Opinnäytetyö (AMK)

Tieto- ja viestintätekniikka

2025

Jere Stenius

Tekoälyn taktisen päätöksenteon parantaminen Unity-pelimoottorissa

TURKU AMK
TURKU UNIVERSITY OF APPLIED SCIENCES

Opinnäytetyö (AMK) | Tiivistelmä

Turun ammattikorkeakoulu

Tieto- ja viestintätekniikka

2025 | 35 sivua

Jere Stenius

Tekoälyn taktisen päätöksenteon parantaminen

Hyvä tekoäly on pelien kannalta tärkeä ominaisuus, sillä sen avulla voidaan tuoda pelaajalle haastetta peliin, elävöittää maailmaa ja antaa pelaajalle tekemistä.

Lähes jokainen nykyaikainen peli käyttää jonkinlaista tekoälyä. Taktisella päätöksenteon parantamisella tarkoitetaan sitä, että tekoäly reagoi uhkiin, ärsykkeisiin ja maailmaan oikealla tavalla ja koittaa selviytyä tilanteista. Tällä tavoin voidaan luoda pelaajalle sopivaa haastetta peliin, parantaa pelin pelattavuutta ja säädellä vaikeustasoa oikein.

Opinnäytetyössä toteutettiin päätöksenteko kahdella eri tekniikalla. Opinnäytetyössä käytiin läpi Unity 6:n mukana tullut käyttäytymispuujärjestelmä, jota vertailtiin itse ohjelmoituun C#-ohjelmointikielellä tehtyyn tilakonejärjestelmään. Näitä toteutuksia vertailtiin projektinhallinnallisesta näkökulmasta ja suorituskyvyn paremmuudesta. Lopullisena tuloksena opinnäytetyössä saatiin kaksi tekoälyjärjestelmää, joita voitiin käyttää ensimmäisen persoonan ammuntapelissä vastustajina ja joukkutovereina. Toinen järjestelmä toteutettiin itse ohjelmoidulla tilakoneella ja toinen toteutettiin käyttäytymispuun avulla. Lopullisissa tuloksissa kävi ilmi, että käyttäytymispuu on tehokkaampi ratkaisu suorituskyvyn puolesta ja projektinhallinnallisesta näkökulmasta.

Asiasanat: tekoäly, peliala, pelisuunnittelu

Bachelor’s Degree | Abstract

Turku University of Applied Sciences

Information and Communication Technology

2025 | 35 pages

Jere Stenius

Improving the Tactical Decision-Making of Artificial Intelligence

Well made artificial intelligence is important feature in games because it can be used to challenge the player, liven up the game world, or provide the player with different activities. Nearly every modern game uses some form of artificial intelligence agents in game.

Improving tactical decision making means that the artificial intelligence agent reacts correctly to threats, stimuli, and the game world, and attempts to handle the situations correctly. A suitable level of challenge can be created for the player, enhancing the gameplay and allowing the difficulty to be adjusted properly by the game developer.

This thesis aimed to study how tactical decision making can be improved using two different artificial intelligence agent techniques. The thesis examined the behavior tree system introduced in Unity 6 and compared it to custom made state machine system created in the C# programming language. The two systems were compared from a project management perspective and the performance perspective. The final result of the thesis was two artificial intelligence agent systems that can be used as enemies or as teammates in a first person shooter game. One system was implemented with a custom made state machine, and the other was implemented using Unity’s behavior tree system. The final results indicated that it was more straightforward to use behavior trees than state machines for the game development projects because it was easier to manage projects using them and the performance was better.

Keywords: Artificial Intelligence, Game Development, Game Design

Sisältö

Kuvat

Käytetyt lyhenteet tai sanasto

Agentti	tekoälyn ohjailema hahmo pelissä.
Kohtaus	Unityn käyttämä tila jossa voidaan muokata pelimaailmaa.
Käyttätymispuu	Puuta muistuttava rakenne, joka koostuu useammista solmuista
Liitutaulu	Käyttäytymispuun muuttujat talllennetaan liitutauluun Unityssä.
Solmu	Käyttäytymispuun kohta, jossa määritellään toiminta tila.
Unity	Opinnäytetyössä käytettävä sisällöntuottoalusta

1 Johdanto

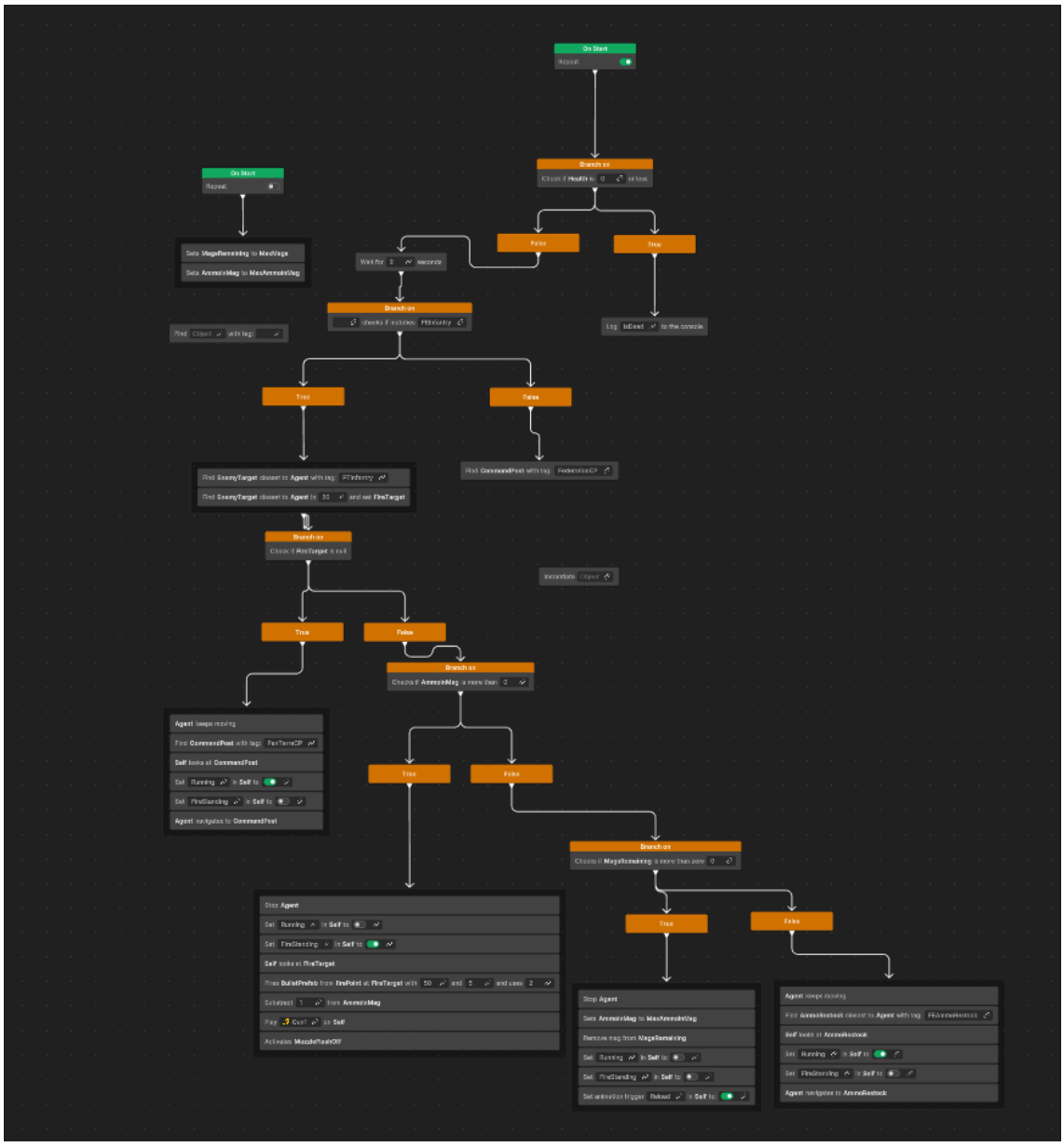
Hyvin toimivat tekoälyagentit ovat pelien yksi tärkeimmistä osista. Tekoälyagenttien avulla pelaajalle voidaan antaa urheilupeleissä joukkuekavereita ja kampanjapohjaisissa peleissä agentit antavat vastusta pelaajalle ja elävöittävät pelimaailmaa. Siksi on tärkeää, että tekoäly on hyvä tehtävässään. (Klosowski, 2024) Tekoälyn on reagoitava uhkiin ensimmäisen persoonan ammuntapelissä oikeilla tavoilla, kuten esimerkiksi heittämällä kranaatin tai hyökkäämällä pelaajan kimppuun muilla tavoin. Pelitekoälyn avulla pelin vaikeustaso voidaan tasapainoittaa juuri siten miten pelisuunnittelija haluaa ja tätä kautta pelistä voidaan saada mielenkiintoinen ja haastava pelaajalle. (Sabbaqh, 2015)

Tekoälyagenttien toteutukseen on useita erilaisia tapoja. Opinnäytetyössä tutustumaan tarkemmin tilakoneeseen (engl. Finite State Machine) ja käyttäytymispuuhun (engl. Behavior tree). Tilakone on yksi vanhimmista tavoista tehdä vastustajan tekoäly videopeliin ja sitä on käytetty esimerkiksi Pacmanissa (Jagdale, 2021, 384). Käyttäytymispuu on taas skaalautuvampi ratkaisu ja löytyy uudemmistakin peleistä, kuten Halo 2. (Sekhavat, 2017, 1)

Tässä opinnäytetyössä vertaillaan Unityn omaa käyttäytymispuuratkaisun avulla toteutettua ja itse ohjelmoitua tilakonetta keskenään. Käyttäen Unityn statistiikkatyökalua niistä vertaillaan suorituskykyä, miten helppo ratkaisu on sisällyttää projektiin, miten projektinhallinta onnistuu ja miten näillä voidaan parantaa tekoälyn taktista päättelykykyä. Lopputuloksena on kaksi erilaista tekoälyagenttitoteutusta, joita voidaan käyttää ensimmäisen persoonan ammuntapelissä Unity-pelimoottorissa. Vaikka opinnäytetyö onkin toteututtu Unity-pelimoottorissa se voidaan myös toteuttaa käyttäen Unreal Enginen käyttäytymispuuta tai blueprint järjestelmää.

2 Tekoäly Unity-pelimoottorissa

Unityssa ei pitkään ollut omaa järjestelmää tekoälyn luontiin vaan käyttäjän piti alusta saakka tehdä kaikki itse. Ratkaisuja pystyi kyllä ostamaan, mutta varsinaista virallista järjestelmää ei ollut. Nyt Unityn versiossa 6 on käyttäytymispuu malliin perustuva järjestelmä, jossa voidaan luoda tekoälyagentteja graafisin työkaluin kertomalla tarinaa, jota pelimoottori pystyy lukemaan vasemmalta oikealle ja ylhäältä alas. Kokonainen käyttäytymispuu muistuttaa puuta rakenteeltaan kuten kuvassa 1 on esitelty.



Kuva 1 Unityn käyttäytymispuu.

Tätä aikaisemmin tekoälyn luontiin on täytynyt aina ohjelmoida itse ja yhtenä vanhimpana tapana on ollut tuottaa se käyttäen tilakonetta (Jagdale, 2021, 384).

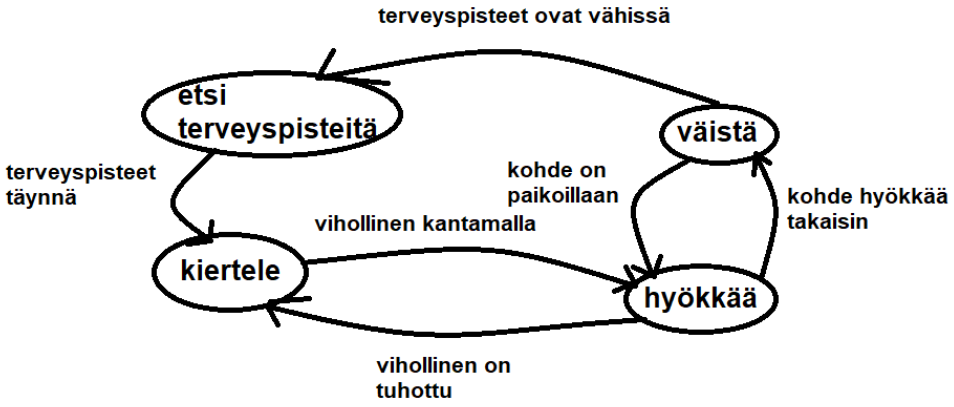
2.1 Tilakone Unity -pelimoottorissa

Unityssä ei ole valmista pelimoottoriin rakennettua ratkaisua tilakonetta varten. Joten tilakone on siten aina itse käyttäjän ohjelmoitava C#-ohjelmointikielelllä tai sitten käyttäjän on ostettava maksullinen versio Unityn lisäosakaupasta.

Tilakoneessa on useampia ohjelmoijan ennalta määrittelemiä tiloja, joiden välillä tilakone vaihtelee riippuen ympäröivän maailman ärsykkeistä.

Tiloja voi olla esimerkiksi partiointi, jossa agentti menee ennalta määrättyjen tai satunnaisten etappien välillä, hyökkäys pelaajaa tai muita agentteja kohtaan, aseen lataaminen, kranaatin heitto tai vaihtoehtoisesti terveyspisteiden palautus etsimällä esineitä tai automaatteja, jotka palauttavat agentille terveyspisteitä.

Ärsykkeinä tilakoneen tilojen muutokseen voi olla esimerkiksi pelaajan etäisyys ja näkyminen, joka vaihtaa tilan hyökkäykseen partiotilasta. Jos pelaaja on päässyt yllättämään agentin, niin terveyspisteiden palautus, mikäli terveyspisteet laskevat liian alas. Eri tilojen välillä tapahtuvat muutokset voivat tapahtua esimerkiksi kuvan 2 mukaisesti.



Kuva 2 Tilakone.

Tilakoneessa voi olla vain yksi tila kerrallaan käytössä. Tämä tarkoittaa sitä ettei agentti voi samaan aikaan ladata asetta ja heittää käsikranaattia vihollista kohti.

Tiloihin on myös määritelty etukäteen, minkä tilakoneen tilojen välillä voi siirtyä. Kaikkiin tiloihin ei voi siirtyä jokaisesta tilasta. Esimerkkinä tilasta, johon voi siirtyä mistä tahansa tilasta, on agentin kuolematila. Mikäli terveyspisteet laskevat nollaan tai alle nollan, agentti käynnistää kuolematilan, jossa määritetään agentin tuhoutumiseen liittyvät asiat.

Tilakoneen tekeminen voi vaatia paljon työtä tilakoneen ohjelmoijalta. Tiloja tilakoneeseen saattaa syntyä useita kymmeniä, ja niiden lisääminen ja korjailu vaativat aikaa ja työtä. Unityssä ei ole omaa ratkaisua tilakoneisiin, mutta niitä löytyy Unityn lisäosakaupasta. Niiden käyttäminen voi olla hyödyllistä, koska isommissa projekteissa ne antavat valmiin ja kustannustehokkaan ratkaisun projektiin. Itse toteutetussa tilakoneessa hyödyt ovat muokattavuus, erityistarpeet sekä se, että silloin käyttäjä saa juuri selllaisen ratkaisun kuin haluaakin. Projektin kasvaessa voi olla hankalaa suorittaa virheenjäljitystä ja uusien tilojen liittämistä toisiinsa.

Tilakone on yksi vanhimmista menetelmistä tuottaa tekoälyä peliin, ja sen lähtökohta on jo ensimmäisessä Pacman pelissä (Jagdale, 2021, 384).

Uudemmissa peleissä sitä on käytetty muunmuassa Half-life 2. Kuvassa 3 esitellään koodin avulla esimerkki tilakoneen tilanhallitsijasta.

Assets > NPC > C# StateManager.cs

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4
5   public class StateManager : MonoBehaviour
6   {
7       public State currentState;
8       // Update is called once per frame
9       void Update()
10      {
11          RunStateMachine();
12      }
13      private void RunStateMachine()
14      {
15          State nextState = currentState?.RunCurrentState();
16          if(nextState != null)
17          {
18              SwitchToNextState(nextState);
19          }
20      }
21
22      private void SwitchToNextState(State nextState)
23      {
24          currentState = nextState;
25      }
26  }
```

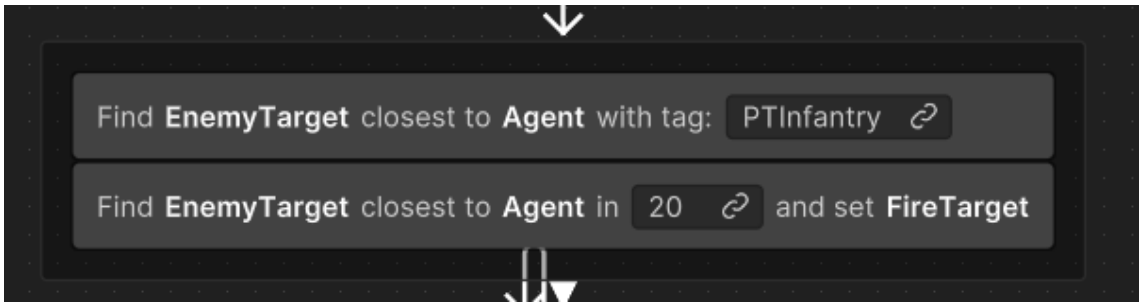
Kuva 3 Tilakoneen koodi esimerkki.

2.2 Käyttäytymispuu Unity -pelimoottorissa

Käyttäytymispuut ovat graafisia työkaluja, joilla voidaan luoda pelimoottorille tarinaa, jonka pelimoottori lukee ylhäältä alas ja vasemmalta oikealle. Käyttäytymispuut ovat hierarkisia rakenteeltaan ja niillä voidaan luoda tekoäly agenteille modulaarisia tehtäviä, joiden välillä agentti pystyy vaihtelemaan tehtävää. Rakenne mahdollistaa kompleksisten ja mukautuvien tekoäly järjestelmien rakentamisen visuaalisesti. (Unity3D. Päiväämätön. A)

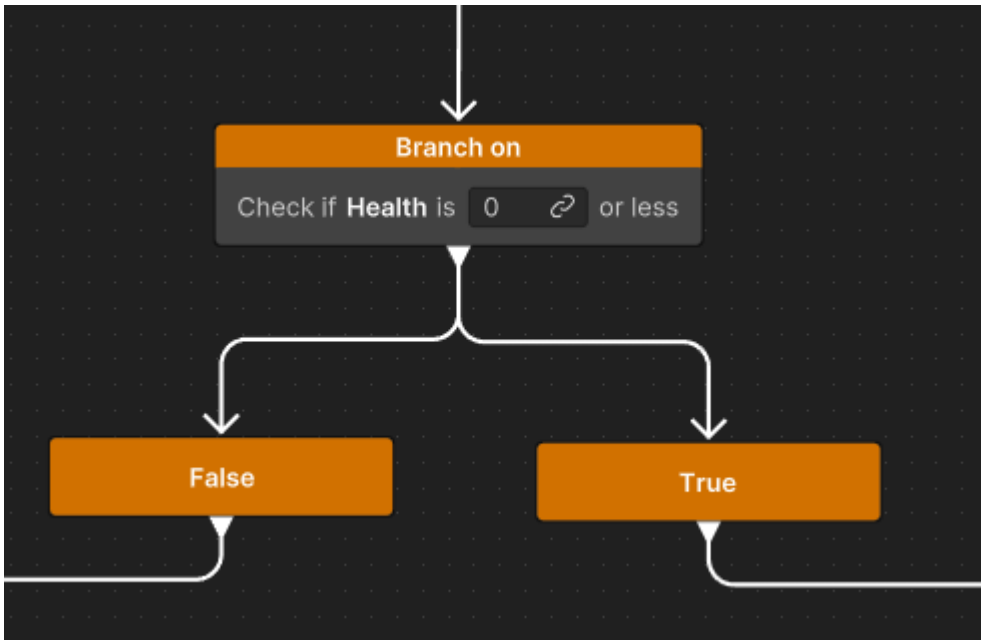
Rakenne koostuu käyttätymispuussa oksista ja oksat taas koostuvat erilaisista solmuista. Solmuilla on erilaisia tehtäviä.

Toimintasolmut (action node) ovat tyypillisimpiä solmuja, joiden avulla agenttia voidaan ohjailla tekemään erilaisia asioita esimerkiksi liikkumaan paikasta a paikkaan b, käyttämään animaatiota, muokata agenttiin liittyviä muuttujia tai muokata Unityn kohtausta. Kuvassa 4 esitellään toimintasolmu.



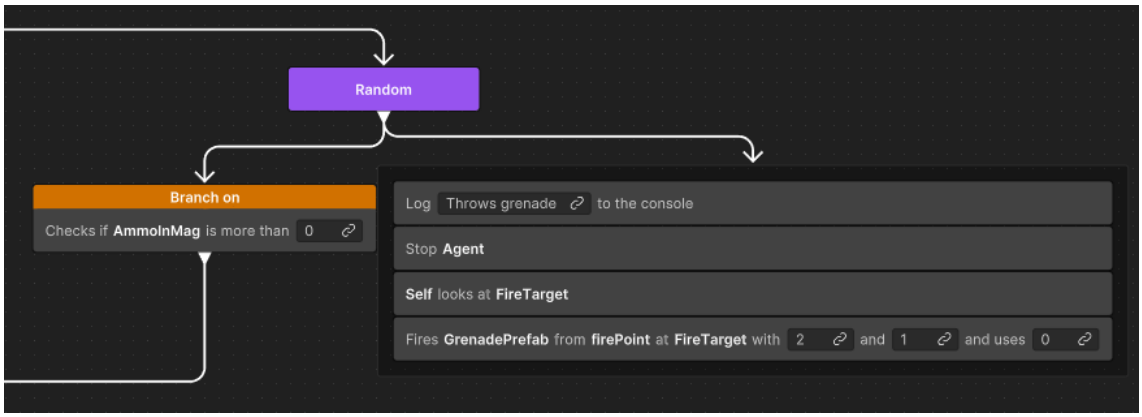
Kuva 4 Toimintasolmusta esimerkki.

Muunninsolmu (modifier node) on solmu, jota voidaan käyttää käyttäytymispuun virtauksen muokkaamiseen. Kuvassa 5 esimerkki muunninsolmusta.



Kuva 5 Muunnin solmusta esimerkki.

Jaksotussolmu (sequencing node) kertoo, miten oksa ajetaan eli lukeeko pelimoottori sen samanaikaisesti toisen oksan kanssa vai ottaako esimerkiksi satunnaisen oksan ja suorittaa sen. Kuvassa 6 näkyvä sattumanvarainen (random) solmu on esimerkki jaksotussolmusta.



Kuva 6 Jaksotussolmusta esimerkki.

Liittymissolmu (join node) mahdollistaa eri oksien yhdistämisen.

Unity käyttää käyttäymispuista termiä behavior graph eli käyttäytymisgraafi. Toisin kuin normaalit käyttäytymispuut, Unityn käyttämä käyttäytymisgraafi mahdollistaa epälineaarisen struktuurin, jossa oksat voivat kiinnittyä yhteen uudelleen ja niiden avulla voidaan luoda kompleksempia käyttätymismalleja. (Unity3D. Päiväämätön. B)

Käyttätymispuita käyttävät pelit ovat selkeästi uudempia kuin tilakonetta käyttävät. Esimerkiksi Halo 2 käyttää käyttäytymispuita ja tuottaa niillä monimutkaista pelitekoälyä.

3 Käytännön toteutus

Opinnäytetyön työprosessin tarkoituksena oli toteuttaa kahdella eri tavalla tekoälyagentit ja vertailla niitä Unity-pelimoottorissa.Tekniikoiksi valittiin Unityn oma käyttäytymispuu ja itse C# -ohjelmointikielellä ohjelmoitu tilakone. Näitä eri tekniikoita vertailtiin projektinhallinnan ja suorituskyvyn suhteen.

3.1 Suunnittelu

Työprosessin suunnittelu aloitettiin pohtimalla erilaisia kysymyksiä agenttien toimintaan. Kysymyksillä yritettiin saada selkeyttä siihen, miten agentit toimivat ympäristössä, miten ne reagoivat ärsykkeisiin ja millaisia solmukohtia tai millaisia tiloja tekoälyagentti vaatii. Mitä agentti tekee syntyessään pelimaailmaan? Miten agentti toimii kohdatessaan pelaajan tai vastustajajoukkueen tekoälyagentin? Mitä tapahtuu jos agentin elämäpisteet vähenevät liikaa ja mitä tapahtuu kun agentit elämäpisteet loppuvat kokonaan ja agentti kuolee? Miten parantaa agentin selviytymiskykyä tulitaistelutilanteissa?

Opinnäytetyössä näiden kysymysten pohjalta on luotu Unityn omaa järjestelmää käyttävä käyttäytymispuu ja omaa C#-pohjaista ratkaisua käyttävä tilakone.

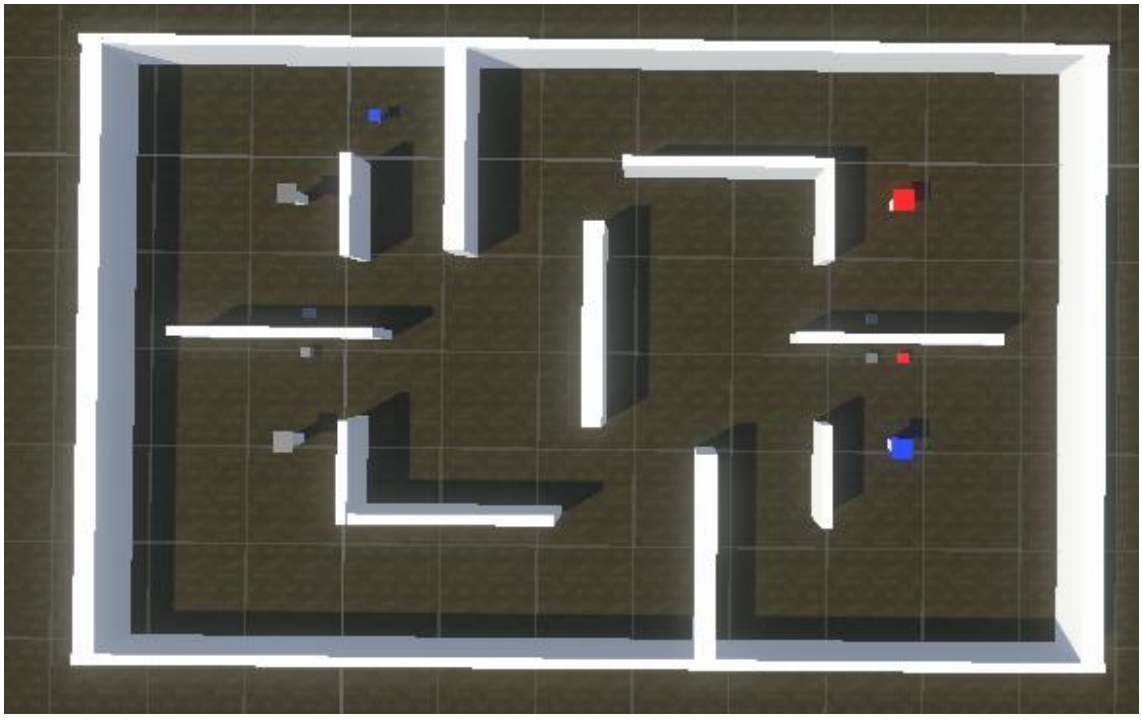
Agenteille myös suunniteltiin ja rakennettiin ympäristö, jossa pystyttiin testaamaan agenttien toimintaa, selviytymiskykyä, suorituskykyä ja toiminnallisuutta. Ympäristöön voidaan synnyttää eri määrä eri joukkueiden agentteja taistelemaan keskenään.

Suunnitteluun kuuluu myös selvittää, miten ja miksi toinen järjestelmä on parempi kuin toinen ja millä tavalla nämä järjestelemät ovat toisiaan parempia. Järjestelmissä tarkastellaan, miten helppoa ja tehokasta projektinhallinta ja projektien ylläpito on. Vaikuttavia tekijöitä ovat esimerkiksi järjestelmien luettavuus ja selkeys, ja kuinka aikaa vievää muokkausten, korjausten ja lisäysten tekeminen on. Työssä myös vertaillaan suorituskykyä kummastakin järjestelmästä, synnyttämällä pelimaailmaan useita kopioita agenteista ja

laittamalla agentit taistelemaan keskenään ympäristöön. Järjestelmäresurssien monitorointi tehdään Unityn omalla statistiikkajärjestelmällä ja seuraamalla sen antamia tuloksia kuvan päivitysnopeudesta.

3.2 Toteutus Unity -pelimoottoriin

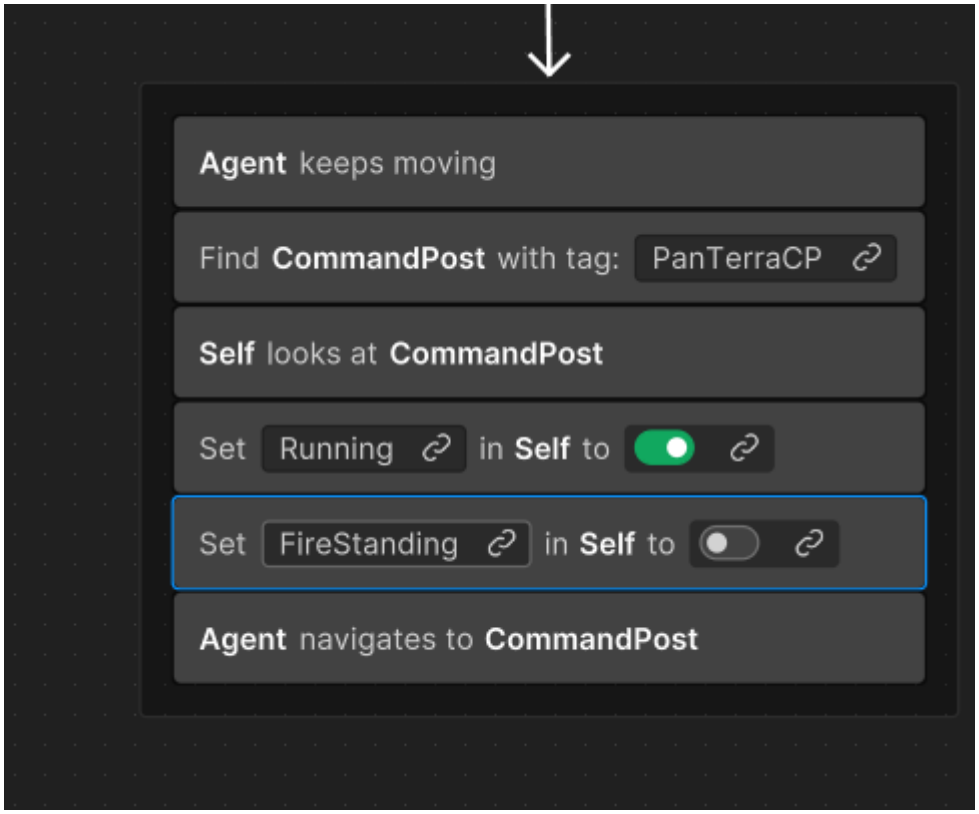
Opinnäytetyön valmistaminen jaktui kysymysten jälkeen selvittämällä, miten nämä asiat saadaan sisällytettyä Unity -pelimoottoriin. Alussa tutkittiin eri blogeista, kirjoituksista ja foorumi viesteistä, mikä on paras ja optimaalisin tapa kirjoittaa oma C#-pohjainen tilakone Unityyn. Ennen tilakoneen ohjelmoimista tarvittiin tekoälyagenteille testiympäristö, jossa agentit voisivat taistella ja liikkua paikasta toiseen.Tätä varten mallinnettiin ympäristö, jossa tekoälyagenttien toimivuutta ja toimintaa voitiin testata. Ympäristössä on vastapäisissä kohdissa komentopisteet, joita toinen joukkue yrittää vallata. Kuvassa 7 esitellään miltä ympäristö näyttää.



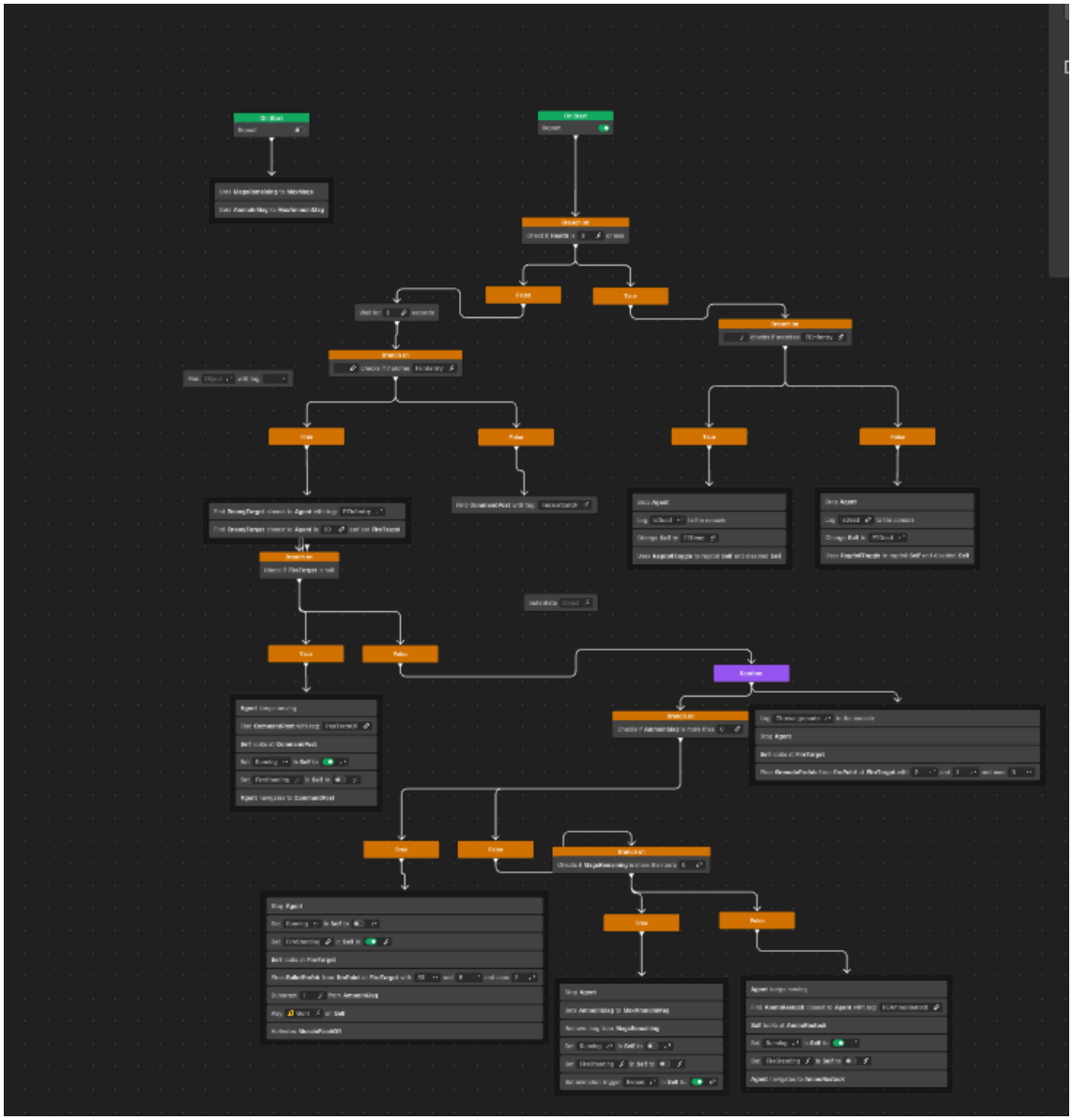
Kuva 7 Testiympäristö tekoälylle.

Tekoälyn tilakoneeseen sisällyttetiin ensimmäisenä tila, jossa agentit koittavat vallata vastustaja joukkueen komentopisteitä ja valloitettuaan pisteen agentit siirtyvät seuraavaan pisteeseen. Tällä tavoin kummankin osapuolen agentit saadaan kohtaamaan kartalla. Seuraavaksi agenttien täytyi reagoida viholliskosketukseen ja toimia oikein. Agenteille sisällytettiin ampumistila, jossa agentit voivat hyökätä vihollisten ja pelaajan kimppuun. Tekoälylle annettiin myös kyky ladata ase ja poistua taistelusta mikäli tekoälyn elämäpisteet vähenivät liikaa. Agenteille annettiin myös kyky etsiä vihollisia, jotka tekoäly oli havainnut, mutta kadottanut. Agenteille annettiin myös kuolemistila, jossa tekoälyltä laitetaan kaikki ylimääräiset järjestelmät pois ja valmistellaan agentti syntymään uudelleen karttaan ja aloittamaan hyökkäyksen vastustaja joukkueen komentopisteisiin uudelleen. Agentit käytännössä toistavat tätä samaa, kunnes toisen joukkueen kaikki komentopisteet on vallattu tai heräämispisteet loppuvat tai pelaaja aloittaa pelin alusta uudelleen.

Käyttäytymispuuna käytettiin Unity 6 mukana tullutta omaa ilmaista ja graafista ratkaisua. Käyttäytymispuuhun on valmiita solmuja, joiden avulla käyttäjä voi rakentaa tekoälyä. Käyttäytymispuuhun kuitenkin otettiin samat kysymykset kuin tilakoneeseen ja mietittiin niiden kautta, miten saadaan käännettyä tilakone käyttäytymispuuksi. Alkuperäiset solmukohdat eivät riittäneet tarvittavan tekoälyn luontiin, joten niitä piti luoda lisää käyttäen C#-ohjelmointikieltä. C#:lla saadaan luotua uusia solmuja, jotka ovat käytännössä pieniä osia tilakoneen tiloista, joita pelimoottori lukee ylhäältä alas ja vasemmalta oikealle. Uusia solmukohtia voidaan lisätä tilakoneeseen niin kuin mitä tahansa muita solmukohtia ja niitä voidaan käyttää myös useamman kerran eri oksilla. Agentille annettiin samat tilat kuin tilakoneessa, mutta ne muokattiin erillisiksi solmukohdiksi ja lisättiin käyttäytymispuuhun. Kuvassa 8 esitellään, miten eri tilat on muokattu erillisiksi solmukohdiksi. Kuvassa 9 taas esitellään, miltä käyttäytymispuu näyttää kokonaisuudessaan.



Kuva 8 Käyttäytymispuun solmukohta.



Kuva 9 Käyttäytymispuu kokonaisuudessaan.

4 Vertailu

4.1 Projektinhallinnallinen toimivuus

Opinnäytetyössä vertailtiin tilakonetta ja käyttäytymispuuta keskenään ja näiden hyviä ja huonoja puolia projektinhallinnallisesta näkökulmasta. Arviointikriteereihin kuului, kuinka helppoa projektia on muokata ja lisätä uusia tiloja agenteille, kuinka siistinä projekti pysyy ja kuinka helposti koodia voidaan käyttää uudelleen eri agenttien välillä.

Tilakoneeseen on nopeaa rakentaa uusia tiloja ohjelmoimalla C#:lla ja suunnittelemalla itse mitä haluaa. Tilakoneesta saa juuri sellaisen ratkaisun kuin käyttäjä haluaa ja tarvitsee pelitekoälyn toimintaan.

Tilakoneessa on tarkkaan suunniteltava, mikä tila johtaa mihinkin tilaan ja mitä tilat tekevät. Tiloja lisätessä tarvitsee aina manuaalisesti lisätä siirtymät uusiin tiloihin ja kirjoittaa ärsykkeet, jotka johtavat uusien tilojen siirtymiin. Tiloja saattaa syntyä useita kymmeniä erilaisia ja jokainen vaatii tilojen siirtymiset. Tämä johtaa lopulta siihen, että koodia on vaikea ylläpitää suuremmissa ja monimutkaisemmissa tekoäly agenteissa. Tässä opinnäytetyössä tiloja oli kuusi ja jo kuuden tilan kanssa projekti alkoi olemaan työläs muokata ja lisätä tiloja. Opinnäytetyössä oli myös ongelmia jäsennellä koodia tehokkaasti ja se teki tiloista entistäkin sekavempia. Ongelmaksi myös muodostuu se, että kaikki tilat ovat erillisia C#-tiedostoja mikä tekee tilojen pikaisesta tarkastestelusta hankalaa ja työlästä. Tilat on aina avattava koodieditorissa tarkastelua varten joten se vie aikaa jo itsessään. Tiloissa on myös vaikea nähdä tästä syystä mitkä tilat menevät mihinkä tilaan ja se vaatii aina tiedostojen erillistä avaamista tarkastelua varten.

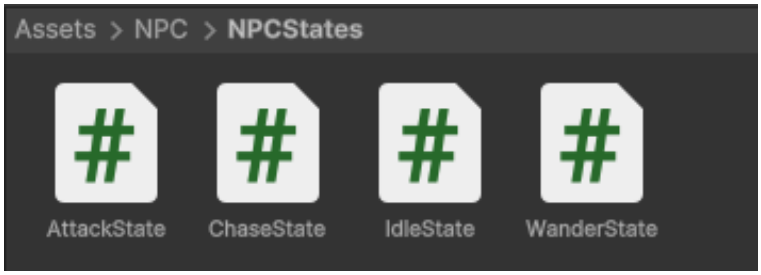
Opinnäytetyössä agenttien toimivuudelle oli vaatimuksena tilakoneelle erillinen päätiedosto joka toimii agentin aivoina. Kaikki muuttujat on tallennettava erikseen agentin päätiedostoon, jossa käsitellään tekoälyn elämänpisteitä, aseen ominaisuuksia kuten etäisyyttä ja sitä, montako laukausta sekunnissa

voidaan ampua, tekoäly agentin kohteita ja joukkuekohtaisia tietoja. Muuttujat löytyvät kyllä yhdestä paikasta, mutta vaativat jälleen C#-tiedoston avaamista, eivätkä ole näkyvissä, kun agentin eri tiloja muokataan. Kuvassa 10 on esimerkki päätiedostoon kuuluvuista muuttujista.

```
public class AIInfantry : MonoBehaviour, IDamageable {
    public NavMeshAgent navMeshAgent;
    public Animator animator;
    public int health = 5;
    public int originalHealth = 100;
    public GameObject blueCP;
    public GameObject redCP;
    public GameObject[] attackableCPs;
    public GameObject[] targets;
    public GameObject closestTarget;
    public InfantryWeaponAI infantryWeaponAI;
    public AISpawnManager spawn;
    public RagdollToggle ragdollToggle;
    public float weaponRange = 20;
    public bool inWeaponRange;
    public LayerMask whatIsEnemy, obstacleMask;
    public float rotSpeed = 5;
    public float fireRate;
    public bool alive = true;
    public Teams teams;
    public GameObject miniMapIndicator;
    public EnemyVision enemyVision;
    public SeekTarget seekTarget;
    public AttackCP attackCP;
    public StateManager stateManager;
    public bool firstSpawn = true;
    public LayerMask layerMask;
    public GameObject targetingLevel;
    public bool canSeeTarget = false;
    public int index;
```

Kuva 10 Tilakoneen päätiedoston muuttujat.

Useampien tiedostojen takia projekti alkaa olemaan sekava ja vaikeasti luettava. Projektin ylläpitoa helpottaa kyllä se, että nimeää tiedostot hyvin ja yksiselitteisesti. Kuvassa 11 esitellään tilakoneen tiloja, jotka on nimetty yksiselitteisesti ja helposti luettavaan muotoon.

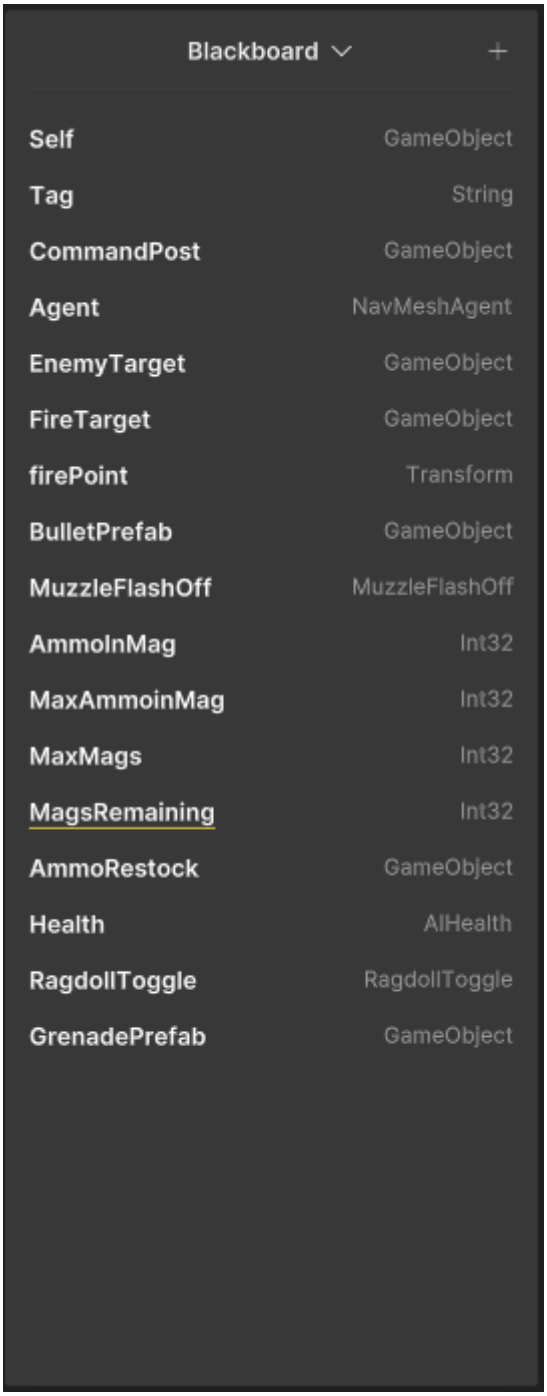


Kuva 11 Tilakoneen tilat.

Tilojen käyttö uudeelleen oli tässä projektissa helppoa. Tiloihin tarvitsi vain lisätä erilliset tagitunnisteet joiden avulla agentit pystyivät tunnistamaan vastustaja joukkueen joukot omistaan ja reagoimaan oikealla tavalla ärsykkeisiin. Testipelissä kumpikin osapuoli käyttää samoja tiloja ja se helpottaa ja nopeuttaa ohjelmointia paljon.

Käyttätymispuu on graafinen ja uudempi järjestelmä, jota on käytetty esimerkiksi Halo 2:n kehittämisessä. Graafisuudesta johtuen jo pelkästään eri tilojen tarkastelu solmuina on huomattavasti helpompaa ja tehokkaampaa kuin tilakoneessa, jossa tarvitsee avata erillisiä C#-tiedostoja. Käyttätymispuussa voi nähdä koko tekoälyn rakenteen kokonaisuudessaan ja käyttäytymispuuhun pystyy myös suurentamaan lähempää tarkastelua varten. Graafisessa järjestelmässä on selkeät oksat, joissa eri solmukohdat yhdistyvät viivojen avulla toisiinsa. Rakenteen ja ohjelman ajosuunnan selvittäminen on huomattavasti helpompaa kuin tilakoneessa.

Kaikki muuttujat ovat helppoja löytää käyttätymispuun liitutautaulusta ja niiden muokkaaminen on helppoa ja intuitiivista. Kuvassa 12 on esimerkki Unityn liitutaulusta, ja sen muuttujista.



Kuva 12 Käyttäytymispuun muuttujat.

Opinnäytetyön projektiin ei tarvittu erillistä päätiedostoa, johon kaikki muuttujat olisivat tallennettuna. Graafisen käyttöliittymän ansiosta uusien tilojen lisääminen, tilojen poistaminen, vanhojen tilojen tarkastelu ja ylipäänsä suoritusjärjestyksen tarkastelu on helppoa ja tehokasta verrattuna

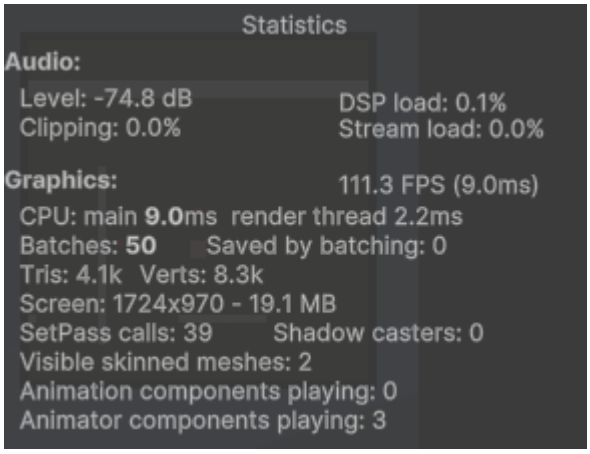
tilakoneeseen. Projekti pysyy vaatimammankin tekoälyagentin kohdalla siistinä ja helposti ymmärrettävänä.

Eri agenttien välillä samaa koodia voitiin käyttää vaihtamalla vain tagitunnisteita agentin prefab-tiedostosta. Agentteja oli helppo saada taistelemaan keskenään ja taistelemaan pelaajaa vastaan.

Ongelmaksi muodostui Unityn aluillaan oleva dokumentaatio käyttäytymispuusta. Haasteena on, että ongelmatilanteiden tullessa on vaikea löytää virheitä ja niiden syitä dokumentaation puutteellisuuden takia.

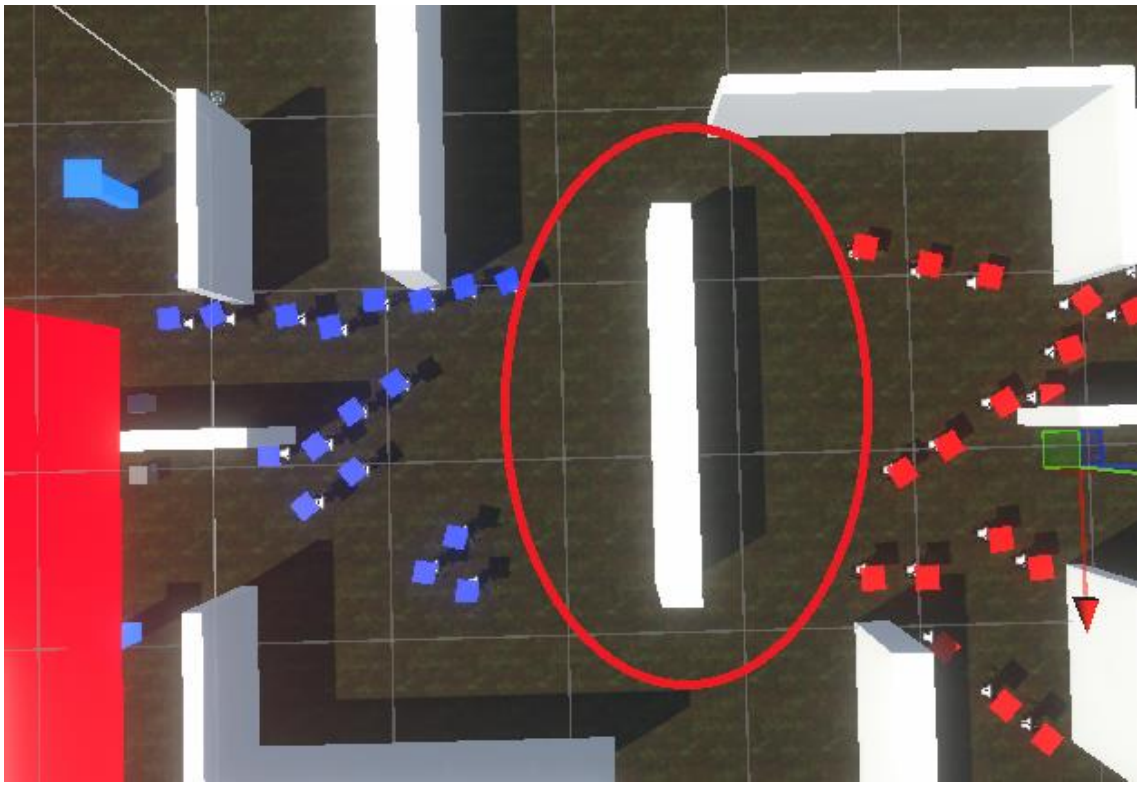
4.2 Suorituskyvyn toimivuus

Suorituskykyä mitattiin Unityn kuvan päivitysnopeutta mittaavalla järjestelmällä ja sillä seurattiin tekoälyagenttien vaikutusta kuvan päivitysnopeuteen. Testi suoritettiin nostamalla kummassakin joukkueessa agenttien määrää aina kolmella ja mittaamalla uudestaan. Kuvanpäivitysnopeus mitattiin aluksi tyhjästä ympäristöstä ja sen jälkeen tekoälyagenttien kanssa. Tyhjässä ympäristössä saatiin parhaimmillaan 111,3 kuvaa/s. Kuvassa 13 Unityn statistiikasta otettu kuvakaappaus, jossa näkyy kuvaa sekunnissa.



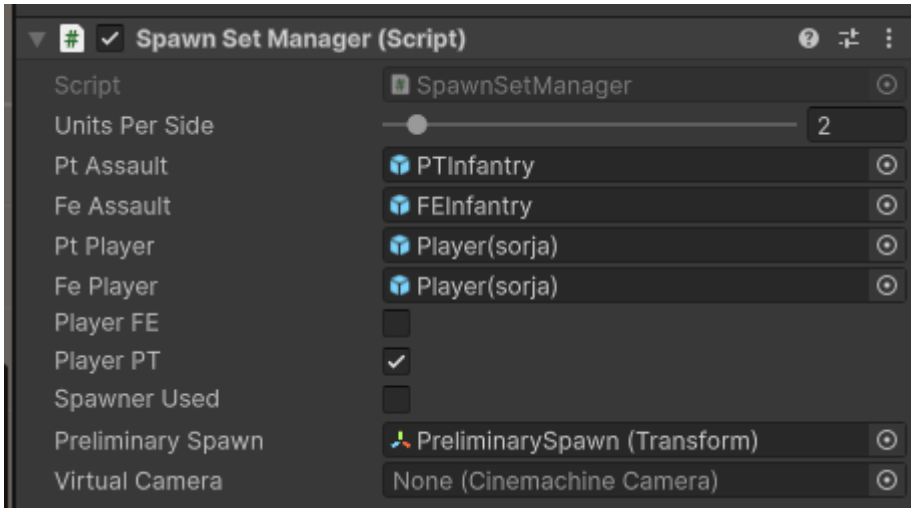
Kuva 13 Unityn statistiikka.

Agentteja lisättiin aina mittausten jälkeen ympäristöön portaittain ja mittaus suoritettiin uudelleen. Mittaus toteutettiin yhteensä kuusi kertaa per metodi. Mittaus on suoritettu siitä kohdasta jossa tekoälyagentit kohtaavat ja alkavat taistelemaan. Silloin agenttien resurssien käyttö on suurimmillaan tietokoneesta. Kuvassa 14 ympyröitynä pelitekoälyjen kohtaamispaikka, jossa joukkueet ottavat yhteen ensimmäisen kerran.



Kuva 14 Agenttien kohtaamispaikka.

Tekoälyagenttien lisäämiseen käytettiin itse ohjelmoitua työkalua, jolla pystytään lisäämään liukusäätimen avulla agentteja kummallekkin joukkueelle haluttu määrä. Kuvassa 15 esitellään lisäämistyökalun komponentin ominaisuuksia, joihin kuuluu liukusäädin ja mahdollisuus valita, mitä tekoälyagentteja peliin lisätään.



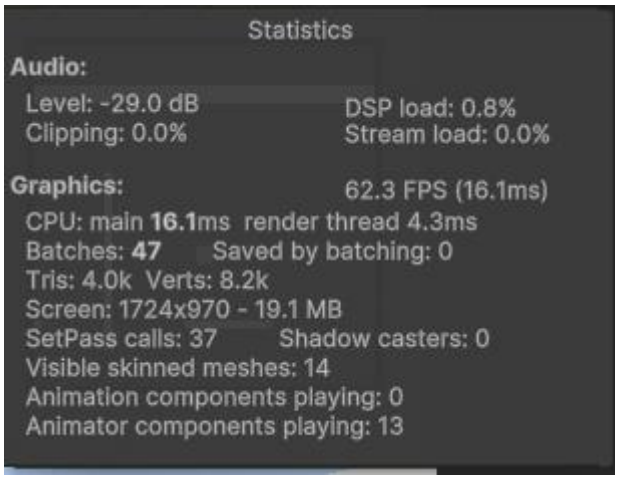
Kuva 15 Pelitekoälyjen lisäämistyökalu.

4.3 Tilakoneen suorituskyky

Testaaminen aloitettiin lisäämällä kummallekin joukkueelle kaksi agenttia eli yhteensä neljä. Agenttien kohdatessa tapahtuu selkeä pudotus pelin suorituskyvyssä. Alunperin tyhjällä ympäristöllä mitattiin 111,3 kuvaa/s ja nyt kun agentit aloittivat tulitaistelun suorituskyky putosi 92,6 kuvaan sekunnissa. Kuvassa 16 näkyy statistiikkaa neljän agentin vaikutuksista suorituskykyyn.

Kuva 16 Neljän tekoälyagentin suorituskyky.

Seuraavaksi peliin lisättiin agentteja siten, että molemmilla puolilla on kuusi tekoälyagenttia. Lisäyksessä tarkoituksena oli, että saataisiin mahdollisimman paljon vaikutusta pelin suorituskykyyn, ja sen takia agentteja lisättiin huomattava määrä. Lisäämisen vaikutuksena kuvanpäivitysnopeus putosi selvästi eli 30,3:lla. Lopullinen kuvaa sekunnissa kohtaamisessa oli 62,3 kuvaa sekunnissa. Kuvassa 17 näkyvissä statistiikka.



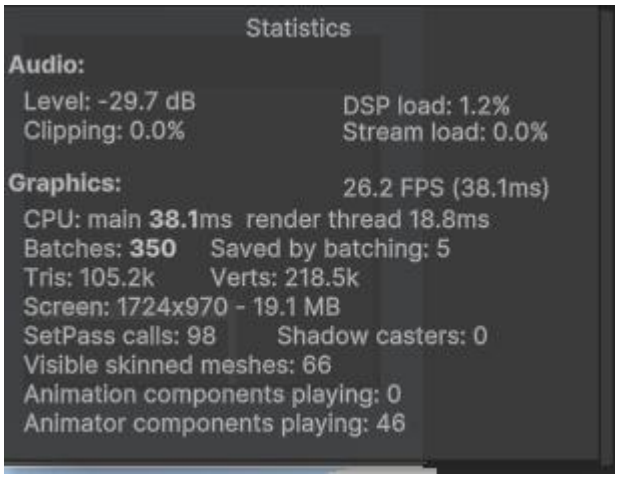
Kuva 17 12 Tekoälyagenttia ympäristössä.

12 tekoälyagentin jälkeen koe suoritettiin uudelleen lisäämällä 6 agenttia per puoli eli tekoälyagentteja oli ympäristössä yhteensä 24. Tällä kertaa pudotus suorituskyvyssä oli 9,2 kuvaa sekunnissa eli pienempi kuin 12 agentilla. Lopullinen kuvanpäivitysnopeus, kun ympäristössä oli 24 tekoälyagenttia oli 52,6 kuvaa/s.

Tämän jälkeen koe suoritettiin lisättiin 8 per puoli eli yhteensä 40. Pudotus oli tällä kertaa 8,8 kuvaa sekunnissa eli lopullinen kuvanpäivitysnopeus oli 43,8. Seuraavaksi koe suoritettiin lisäämällä tekoälyagentteja 5 per puoli eli yhteensä tekoälyagentteja pelissä oli 50. Lisäyksen johdosta kuvanpäivitysnopeus putosi 35,5 kuvaan sekunnissa, joten pudotusta viime kertaiseen oli 8,3 kuvaa sekunnissa.

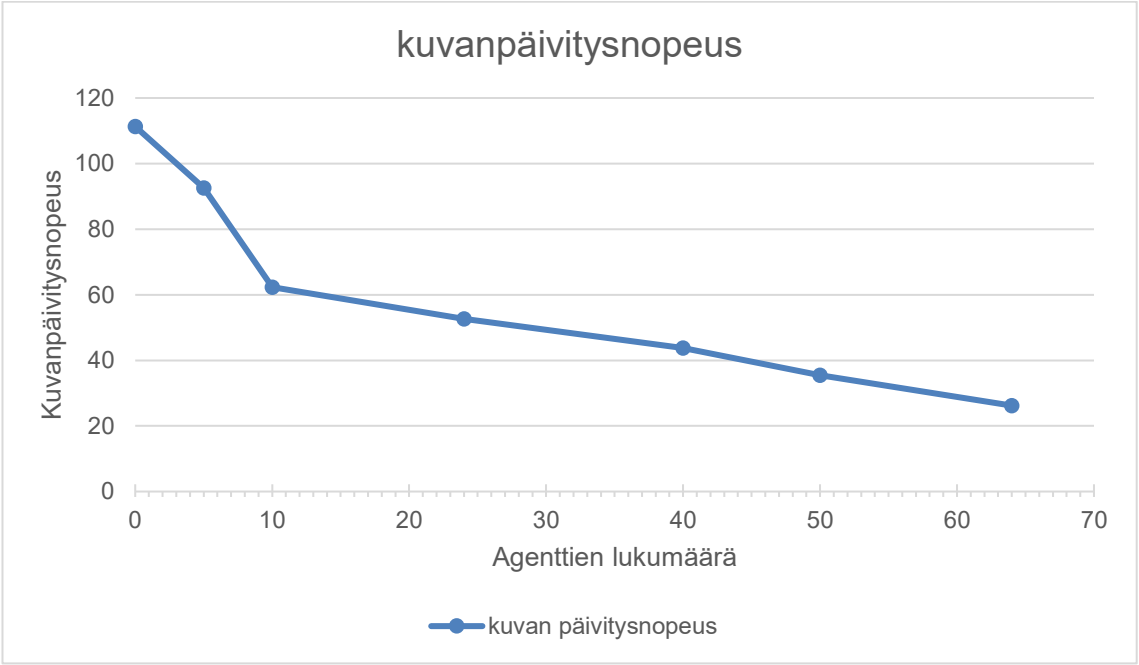
Viimeisenä testinä koe ajettiin siten, että kummallakin puolella oli 32 tekoälyagenttia eli lopullisessa kokeessa oli yhteensä 64 tekoälyagenttia. Verrattuna viimesimpään kokeeseen kuvanpäivitysnopeus putosi 9,2 kuvaa

sekunnissa verran. Kuvanpäivitysnopeus oli lopulta siis 26,2. Verrattuna tyhjään ympäristöön kuvanpäivitysnopeus on pudonnut kokonaisuudessaan yli 84 kuvaa sekunnissa. Kuvassa 18 näkyy lopulliset tulokset testistä eli paljonko suorituskykyyn oli vaikutusta tekoälyagenteilla.



Kuva 18 Lopulliset tulokset testistä.

Kuvassa 19 on havainnollistettuna, miten paljon tekoälyagenttien määrä vaikutti kuvanpäivitysnopeuteen.

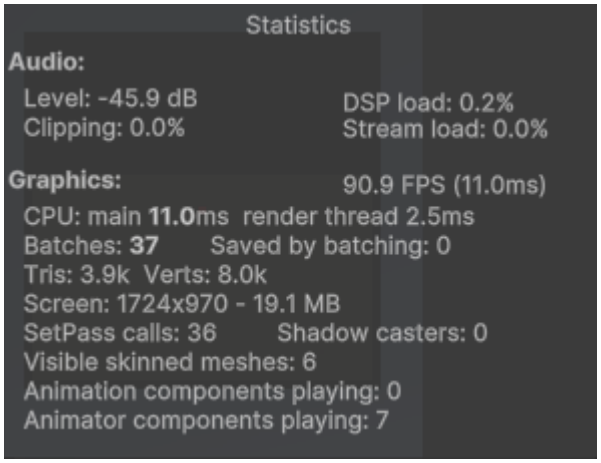


Kuva 19 Tilakoneen kuvanpäivitysnopeus.

4.4 Käyttäytymispuun suorituskyky

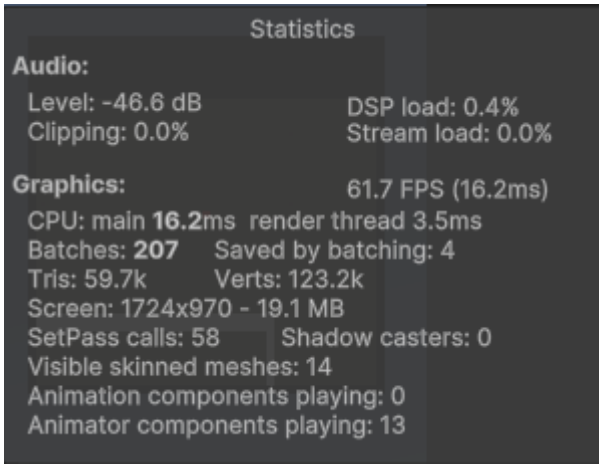
Sama koe suoritettiin myös käyttäytymispuun tekoälyagenteille.

Aluksi aloitettiin kahdella tekoälyagentilla per joukkue ja tällöin kuvanpäivitysnopeus oli 90,9 kuvaa sekunnissa. Pudotusta alkuperäiseen 111 kuvaan sekunnissa siis oli 20,1 kuvaa sekunnissa. Kuvassa 20 esitellään neljän tekoälyagentin vaikutusta suorituskykyyn.



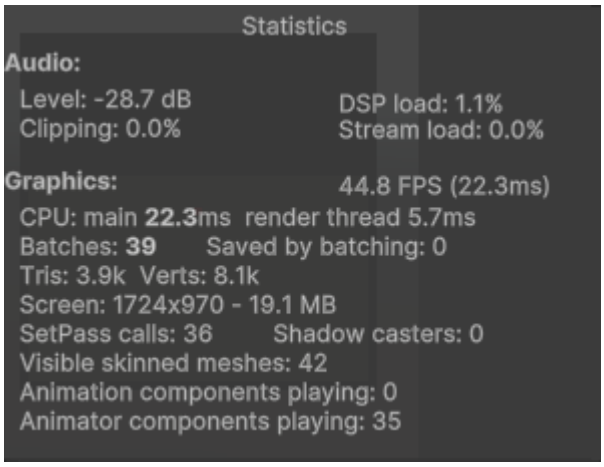
Kuva 20 Neljä tekoälyagenttia.

Seuraavaksi koe suoritettiin kuudella tekoälyagentilla per joukkue. Kuvien päivitysnopeus putosi 61,7 kuvaan sekunnissa. Pudotusta aikaisempaan oli yhteensä 29,2 kuvaa sekunnissa. Kuvassa 21 esitellään suorituskyky kuudella tekoälyagentilla per joukkue.



Kuva 21 12 tekoälyagenttia.

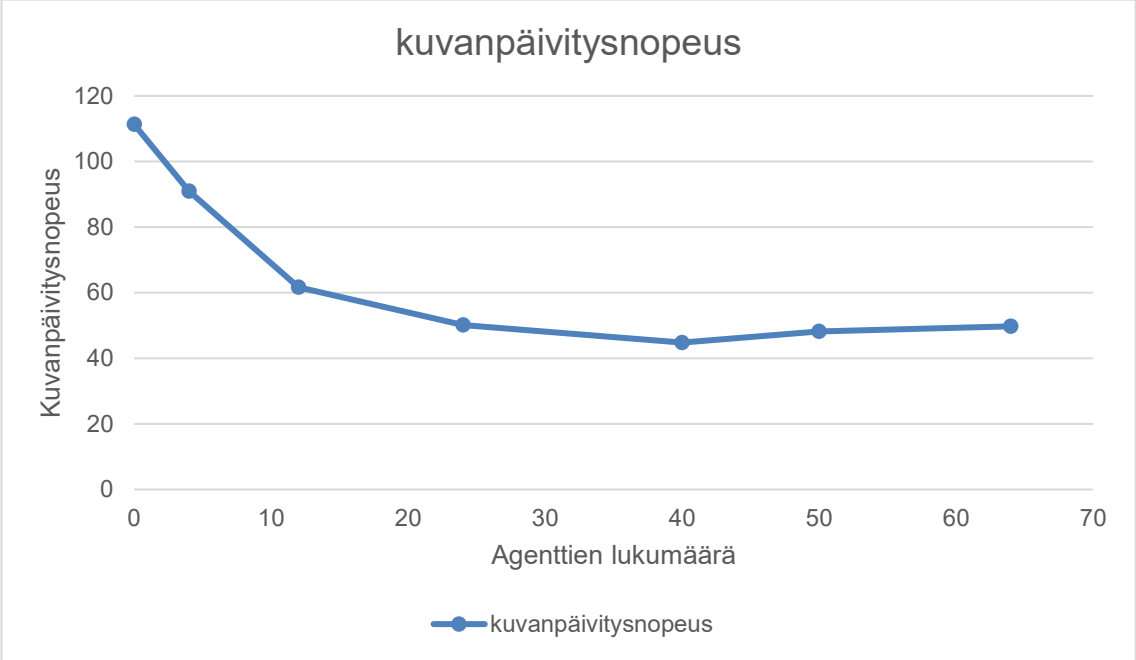
Tämän jälkeen koe suoritettiin käyttäen 12:sta tekoälyagenttia per joukkue ja pudotustusta aikaisempaan suorituskykyyn oli 11,6 kuvaa sekunnissa eli lopullinen kuvanpäivitysnopeus oli 50,1 kuvaa sekunnissa. Koe suoritettiin seuraavaksi 20:llä tekoälyagentilla per joukkue. Nyt kuvanpäivitysnopeus oli 44,8 kuvaa sekunnissa eli pudotusta aikasempaan tuli 5,3 kuvaa sekunnissa. Kuvassa 22 esitellään 40 tekoälyagentin vaikutus pelin suorituskykyyn.



Kuva 22 40 tekoälyagenttia.

Koe suoritettiin uudelleen 25:llä tekoälyagentilla per joukkue eli agentteja oli yhteensä 50. Tekoälyagenttien vaikutus suorituskykyyn oli olematon tällä kertaa ja kuvia sekunnissa saatiin enemmän kuin aikasemmin. Suorituskyky parani

3,4:llä kuvalla sekunnissa. Tulos oli yllättävä, koska oletettavasti suorituskyvyn tulisi laskea. Lopuksi koe suoritettiin lisäämällä 64 tekoälyagenttia eli tekoälyagentteja oli kummallakin puolella 32. Tulokseksi saatiin 49,8 kuvaa sekunnissa eli nousua viime kokeeseen oli yhteensä 1,6 kuvaa sekunnissa. Kuviossa kaksi käydään läpi vaikutukset kuvan päivitysnopeuteen.



Kuva 23 Käyttäytymispuun kuvanpäivitysnopeus.

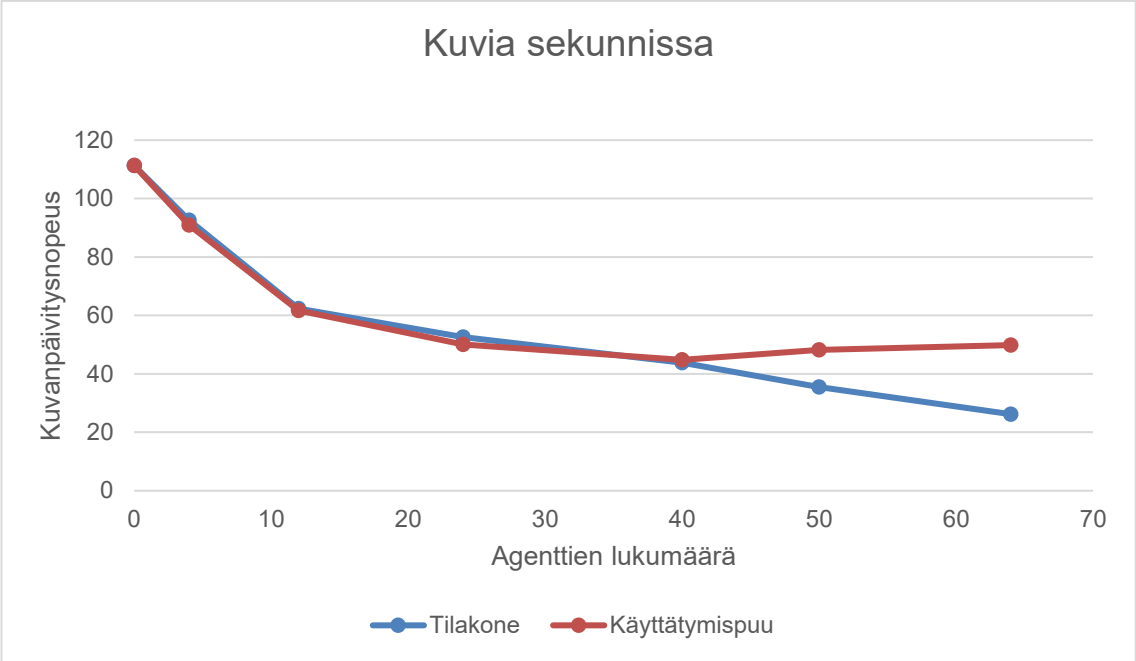
5 Tulokset

Opinnäytetyön tuloksena selvisi, että pelitekoälymalleista projektinhallinnallisesti selvästi tehokkaampi ratkaisu oli käyttäytymispuumalli. Käyttätymispuussa pystytään näkemään koko pelitekoälyn rakenne yhdellä sivulla ja nähdään kokonaan miten ja missä kohtaa mikäkin käyttäytymispuuhun määritetty asia tapahtuu. Käyttäytymispuun oksia on helppo muokata ja niihin on helppo lisätä uusia solmuja. Muuttujat pysyvät myös yhdellä sivulla helposti nähtävissä ja tilakoneen tavoin ei ole tarvetta suurelle tiedostolle joka toimisi tekoäly agentin aivoina, sillä kaikki tarvittava on jo liitutaululla. Hyvänä puolena käyttäytymispuussa on myös se, että se on valmis malli, jota ei tarvitse ohjelmoida itse alusta asti, vaan sen päälle voi aloittaa suoraan rakentamaan pelitekoälyä. Huonona puolena käyttäytymispuussa oli Unityn dokumentaation puuttellisuus, mutta tulevaisuudessa dokumentaatio todennäköisesti paranee sillä käyttäytymispuu on hyvin uusi lisäys Unityyn. C#-ohjelman muuntaminen tilakoneesta käyttäytymispuuksi oli lopulta myös helposti toteutettavissa.

Tilakoneessa eri C#-tiedostojen määrä voi kohota kymmeniin eri tiedostoihin ja niiden tarkastelua varten tiedostot on aina avattava erikseen. Tilakoneen päätiedosto, jonka tarkoituksena on pitää kirjaa kaikista eri tilakoneen komponenteista voi kasvaa satojen rivien tiedostoksi, mikä tekee sen hallinnoimisesta ja pikaisesta tarkastelusta hankalaa. Tilakoneessa on myös mietittävä tarkkaan miten, mistä ja miksi eri tilat siirtyvät toisiin tiloihin. Käyttäjän on myös itse muistettava pelitekoälyn rakenne eikä rakennettakaan voi suoraan tarkastella tehokkaasti. Verrattuna käyttätymispuuhun uuden projektin aloittamista hidastaa tilakoneessa myös se, että tilakone on ohjelmoitava alusta saakka itse. Tilakoneen hyviä puolia on kuitenkin se, että siitä saa ohjelmoitua ja muokattua sellaisen kuin projekti vaatii eli se ei ole sidoksissa valmiisiin raameihin, kuten Unityn käyttäytymispuu on. Vianmääritys voi olla helpompaa myös, koska ohjelmoija tuntee kuitenkin oman ohjelmansa ja kykenee toimimaan Unityn antamien virhekoodien mukaisesti.

Suorituskyvyn puolesta testeissä kävi ilmi, että tilakoneen suorituskykyyn vaikuttaa huimasti, paljonko tekoälyagentteja on ympäristössä taistelemassa. Alkuperäisestä kuvan päivitysnopeuden määrästä pudotusta oli yli 84 kuvaa sekunnissa. Testi suoritettiin maksimissaan 64:llä tekoälyagentilla ja pudotus alkuperäiseen 111 kuvaan sekunnissa oli suurta. Syitä suorituskyvyn huonontumiseen on monia, joista yksi on se, että tekoälyagentit pyrkivät hakemaan jokaisella kuvan päivityksellä uutta paikkaa, johon siirtyä. Tekoälyagentit suorittavat myös lukuisia tarkastuksia etäisyyksiin ja kohteiden näkemiseen jokaisella ruudun päivityksellä. Nämä kaikki vaikuttavat tekoälyn suorituskykyyn ja erityisesti navigaatioverkon tarkastelut ovat raskaita laskutoimituksia tietokoneelle.

Käyttäytymispuulle suoritettiin samat testit samoilla määrillä tekoälyagentteja kuin tilakoneelle. Pudotus alkuperäisestä kuvan päivitysnopeudesta oli pienempää kuin tilakoneen kanssa. Alkuperäisestä 111 kuvasta sekunnissa pudottiin pahimmillaan vain 44,8 kuvaan sekunnissa. Mielenkiintoisena huomiona oli, että suorituskyky parani 25 tekoälyagentin kohdalla ja kuvia sekunnissa saatiin lisää aiempaan tulokseen nähden. Sama jatkui vielä 32 tekoälyagentin kohdalla. Yhtenä tärkeimpänä syynä suorituskyvyn paranemiseen on ettei käyttäytymispuun versio tee jatkuvia kyselyitä navigaatioverkkoon. Tekoälyagentissa käytettiin myös Unityn valmiita solmuja tekoälyagentin ohjelmointiin ja ne voivat olla paremmin optimoituja kuin itse ohjelmoitu tilakoneen koodi. Kuviossa kolme esitellään kummankin testin vaikutukset suorituskykyyn ja kuviossa myös havainnollistetaan niiden erot.



Kuva 24 Testien tulokset esiteltynä.

6 Pohdinta

Opinnäytetyön tehtävänä oli selvittää miten pelitekoälyn taktista päätöksentekoa voidaan parantaa erilaisilla tavoilla. Työssä tarkasteltiin pelitekoälyn suorituskyvyn ja projektinhallinnan parantamista. Tarkastelua varten pelitekoäly toteutettiin kahdella eri tavalla, ensimmäiseksi hyödynettiin tilakonetta ja toisena käyttäytymispuuta. Tilakone ohjelmoitiin ensin C#:lla ja käännettiin käyttäytymispuuksi, joka hyödynsi Unityn omaa käyttäytymispuumallia. Opinnäytetyössä myös rakennettiin ympäristö, jossa voitiin testata pelitekoälyn toimivuutta ja sitä, miten pelitekoäly reagoi toisiin tekoälyagentteihin. Työssä myös tarkasteltiin pintapuolisesti tekoälymallien menneisyyttä ja nykyhetkeä.

Kumpaa tekoälyälymallia olisi parempi käyttää projektissa? Suurempaan projektiin sopisi paremmin Unityn käyttäytymispuu, koska projektinhallinnallisesti se on tehokkaampi. Visuaalinen käyttöliittymä helpottaa työkalun käyttöä todella paljon ja oksien muokkaaminen tuntuu helpommalta, kuin tilakoneessa eri tilojen muokkaaminen. Tilakonetta voitaisiin silti hyödyntää pienemmissä projekteissa ja myös prototyyppaukseen. Tilakone on nopea ja helppo sisällyttää projektiin ja sen avulla pääsee testailemaan tekoälyagentteja nopeasti. Lopulliseen versioon olisi parempi käyttää silti käyttäytymispuuta. Testien perusteella käyttäytymispuun suorituskyky on myös parempi, kuin itse ohjelmoidun tilakoneen suorituskyky.

Viimeisen muutaman vuoden aikana tekoälymallit ovat kehittyneet paljon ja nopeasti. Tulevaisuudessa kehitys oletettavasti kiihtyy entisestään. Kehityksen myötä on syntynyt itseoppivia tekoälymalleja, joiden avulla voidaan luoda tehokkaampia ja entistäkin fiksumpia tekoälyratkaisuja peleihin.Tilakone tai käyttäytymispuu eivät silti katoa lopullisesti käytöstä. Niiden avulla on nopeaa ja tehokasta luoda vähintäänkin prototyyppejä, joilla voi testata mitä tekoälyagenttien halutaan tekevän. Vanhempien metodien avulla pelikehittäjät myös pysyvät täydellisessä kontrollissa siitä mitä tekoälyagentit tekevät ja millaista vastusta ne antavat pelaajalle.

Lähteet

Klosowski, D. 2024. The Art of the Enemy: AI in Video Games. Viitattu 22.4.2025. https://medium.com/@danielklosowski/the-art-of-the-enemy-ai-in-video-games-ab565eb97f8e.

Sabbagh, M. 2015. How to design formidable and unforgettable video game enemies. Viitattu 15.4.2025. https://medium.com/@Watfen64/how-to-design-formidable-and-unforgettable-video-game-enemies-5ca8ed21fd8e.

Jagdale, D. 2021. Finite State machine in Game Development. Viitattu 15.4.2025. https://www.researchgate.net/publication/355518086_Finite_State_Machine_in_Game_Development.

Sekhavat, Y. 2017. Behavior Trees for Computer Games. Viitattu 15.4.2025

https://www.researchgate.net/publication/312869797_Behavior_Trees_for_Computer_Games.

Unity3D. Päiväämätön. A. Behavior graphs. Viitattu 22.4.2025

https://docs.unity3d.com/Packages/com.unity.behavior@1.0/manual/behavior-graph.html.

Unity3D. Päiväämätön. B. About Unity Behavior. Viitattu 22.4.2025.

https://docs.unity3d.com/Packages/com.unity.behavior@1.0/manual/index.html.