculatePath-metodissa, jossa reitti käydään läpi lopusta alkuun ja kaikki solmut asetetaan dynaamiseen puskuriin (kuva 19), josta entiteetti käyttää niitä etappeina.

```
pathPositionBuffer.Add(new PathPosition { position = new int2(endNode.x, endNode.y) });

PathNode currentNode = endNode;

while(currentNode.cameFromNodeIndex != -1)
{
    PathNode cameFromNode = pathNodeArray[currentNode.cameFromNodeIndex];
    pathPositionBuffer.Add(new PathPosition { position = new int2(cameFromNode.x, cameFromNode.y) });
    currentNode = cameFromNode;
}
```

KUVA 19. CalculatePate-metodi.

7 Pohdinta

Unityn Data-oriented Technology Stack (DOTS) toimi hyvin tekoälyä tehdessä, sillä ECS koodissa ja rajallisessa tilakoneessa on paljon samankaltaisuuksia. Koodin rakenne oli selkeää ja bugit olivat helposti havaittavissa. Job Systemin ja Burst Compilerin käyttö reitinhaussa paransivat suorituskykyä huomattavasti. Varsinkin monisäikeisyys nopeutti reitinhaun ajoa moninkertaisesti. DOTS oli kuitenkin vielä hyvin keskeneräisessä vaiheessa. Siitä puuttui monia tärkeitä ominaisuuksia, joita pelinkehittämisessä tarvitaan kuten fysiikat, äänet ja kuvien animointi. DOTSin ja Unityn perinteisen järjestelmän käyttö yhdessä on myös mahdollista ja varmasti kannattavin vaihtoehto tällä hetkellä. Esimerkiksi Job Systemin ja Burst Compilerin käyttö raskaissa operaatioissa voi olla järkevä ratkaisu.

Opinnäytetyön käytännön osuuden alussa oli hieman ongelmia ymmärtää, mitenkä ECS-koodia tulisi kirjoittaa ja koodi näytti enemmänkin ECS-koodin ja olio-ohjelmoinnin yhdistelmältä. Ongelmia tuotti myös se, että DOTS on uusi paketti, joten ohjeita sen käyttämiseen on vähän ja muutoksia tulee usein. Vähäisetkin ohjeet saattoivat sisältää vanhaa informaatiota. Projektissa ei varmasti käytetty DOTSin kaikkea potentiaalia.

Job Systemin ja Burst Compilerin käyttö oli haastavaa, kun niitä käytettiin jo olemassa olevien systeemien kanssa, joka teki koodista monimutkaisempaa. Jatkotutkimuksena voisi perehtyä Job Systemin ja Burst Compilerin käyttöön pienemissä ympäristöissä, joissa niiden hyödyllisyyden pystyy havaitsemaan paremmin.

LÄHTEET

Amat, C. 2018. Unity Job System Explained. Julkaistu 7.11.2018. Luettu 2.4.2020. http://infalliblecode.com/unity-job-system/

Amat, C. 2018. Unity Burst Compiler: Performance Optimization Made Easy. Julkaistu 15.11.2018. Luettu 2.4.2020. http://infalliblecode.com/unity-burst-compiler/

Aversa, D. 2019. What is Unity's new Data-Oriented Technology Stack (DOTS) Julkaistu 4.12.2019. Luettu 19.3.2020. https://hub.packtpub.com/what-is-unitys-new-data-oriented-technology-stack-dots/

Batoćanin, V. n.d. Basic AI Concepts: A* Search Algorithm. Luettu 20.10.2020. https://stackabuse.com/basic-ai-concepts-a-search-algorithm/

Brilliant.org n.d. A* Search. Luettu 20.10.2020. https://brilliant.org/wiki/a-star-search/

Clark, D. 2011. Beginning C# Object-Oriented Programming. Julkaistu 2011. Luettu 14.10.2020. https://learning.oreilly.com/library/view/beginning-c-object-oriented/9781430235309/ch01.html#inheritance

Ediriweera, S. 2017. Polymorphism explained simply! Kirjoitettu 12.10.2017. Luettu 14.10.2020. https://medium.com/@shanikae/polymorphism-explained-simply-7294c8deeef7

Educba n.d. What is Inheritance in Programming? Luettu 14.10.2020. https://www.educba.com/what-is-inheritance-in-programming/

Gameranx 2016. How Does VIDEO GAME AI Work Youtube-video. Julkaistu 8.9.2016. Viitattu 16.10.2020. https://www.youtube.com/watch?v=cIfow-wJj_GA&ab_channel=gameranx

Janssen, M. n.d. Inheritance among Classes in Object-Oriented Programming (OOP). https://www.researchgate.net/figure/Inheritance-among-Classes-in-Object-Oriented-Programming-OOP_fig4_228839172

Janssen, T. 2017. OOP Concept for Beginners: What is Abstraction? Julkaistu 23.11.2017. Luettu 14.10.2020. https://stackify.com/oop-concept-abstraction/

Kenneth, R. 2017. Chapter3: What is Object-Oriented Programming? Julkaistu 5.6.2017. Luettu 19.3.2020. https://medium.com/learn-how-to-program/chapter-3-what-is-object-oriented-programming-d0a6ec0a7615

Petty, J. n.d. What is Unity 3D & What is it Used For. Luettu 19.3.2020. https://conceptartempire.com/what-is-unity/

Rabin, S. 2006. Introduction to Game Development, Second Edition. Kirjoitettu 1.6.2006. Luettu 16.10.2020. https://ebookcentral.proquest.com/lib/tampere/reader.action?docID=3136189

Rafiq, A. 2020. Pathfinding Algorithms in Game Development. Kirjoitettu 2020. Luettu 20.10.2020. https://iopscience.iop.org/article/10.1088/1757-899X/769/1/012021/pdf

Unity Technologies. 2020a. Scripting in Unity for experienced programmers. Luettu 19.3.2020. https://unity.com/how-to/programming-unity

Unity Technologies. 2020b. Entity Component System. Luettu 24.3.2020. https://docs.unity3d.com/Packages/com.unity.entities@0.1/manual/index.html