LAB University of Applied Sciences

Tekoälyn toteutus VR-harjoitteluympäristöön

LAB-ammattikorkeakoulu
Insinööri (AMK)
2022
Raoul Ölander

Tiivistelmä

Tekijä(t)	Julkaisun laji	Valmistumisaika
Raoul Ölander	Opinnäytetyö, AMK	2022
	Sivumäärä	
	43	

Työn nimi

Tekoälyn toteutus VR-harjoitteluympäristöön

Tutkinto ja koulutusala

Insinööri (AMK), tieto- ja viestintätekniikan koulutus

Toimeksiantajaorganisaatio (jos opinnäytetyöllä on toimeksiantaja)

Cross-Border Safety -hanke

Tiivistelmä

Opinnäytetyön tavoitteena oli toteuttaa tekoäly pelastustoimen operatiiviseen johtamiseen luotavalle VR-harjoitteluympäristöön. Tekoälyn tehtäviin kuului päätöksen teon lisäksi hahmojen liikuttaminen ja animaatioiden kontrolloiminen. Pelaajan oli kyettävä antamaan käskyjä miehistölle. Koska kyseessä on moninpeli, näiden synkronoiminen pelaajien välillä oli osana tavoitteita.

Toteutuksessa tekoäly toteutettiin tilakoneena ja yhteensä tilakoneita luotiin yhdeksän kappaletta. Näistä seitsemän hoitaa pelaajan yksikön toimintaa ja on siten pelaajan kontrolloitavissa. Hahmojen liikkuminen toteutettiin Unityn sisäänrakennetuilla navigaatioverkkotyökaluilla. Animaatioita varten luotiin oma kontrolleri, joka toimii hahmon tilakoneen mukaisesti. Pelaajien välinen synkronointi toteutettiin Photon-työkalun avulla.

Tekoälyn toteutus onnistui usean tilakoneen ja niiden välisen synkronoinnin avulla. Tämä mahdollisti tekoälyn kontrolloimisen käskyjen avulla. Animaatioiden siirtäminen tilakoneiden ulkopuolelle oli järkevä ratkaisu. Photon toimi synkronointiratkaisuna.

Asiasanat

Unity, tekoäly, tilakone, navigointi, animaatiot, moninpeli, Photon

Abstract

Author(s)	Type of Publication	Published
Raoul Ölander	Thesis, UAS	2022
	Number of Pages	
	43	

Title of Publication

Creating an artificial intelligence for the VR-training ground

Degree, Field of Study

Engineer (UAS), Information and Communications Technology

Organisation of the client (if the thesis work is commissioned by another party)

Cross-Border Safety project

Abstract

The aim for this thesis was to implement artificial intelligence (AI) for a VR-training ground used by the operative leaders of rescue operations. The goals for the AI were decision making, character movement and controlling character animations. The player should give orders to their crew. Due to the multiplayer nature of the game, player synchronization belonged to the goals.

The implementation was created by using state machine, which there were nine in total. Seven of them were part of the unit and thus controllable by the player. Character movement was made using the built-in tools in Unity. A separate controller was created for animation control. This controller uses the state machine of the character as a guide. Player synchronization was created with Photon tool.

The implementation of the AI was successful with multiple state machines and synchronization components. Moving the animations into a separate system proved to be the correct choice. Photon turned out to be a good solution for synchronization between the players.

Keywords

Unity, artificial intelligence, state machine, navigation, animations, multiplayer, Photon

Sisällys

1 Johdanto 1
2 Unity 2
2.1 Unityn historia 2
2.2 Unity 2
2.3 Unityn elementit 3
2.4 Navigointi 6
2.5 Synkronointi 9
3 Tilakone 11
3.1 Tilakoneen toimintaperiaate 11
3.2 Tekoäly ja tilakone 12
3.3 Eri tapoja toteuttaa tilakone 13
4 CB-Safe -simulaatio 15
4.1 Tilakoneen toteutus 15
4.1.1 Pelaajan syöte siirtymän laukaisijana 18
4.1.2 Useita tilakoneita eri tarkoituksiin 25
4.2 Navigointi 31
4.3 Synkronointi 35
4.3.1 Usean hahmon tila yksikön sisällä 36
4.3.2 Synkronointi pelaajien välillä 37
5 Yhteenveto ja pohdinta 40
Lähteet 42

Liite 1. Unityn skriptin elinkaari

Liite 2. Witness-tilakoneen lähdekoodi

1 Johdanto

VR-teknologioiden kehitys on mahdollistanut uuden tavan harjoitella muuten vaarallisia tai vaikeasti järjestettäviä asioita. Näihin lukeutuvat myös pelastuslaitosten miehistön harjoitukset. Varsinkin suurien harjoitusten järjestäminen on aikaa vievää ja mukana on aina riski onnettomuuksille.

Cross-Border Safety (CB-Safe) on LAB-ammattikorkeakoululta tilattu hanke. Tilaajina toimivat Etelä-Karjalan ja Kymenlaakson pelastuslaitokset. Yhtenä hankeen tavoitteista on toteuttaa VR-harjoitteluympäristö pelastustoimen operatiiviseen johtamiseen. Harjoitteluympäristön avulla on tarkoitus selvittää, voisiko osan koulutuksesta ja harjoittelusta siirtää virtuaalimaailmaan.

Pelaajat toimivat pelastustoimen ja ryhmien johtajina - miehistö ei osallistu simulaatioon, vaan tekoälyn olisi korvattava heidät. Tavoitteena on toteuttaa tekoäly, jonka avulla pelaajat pääsevät näkemään tekemiensä päätösten vaikutuksen simulaation avulla.

Miehistön lisäksi tekoälyä tarvitaan myös yksiköiden ajoneuvoille, heidän käyttämille laitteille sekä pelastettaville. Jokaisella on omat vaatimuksensa, joten tekoälyn pohjan on oltava helposti muutettavissa ja ylläpidettävissä. Tekoälyn on lisäksi oltava todenmukainen, sillä kyse on todellisuutta vastaavasta ympäristöstä.

Tekoälyn tehtäviin lukeutuu ennen kaikkea päätöksenteko. Miehistö noudattaa sille annettuja käskyjä, joten heidän on saatava tieto pelaajan päätöksestä. Ajoneuvojen ja laitteiston on myös toteutettava annetut käskyt. Pelastettavien puolestaan on kyettävä esittämään heidän jatkuvasti muuttuva tilanteensa. Harjoitteluympäristön ensimmäiseksi tapahtumaksi on valittu veneonnettomuus, minkä vuoksi pelastettavat ovat vedessä. Pelastettavien uupuminen johtaa siten hukkumiseen. Tämä tieto antaa suunnan pelastettavien tekoälyn toiminnan toteuttamiseen.

Tekoäly on lisäksi vastuussa hahmojen liikuttamisesta. Päätöksenteko ja liikkuminen muodostavat suurimman osan tekoälyn toiminnasta. Toiminta puolestaan ohjaa animaatioita, joten animaatioiden kontrolloiminen lisätään tekoälyn tehtäviin.

Huomattava haaste muodostuu tekoälyn toteutuksen lisäksi moninpeliympäristöstä. On selvitettävä, kuinka tekoäly saadaan toimimaan siten, että kaikki pelaajat näkevät hahmot toimimassa samalla tavalla ja samassa paikassa.

2 Unity

2.1 Unityn historia

Brodkin (2013) kirjoittaa artikkelissaan Dice Insights -sivustolla Unityn historiasta seuraavasti. 2000-luvun alkupuolella David Helgason, Joachim Ante ja Nicholas Francis kokoontuivat Tanskassa työstämään yhteistä projektia, jonka esikuvana toimi Applen Final Cut Pro-sovellus. Siinä missä Final Cut tarjoaa ammattilaistason työkaluja amatööreille elokuvien tekemiseen, Unity toteuttaisi saman videopelien tekijöille. (Brodkin 2013.)

Huolimatta liiketoimintasuunnitelman puutteesta ja rahoitusvaikeuksista, he julkaisivat ensimmäisen version Mac-alustalle vuonna 2005. Vuoteen 2008 mennessä tuki Windows- ja selainalustoille oli lisätty ja työkalusta oli tullut kehittyneempi. Tämän myötä yrityksen talous oli vakiintunut, ja he pystyivät kasvattamaan työntekijöidensä määrää. (Brodkin 2013.)

2008 vuoden puolivälissä heidän yrityksensä teki läpimurron. Apple julkaisi sovelluskaupan puhelimilleen, ja Unity onnistui ensimmäisenä tukemaan iPhonea alustana. Saman aikaisesti Cartoon Network loi ja julkaisi FusionFall verkkoroolipelin lapsille käyttäen Unity3D:tä kehitystyökaluna. Vuonna 2009 Electronic Arts toteuttu Tiger Woods PGA Tour Online-pelin Unityllä ja samalla Microsoft ja Ubisoft siirtyivät käyttämään Unityä. Unity osti animaatioita tekevän Mecanimin vuonna 2011, mikä vauhditti heidän omaa kehitystyötään huomattavasi. (Brodkin 2013.)

Vuonna 2021 Unityllä oli 5245 työntekijää (MacroTrends 2022). Tuettuja kehitysalustoja ovat Mac, Windows ja Linux. Kohdealustoista tuettuina ovat edellä mainittujen lisäksi Universal Windows Platform. Mobiilipuolella Android, iOS ja tvOS. Pelikonsoleista PlayStation 4 ja 5, Xbox One, Xbox Series X|S, Nintendo Switch ja Google Stadia. Selainten kautta WebGL. (Unity 2022a.)

2.2 Unity

Unity on 2D ja 3D pelimoottori, jonka perusajatuksena on tarjota työkalut jokaiselle pelinkehittäjälle osaamistasosta riippumatta. Vuosien varrella Unity on pysynyt uusien teknologioiden mukana ja laajentanut muille toimialoille, jotka hyötyvät reaaliaikaisesta 3D-kehitysympäristöstä. Unity ei ole tarkoitettu pelkästään pelien tekemiseen, vaan sitä voi hyödyntää muun muassa animaatioiden luomiseen, VR-simulaatioihin ja rakennuksien suunnitteluun. (Schardon 2022.)

Aikaisemmin jokaisella peliyrityksellä oli omat työkalunsa, joiden avulla pelejä kehitettiin. Käyttöjärjestelmien ja laitteistojen kehityksen myötä näiden työkalujen ylläpito ja

jatkokehittäminen vaikeutui ja uusien pelien tekeminen hidastui. Unity luotiin vastaamaan tähän kasvavaan tarpeeseen. Unity yhdistää muun muassa animaatiot, fysiikan, renderöinnin, tekoälyn ja äänen samaan työkaluun ja keskittyy kehittämään itse työkalua, jolloin käyttäjille jää enemmän aikaa varsinaiseen pelien tekemiseen. (Brodkin 2013.)

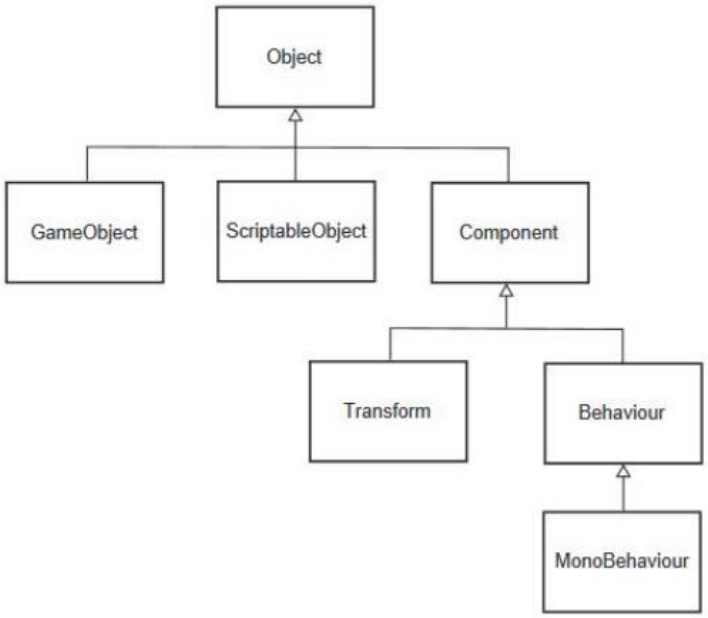
Unity on helposti lähestyttävä työkalu, jonka oppimiskynnystä madaltaa suuri tuki yhteisöltä. Unityn suuren suosion vuoksi yhteisö on kasvanut merkittäväksi tueksi käyttäjille. Käyttäjille löytyy paljon oppaita ja apua ongelmien ratkaisuun. Yhteisö on myös suurin syy, miksi Unityn markkina-alusta on kasvanut suureksi kokoelmaksi niin pieniä kuin suuriakin laajennuksia, työkaluja ja mitä yleensä pelien tekemiseen tarvitaan. (Schardon 2022.)

2.3 Unityn elementit

Peliobjekti (GameObject) on yksi Unityn tärkeimmistä perusasioista. Se on perusobjekti, joka ei itse tee mitään, vaan se toimii säiliönä komponenteille. Jokaisella säiliöllä on aina Transform-komponentti, joka kertoo objektin sijainnin ja rotaation. Lisäämällä muita komponentteja voidaan määritellä esimerkiksi valoja ja kameroita. Komponenteilla siis määritellään peliobjektin toiminnallisuus. Yhdellä peliobjektilla voi olla useampia komponentteja. Unity sisältää paljon valmiita komponentteja ja uusien tekeminen on mahdollista Unityn Scripting API:n avulla. (Unity 2022b.)

Peliobjektia ja komponentteja yhdistää sama kantaluokka, Object. Ne jakavat kantaluokkansa kanssa UnityEngine-nimiavaruuden. Object-luokka määrittelee yleisiä ja yhteisiä

metodeita, kuten Destroy ja DontDestroyOnLoad. Kuva 1 esittää UnityEngine-nimiavaruuden tärkeimmät luokat luokkakaaviona. (William 2020, 94.)



Kuva 1. Unityn Object-luokan periytyminen (William 2020, 95)

MonoBehaviour-luokka toimii perustana uusien komponenttien luomiseen. Se tarjoaa käyttöön tapahtumaviestit, joita Unity hyödyntää omassa pelisilmukassaan. Pelisilmukka on nähtävissä liitteessä 1. MonoBehaviour tarjoaa myös korutiinit, jotka ovat itsenäisesti suoritettavia asynkronisia metodeita. Niitä voidaan suorittaa muun toiminnan ohella, tai niiden avulla voidaan muu toiminta pysäyttää odottamaan korutiinin suorittamisen ajaksi. (Unity 2022c.)

Awake-metodia kutsutaan, kun komponentti ladataan ensimmäisen kerran. Kun sovellus ajetaan, ensimmäisenä jokainen aktiivinen komponentti initialisoidaan, minkä jälkeen niiden Awake-metodeja kutsutaan. Awake-metodit suoritetaan ennalta määrittelemättömässä järjestyksessä. Koska initialisointi on jo suoritettu, Awake-metodissa voidaan luoda viittauksia muihin komponentteihin ja niiden säiliöinä toimiviin peliobjekteihin. Vastaavasti, jos ajon aikana inaktiivinen komponentti tai peliobjekti aktivoidaan tai uusi objekti komponentteineen luodaan, seurataan samaa järjestystä kuin edellä mainitussa tilanteessa. Awake-metodia kutsutaan vain kerran, ja se on tarkoitettu komponentin initialisointia varten. Awake-metodi on luonteeltaan samanlainen kuin konstruktori, mutta määrittelemättömän suoritusjärjestyksen vuoksi on huomioitava, että sen aikana luotujen viittausten käyttö muiden skriptien kautta on epävarmaa. (Unity Scripting API 2022a.)

Awake-metodin ennalta määrittelemättömän suoritusjärjestyksen tuomat haasteet on mahdollista ratkaista Start-metodin avulla. Start-metodi on Awake-metodin tapaan kuin konstruktori, jota kutsutaan vain kerran, mutta se suoritetaan vasta, kun jokainen aktiivinen Awake-metodi on suoritettu. Tämä antaa mahdollisuuden ajoittaa eri komponenttien alustamista ja sitä kautta ratkaista mahdollisia ongelmia viittausten kautta käytettävän datan kohdalla. Vastaavasti, jos komponentti luodaan sovelluksen ajon alkamisen jälkeen, sen Awake-metodi ajetaan ennen Start-metodia, vaikka olemassa olevien komponenttien vastaavat metodit on jo suoritettu. Start-metodi on mahdollista määritellä korutiineiksi, toisin kuin Awake-metodi, jolloin korutiinin asynkronisia ominaisuuksia voidaan hyödyntää Start-metodin aikana. (Unity Scripting API 2022a.)

FixedUpdate-metodi ajetaan itsenäisesti omassa silmukassaan ja on siten riippumaton ruudunpäivitysnopeudesta. Oletuksena se ajetaan 50 kertaa sekunnissa. FixedUpdate-metodi suoritettaan ennen Unityn fysiikoiden laskemista, mikä toimii samalla intervallilla. Tämän vuoksi fysiikoihin liittyviä päivityksiä suositellaan tekemään FixedUpdate-metodissa. (Unity Scripting API 2022b.)

Update-metodi ajetaan ennen jokaista ruudunpäivityksellä. Se on eniten käytetty Unityn tapahtumaviestien kontrolloimista metodeista. On huomioitava, että ruudunpäivitysten laskemiseen kuluva aika ei ole vakio, vaan vaihtelee jatkuvasti. Tämän vuoksi on muistettava skaalata Update-metodissa tehtävät aikaan sidotut päivitykset Time-luokan deltaTime-arvolla. Time.deltaTime sisältää ajan, joka on kulunut edellisen ruudunpäivityksen jälkeen. (William 2022, 103.)

LateUpdate-metodi on vähemmän käytetty, mutta hyödyllinen silloin, kun Update-metodissa tehdyt päivitykset vaikuttavat toiseen päivitettävään asiaan. LateUpdate-metodi suoritetaan Update-metodin jälkeen, mutta ennen ruudunpäivitystä. Se onkin suositeltavin paikka päivittää kameran sijaintia, jos kameran sijainti on riippuvainen jonkin muun objektin sijainnin muutoksista. (William 2022, 107.)

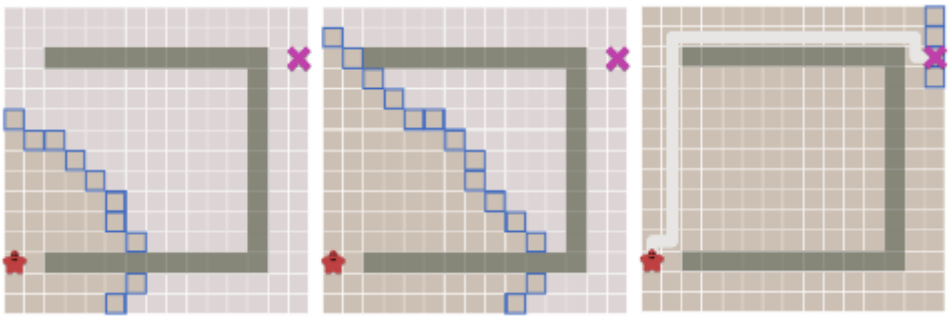
OnGUI-metodi poikkeaa edellisistä useamman kerran suoritettavista tapahtumaviestien laukaisemista metodeista siten, että se suoritetaan oman tapahtumakutsunsa perusteella. Tämä tapahtumakutsu ei ole aikaan sidottu, kuten esimerkiksi Update-metodin tapahtumakutsu lähetetään ennen jokaista ruudunpäivitystä tai FixedUpdate-metodin tapahtumakutsu lähetetään säännöllisesti ennen fysiikkalaskujen suorittamista. OnGUI-metodi suoritetaan, kun käyttöliittymäobjekti lähettää tapahtumakutsun. Kutsun lähettää mikä tahansa käyttöliittymäobjekti, jonka käyttäjä aktivoi. (William 2022, 106.)

2.4 Navigointi

Millington (2019) kirjoittaa kirjassaan, että pelihahmojen on yleensä kyettävä navigoimaan eli liikkumaan itsenäisesti. Hahmojen liikkuminen voi olla ennalta määrättyä valmiin reitin kulkemista tai satunnaisesti liikkumista rajatulla alueella. Molempia tapoja yhdistää se, ettei niitä voi erikseen kontrolloida. Hahmojen kontrolloimista varten tekoälyn on osattava laskea sopiva reitti, jota pitkin hahmo voi saavuttaa määränpäänsä. Reitin olisi oltava järkevä sekä mahdollisimman lyhyt tai helppo kulkea. Tämänlaisen reitin laskemista kutsutaan polunetsinnäksi (pathfinding). (Millington 2019, 195.)

Kyaw ym. (2013, 13) määrittelevät, että A* on yleinen polunetsintäalgoritmi, jonka hyötyjä ovat algoritmin tehokkuus ja tarkkuus. Patelin (2022a) mukaan A* yhdistää Dijkstran algoritmin tavan löytää lyhin mahdollinen reitti ja Greedy Best-First-Search -algoritmin heuristiikkaa hyödyntävän ominaisuuden. Tämä ominaisuus on yleensä arvio siitä, kuinka kaukana maalipaikka on. Molemmat algoritmit voivat ottaa huomioon eri solujen eroavat kertoimet. Kertoimella voidaan ilmaista solun esittämä maastotyyppi. Tämä mahdollistaa esimerkiksi nopeamman (kertoimeltaan matalamman) tien hyödyntämisen tai hitaamman (kertoimeltaan korkeamman) suon välttämisen. Kertoimen lisäksi molemmat algoritmit osaavat välttää esteitä ja siten löytää reitin myös esteiden ohi. (Patel 2022a.)

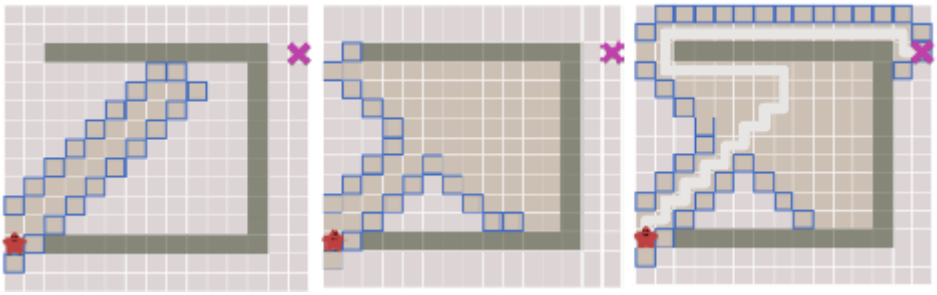
Dijsktran algoritmi tutkii lähtöpaikkaa ympäröiviä soluja, ja tutkittava alue kasvaa lähtöpaikasta ulospäin, kunnes haluttu kohde on löydetty. Seuraavassa kuvassa (Kuva 2) on visualisoitu Djikstran algoritmin toimintaa. Dijkstran algoritmi aloittaa tutkimalla lähtöpaikkaa ympäröivät solut järjestyksessä ja siirtyy sen jälkeen tutkimaan edellisellä iteraatiolla löydettyjä soluja. Lopullinen kerroin reitille on sen kulkemien solujen yhteen laskettu kerroin. (Patel 2022a.)



Kuva 2. Djikstran algoritmi visualisoituna (Patel 2022b)

Greedy Best-First-Search -algoritmi puolestaan lähtee etenemään maalipaikasta lähtöpaikkaa kohden vastaavalla iteraatioihin perustuvalla tavalla. Erona on kuitenkin se, että Greedy Best-First-Search -algoritmi painottaa kerrointa, joka johtaa maalipaikkaan. Tämä tarkoittaa

sitä, että se ei aina löydä lyhintä reittiä. Algoritmi saattaa suosia reittiä, joka palaa takaisinpäin, sillä siinä suunnassa kerroin maalia kohtaan on pienempi. Algoritmin käyttäytymistä on havainnoitu Kuva 3. Heuristiikkansa ansiosta Greedy Best-First-Search -algoritmi on kuitenkin nopeampi kuin Dijkstran algoritmi. Se suosii tavoiteltua suuntaa, kun Dijkstran algoritmi puolestaan tutkii kaikkia mahdollisia suuntia. (Patel 2022a.)



Kuva 3. Greedy Best-First-Search -algoritmi visualisoituna (Patel 2022b)

A* yhdistää Dijkstran algoritmin lasketun kertoimien lähtöpaikasta maalipaikkaa kohden ja laskee jokaisella iteraatiolla käsitellyille soluille arvioidun kertoimen maalipaikkaan asti. Kuva 4 esittää, kuinka A* hyödyntää edellä mainittujen algoritmien ominaisuudet. Matemaattisesti käytetään termejä g (laskettu kerroin) ja h (arvioitu kerroin), jotka yhdistyvät f-arvoksi yhteenlaskulla. A* käyttää tällä tavoin saatua f-arvoa seuraavan iteraation lähtökohtana. (Patel 2022a.)

Kuva 4. A* yhdistää kahden algortmin parhaat ominaisuudet (Patel 2022b)

Heuristiikkaa vaihtelemalla voidaan kontrolloida A*-algoritmin käyttäytymistä. Käytännössä tämä tapahtuu muuttamalla g- ja h-arvojen painokerrointa laskuissa. Ääripäissä A* käyttäytyy kuten tilannetta vastaava perusalgoritmi. Jos h-arvo on nolla, vain g-arvolla on merkitystä ja kyse on puhtaasta Dijkstran algoritmista. Vastaavasti, jos g-arvo on nolla, vain h-arvolla on merkitystä ja kyse on puolestaan Greedy Best-First-Search -algoritmista. Näiden ääripäiden välissä g-arvon ja h-arvon suhde määrittelee algoritmin nopeuden ja sitä kautta tehokkuuden. Jos h-arvo on pienempi kuin g-arvo, lyhimmän reitin löytäminen on varmaa,

mutta mitä pienempi h-arvo on, sitä useampia soluja on tutkittava kyseisen iteraation aikana. Jos h-arvo ja g-arvo ovat täsmälleen samat, pysytään parhaalla reitillä ja algoritmin tehokkuus on optimaalinen. Tämä on samalla harvinaisin tapaus. Nopeutta tuo myös se, jos h-arvo on suurempi kuin g-arvo, tosin tämä ei takaa lyhimmän reitin löytymistä. (Patel 2022c.)

Heuristiikan ja solujen kertoimien mukauttaminen tuovat A*-algoritmille paljon muuntautuvuutta, jota voidaan hyödyntää pelin toiminnan ja optimoinnin toteutuksessa. Yleisellä tasolla pelien ei tarvitse aina löytää parasta reittiä, vaan sitä lähellä oleva reitti riittää. Esimerkkitilanteessa pelissä voisi olla tasamaata ja vuoria. Tasamaan kerroin per solu olisi yksi ja vuorien kerroin per solu olisi kolme. A*-algoritmia voi tässä tilanteessa nopeuttaa muuttamalla tasamaan kerrointa hieman suuremmaksi, kuten 1.5, jolloin tasamaan reitti ja sen laskeminen nopeutuu. Vastaavasti vuorien kerrointa voisi laskea hieman, mikä puolestaan suosii vuoren ylittämistä ja samalla vähentää tasamaan reitin laskemiseen käytettävää aikaa. Lopputulos on siis sama molemmilla tavoilla, ja hyöty on selvä. (Patel 2022c.)

Kyaw ym. (2013, 20–21) mukaan yksinkertainenkin esimerkki A*-algoritmista osoittaa jo sen, että reitin laskemiseen tarvitaan paljon iteraatioita ja sitä kautta suoritustehoa. Ensimmäisenä ratkaisuna alettiin käyttämään reittipisteiden avulla luotua reittiä. Tämä tapa on tehokas, mutta ongelmia ilmestyy, jos esteitä halutaan muuttaa. Tilanne johtaa siihen, että reitti ja sen reittipisteet on laskettava uudelleen, mikä puolestaan aiheuttaa siksak-liikettä. Vaikka siksak-liike saataisiin optimoitua, reittipisteet eivät tarjoa tietoa ympäristöstä. Reitti saattaakin uudelleenlaskemisen jälkeen kulkea esimerkiksi jyrkänteen yli. Vaatisi paljon reittipisteitä ja paljon optimointia niiden laskemiseen, jotta menetelmä säilyisi toimivana, tehokkuudesta puhumattakaan. (Kyaw ym. 2013, 20–21.)

Vastaus reittipisteiden tuomiin haasteisiin löytyy navigaatioverkosta (navigation mesh), joka toimii periaatteessa samalla tavalla kuin A*-algoritmin solut. Erona on se, että yksittäiset solut voivat olla erikokoisia ja erimuotoisia. Jokainen solu voisi siten esittää omaa elementtiään ympäristöstään. Toisena etuna on mahdollisuus luoda erilaisia AI-kokonaisuuksia (agentteja), jotka voivat käyttää samaa navigaatioverkkoa. Jokainen agentti voi pitää sisällään tietoa esimerkiksi koosta, nopeudesta ja liikkumiskyvystä. Näiden navigaatioverkkojen sisällä voidaan hyödyntää reittipisteitä turvallisesti ja reittipisteiden uudelleenlaskeminen helpottuu, jos navigointiverkkoa päivitetään. Agentit puolestaan mahdollistavat toisistaan eroavat liikkumismallit eri agenttien välillä. Ihmiselle vuori voi olla este, mutta lintua se ei edes hidasta. Unity esitteli navigaatioverkon ensimmäisen kerran Pro-ominaisuutena versiossa 3.5 vuonna 2012. (Kyaw ym. 2013, 22.)

Unity jakaa navigoinnin kahteen ongelmaan: määränpään löytäminen tason avulla ja miten määränpäähän liikutaan. Määränpään löytäminen luokitellaan globaaliksi ongelmaksi, sillä sen ratkaisemiseksi tarvitaan koko tasoa. Määränpäähän liikkuminen puolestaan luokitellaan lokaaliksi ongelmaksi, koska ratkaisuun vaikuttavat vain liikkumissuunta ja esteiden väistäminen. (Unity 2022d.)

Määränpään löytämistä varten Unity hyödyntää navigointiverkkoa, joka luodaan automaattisesti tason geometriaa ja agentin määrittelyitä käyttämällä. Navigaatioverkko muodostuu polygoneista, jotka määrittelevät alueen rajat, jokaisen polygonin naapurit sekä alueen nimen ja kertoimen. Tämän tiedon avulla tiedetään, että alueen sisällä ei ole staattisia esteitä, joten agentti voi liikkua sen sisällä vapaasti. Reitin löytämistä varten Unity käyttää A*-algoritmia ja reittipisteitä, kuten aikaisemmissa kappaleissa on kerrottu. (Unity 2022d.)

Navigointiverkko ratkaisee ympäristön vaikutuksen reitin valintaan, mutta se ei huomioi liikkuvia esteitä kuten muita agentteja. Ratkaisu liikkuvien esteiden väistämiseen on annettu agentin tehtäväksi. Reittiä ei tarvitse laskea uudestaan, vaan agentti hyödyntää seuraavan reittipisteen sijaintia, määränpään sijaintia sekä omaa nopeuttaan välttääkseen törmäyksen toisen agentin kanssa. Toisin sanoen, agentti pyrkii seuraamaan navigointiverkon avulla laskettua reittiä, mutta poikkeaa siitä väistääkseen liikkuvia esteitä. (Unity 2022d.)

2.5 Synkronointi

Photon Unity Networking (PUN) on Unitylle luotu paketti, joka on tarkoitettu moninpelejä varten. PUN käyttää omia Photon-palvelimia, joiden kautta pelaajat voivat liittyä samaan peliin ilman tarvetta luoda yhteyksiä suoraan pelaajien välille. Palvelimella peli puolestaan jaetaan huoneisiin, joiden sisällä huoneen ja pelaajan ominaisuudet sekä objektien synkronointi on mahdollista. (Photon 2022a.)

Peliobjekti voidaan muuntaa verkko-objektiksi lisäämällä siihen PhotonView-komponentti. Tämä komponentti määrittelee objektille omistajan ja huolehtii objektiin liitettyjen synkronointikomponenttien toiminnasta. Valmiita synkronointikomponentteja ovat muun muassa PhotonTransformView, joka vastaa objektin Transform-komponentin synkronoinnin. Toinen esimerkki valmiista synkronointikomponentista on PhotonAnimationView, joka puolestaan synkronoi objektiin liitetyn Animator-komponentin pelaajien välillä. (Photon 2022b.)

Omien synkronointikomponenttien tekeminen onnistuu IPunObservable-rajapinnan kautta. Tätä kutsutaan objektisynkronoinniksi. Komponentin on toteutettava kyseinen rajapinta ja siihen liittyvä OnPhotonSerializeView-metodi. Metodin on huolehdittava siitä, että halutut arvot kirjoitetaan ja luetaan verkosta. (Photon 2022c.)

Remote Procedure Call (RPC) laajentaa metodeja siten, että kuka tahansa voi lähettää kutsun palvelimelle. Kutsun saamisen jälkeen, palvelin lähettää sen jokaiselle asiakkaalle, myös alkuperäisen kutsun lähettäjälle. Palvelimen lähettämän kutsun jälkeen asiakkaat suorittavat kutsun mukaisen metodin. PhotonView-komponentti huolehtii myös RPC-kutsuista, joten RPC-kutsun sisältämä komponentti vaatii lisäksi PhotonView-kopmponentin samaan peliobjektiin. (Photon 2022d.)

RPC on yksinkertaistettu malli Photonin tapahtumakutsuista. Sen käsittelystä vastaa PhotonView-komponentti, mikä sitoo RPC:t peliobjekteihin. Photon tarjoaa myös mahdollisuuden luoda omia tapahtumakutsuja, mikä tarjoaa monipuolisen alustan tapauskohtaiselle synkronoinnille. Photonin tapahtumakutsut eivät ole sidottuja peliobjekteihin, mikä tekee niistä joissain tapauksissa ainoan synkronointivaihtoehdon. Tapahtumakutsut ovat turvallisin tapa synkronoida tärkeää tietoa, sillä ne voidaan salata. (Photon 2022d.)

Edellä mainitut synkronointimenetelmät eroavat toisistaan toiminnallisesti ja tarjoavat ratkaisuja eri tilanteisiin. Objektisynkronointi on parhaimmillaan, kun synkronointi koskee usein muuttuvaa dataa. Tämän vuoksi esimerkiksi aikaisemmin mainittu PhotonTransformView käyttää objektisynkronointia. RPC- ja tapahtumakutsut soveltuvat epäsäännöllisten muutosten synkronointiin. Yleensä pelaajan toimintaan liittyvät asiat, kuten varusteiden vaihtaminen, ovat epäsäännöllisiä muutoksia. (Photon 2022c.)

Objektisynkronoinnin ja kutsupohjaisten synkronointien ero ei kuitenkaan ole selkeä. Jos objektisynkronointia käytetään jo saman peliobjektin synkronoimiseen, ei yleensä ole syytä lisätä samaan kokonaisuuteen RPC- tai tapahtumakutsuilla toteutettua synkronointia. Objektisynkronointi voi hoitaa kaiken. Huomattavaa on se, että RPC- ja tapahtumakutsujen synkronointi voidaan puskuroida, toisin kuin objektisynkronointi. Tämä mahdollistaa niiden kautta synkronoitavan datan lähettämisen myös myöhemmin peliin liittyville pelaajille. (Photon 2022c.)

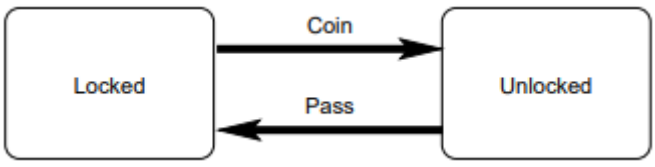
3 Tilakone

3.1 Tilakoneen toimintaperiaate

Tepfenhart & Wang (2020, 34) määrittelevät tilakoneen matemaattiseksi malliksi, jonka avulla pyritään kuvaamaan abstraktia konetta. Tämä kone voi olla yhteen aikaan vain yhdessä tilassa. Tila voidaan määritellä siten, että kone toteuttaa tietyt toiminnot tietyssä tilassa, ja jokainen tila eroaa toiminnoiltaan toisistaan. Tiloja on ennalta määritelty määrä, ja jokainen tila määrittelee siitä eteenpäin johtavat siirtymät. Siirtymien laukaisijana voivat olla ulkopuoliset syötteet tai sisäinen tapahtuma. (Tepfenhart & Wang 2020, 34.)

Matemaattisesti tilakone määritellään yksinkertaisimmillaan neljän arvon mukaan. Nämä arvot ovat: määritetyt tilat, määritetyt syötteet, määritetyt siirtymät tilasta toiseen ja aloitustila. Jos jokainen tila johtaa vain yhteen uuteen tilaan minkä tahansa syötteen perusteella, tilakone on luonteeltaan deterministinen (deterministic finite automata, DFA). Jos jokin tila voi johtaa useampaan eri tilaan syötteen perusteella, kyseessä on epädeterministinen tilakone (nondeterministic finite automata, NFA). Matemaattinen malli kuvastaa tilakoneen toimintaa erittäin hyvin, mutta sitä on vaikea tulkita. Tämän vuoksi tilakoneen toimintaa kuvataan yleisemmin graafisessa muodossa. Yksinkertainen graafi tilakoneelle ilmaisee jokaisen tilan soluna ja piirtää solujen välille siirtymät nuolilla, joilla on selitteenä siirtymän laukaiseva ehto. (Tepfenhart & Wang 2020, 35.)

Wrightin (2005, 1) mukaan yksinkertainen esimerkki tilakoneesta on kääntöportti, joka toimii kolikolla. Portti on aluksi lukossa eikä sen läpi voi kulkea. Kun portin aukosta tiputetaan kolikko, lukko aukeaa. Portin läpi voi kulkea, ja se lukittuu uudelleen ensimmäisen kulkijan jälkeen. Kuvauksen perusteella kääntöportilla on kaksi tilaa: lukittu ja auki. Portti ei voi olla samanaikaisesti molemmissa tiloissa. Tilojen välillä on myös selvät siirtymät. Kolikko avaa portin, ja läpikulkeminen lukitsee sen. Tilakonemalli kääntöportista voidaan esittää Kuva 5 mukaisesti. (Wright 2005, 1.)



Kuva 5. Tilakonemalli kääntöportista (Wright 2005, 2)

Tilalla voi olla toimintaa, joka voidaan jakaa kolmeen vaiheeseen: tilan aluksi, tilan aikana ja tilan lopuksi. Toiminta kuvastaa mitä tilan aikana tapahtuu, muttei suoraan johda siirtymään. Tilalla ei aina ole toimintaa, ja sen jokaisella vaiheella ei tarvitse olla erillistä

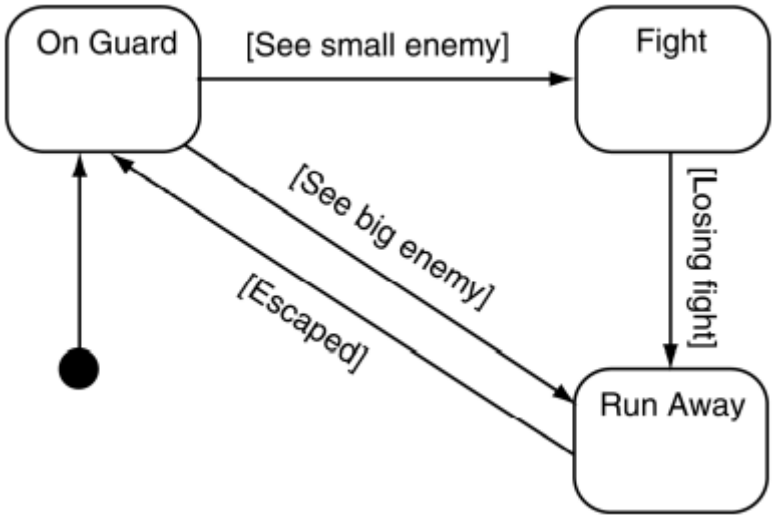
toimintaa. Siirtymän aiheuttaa erillinen tapahtuma (event) tai tilan toiminnan suorittaminen loppuun. Tapahtuma voi olla ehdollinen, jolloin siirtymä on suojattu muiden ehtojen avulla. Siirtymällä voi myös olla toimintaa, joka suoritetaan ennen seuraavan tilan alkamista. Jos tilalle on määritelty toimintaa, joka suoritetaan tilan lopuksi, siirtymä alkaa sen suorittamisesta, minkä jälkeen siirtymä suoritetaan loppuun. (Wright 2005, 15–17.)

3.2 Tekoäly ja tilakone

Millington (2022, 9–10) kirjoittaa kirjassaan, että Pac-Man oli ensimmäinen tekoälyä hyödyntävä peli. Pelissä vihollisten käyttäytyminen on toteutettu yksinkertaisella tilakoneella, joka koostuu kolmesta tilasta. Viholliset voivat jahdata pelaajaa, hajaantua (hakeutua kohti pelialueen kulmia) ja pyrkiä karkuun, kun Pac-Man syö ”power up”:n. Jatkuvasti liikkeellä olevat viholliset valitsevat kohteensa tilan määrittämän säännön mukaisesti. (Millington 2022, 9–10.)

Pac-Man pelin viholliset seuraavat yksinkertaista logiikkaa. Ne liikkuvat suoraan, kunnes saavuttavat risteyksen. Karatessaan ja hajaantuessaan viholliset käyttäytyvät kaikki samalla tavalla. Eroja syntyy silloin, kun ne jahtaavat pelaajaa. Jokainen vihollinen määrittelee kohteensa eri säännön mukaan. Punainen vihollinen, Blinky, pyrkii aina pelaajan luo. Pinkki Pinky tähtää neljä ruutua pelaajan eteen. Vaaleansininen Inky valitsee offsetin oman ja pelaajan sijainnin perusteella. Oranssi Clyde tähtää pelaajaan, jos se on kaukana pelaajasta, tai lähimpään kulmaan, kun pelaaja on sen lähellä. Koska jokainen vihollinen käyttäytyy eri tavalla, niiden liikkeiden ennakoiminen on huomattavasti vaikeampaa. Yksinkertainen tekoäly voi vaikuttaa älykkäämmältä kuin se on, jos se toteutetaan hyvin. (Millington 2022, 22.)

Tilakone on edelleen yleisin tekniikka, jonka avulla pelin hahmot tekevät päätöksiä. Jokainen hahmo on tietyssä tilassa, joka kontrolloi hahmon toimintaa ja käyttäytymistä. Siirtymät yhdistävät eri tiloja, ja tila tarkkailee ehtoja, jotka laukaisevat siirtymän. Yksinkertainen malli pelihahmon mahdollisesta tilakoneesta on näkyvillä Kuva 6. Tilakoneessa on kolme tilaa, joita yhdistää siirtymät ja niiden ehdot. Toisin kuin esimerkiksi päätöksentekopuussa, tilakoneen siirtymät koskevat vain sitä tilaa, josta siirtymä on mahdollista. Tämä mahdollistaa yksisuuntaiset siirtymät ja siten monipuolisemmat mahdollisuudet päätöksenteolle. (Millington 2019, 314–315.)



Kuva 6. Yksinkertainen tilakonemalli pelihahmolle (Millington 2019, 315)

3.3 Eri tapoja toteuttaa tilakone

Wrightin (2005, 19–22) mukaan tilakoneen perustana on siirtymät, jotka laukeavat, kun jokin tietty ehto täyttyy. Tämän toteuttamiseksi funktionaalisen ohjelmoinnin periaatteella esille nousevat if- ja switch-rakenteet. Lisäksi tarvitaan silmukka, jonka vastaa tilan toiminnoista ja valvoo siirtymien ehtojen toteutumista. While-rakenne toimii hyvin silmukan runkona. Yksinkertaisimmillaan tilakone voidaan siten toteuttaa while-silmukassa, jonka sisällä on joko peräkkäisiä if-lausekkeita tai switch-rakenne, joka valvoo, missä tilassa tilakone milloinkin on. Lisäämällä jokaisen tilan toiminnot switch-rakenteen sisälle, saadaan tilakoneesta toimiva kokonaisuus. Haasteena tässä tavassa on sovelluksen ylläpito, kun tilakone kasvaa muutamia tiloja suuremmaksi. (Wright 2005, 19–22.)

Siirryttäessä olio-ohjelmointiin, voidaan tilakone luoda omana luokkana. Tämä lähestymistapa mahdollistaa luokkien ominaisuuksien, kuten periytyminen ja polymorfismi, hyödyntämisen. Tilat jakavat osan tarvittavista attribuuteista ja metodeista, joista luodaan kantaluokka, josta jokainen tila periytetään. Jokainen periytetty tila määrittelee sille ominaiset osat toiminnasta ja siirtymien ehdoista. Tilakoneen ei tarvitse tietää muuta, kuin sillä hetkellä aktiivisena oleva tila. Tämä toteutus on joustavampi ja helpommin ylläpidettävissä kuin funktionaalisen periaatteen avulla luotu toteutus. (Wright 2005, 22–23.)

Millington (2019, 316–320) esittelee kirjassaan kovakoodatun tilakoneen, jossa jokaisen tilan toiminta, siirtymät ja niiden laukaisijat on kirjoitettu jokaiseen kantaluokasta periytettyyn tilaan. Tietorakenteita ja rajapintoja hyödyntämällä voidaan luokkien polymorfismista luonnetta parantaa ja siirtää tilan toiminnasta vastaava data erilleen. Näin tarvitaan vain yksi toteutus tilaa kuvaavasta luokasta, jossa tilan toiminta, siirtymät ja niiden laukaisijat haetaan muualta, kun luokka luodaan. Tämä vaatii hieman vaativamman logiikan toimiakseen ja

usein siirtymien laukaisijoille on luotava oma rakenteensa, jotta useampia ehtoja voidaan verrata samanaikaisesti. Tapa on kovakoodattua versiota joustavampi, sillä toteutettua logiikkaa kontrolloidaan muuttamalla sille tallennettua dataa. Tämä mahdollistaa skriptikielien ja pelin ulkoisten työkalujen käytön tilakoneen toiminnan luomisessa. (Millington 2019, 316–320.)

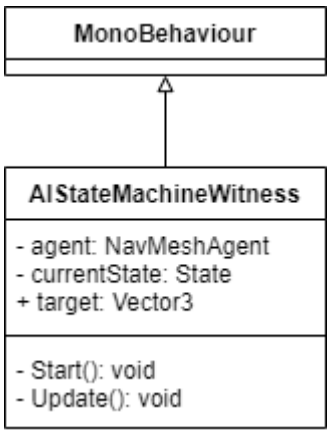
Pelimoottoreiden yleistymisen myötä niiden sisältämät työkalut ovat tuoneet uusia tapoja toteuttaa pelejä ja niiden sisältämiä osia. Datan ja toiminnan lisääminen suoraan objektiin editorin kautta mahdollistaa erilaisen lähestymistavan luoda pelimaailmaa ja vaikuttaa pelin toimintaan. Itse pelimoottorin sisältämät työkalut eivät aina vastaa niiden käyttäjien tarpeita, joten pelimoottoreihin on usein saatavilla muiden tekemiä laajennuksia. Näihin laajennuksiin sisältyy myös tekoälyn luomiseen soveltuvia työkaluja, kuten Opsiven luoma Behavior Designer Unitylle. Kyseinen työkalu mahdollistaa käyttäytymispuiden luomisen visuaalisen käyttöliittymän avulla. Vastaavien työkalujen yleistyminen on johtanut siihen, että yleisimmät pelimoottorit tukevat visuaalista ohjelmointia toiminnan, kuten tekoälyn, luomiseen ilman ohjelmointiosaamista. Yhteistä visuaalisille työkaluille on se, että niissä toiminta toteutetaan luomalla yksittäisiä toteutuksia, joita voidaan yhdistellä. Tapa soveltuu hyvin päätöksentekopuille, käyttäytymispuille ja tilakoneille. (Millington 2019, 891–893.)

4 CB-Safe -simulaatio

4.1 Tilakoneen toteutus

Tekoälyn toteuttamiseksi oli selvitettävä, mitkä ovat tilakoneen tehtävät. Tilakone toimii ensisijaisesti päätöksentekijänä, mutta sen toimintaan lukeutuvat myös hahmojen liikuttaminen sekä animaatioiden kontrolloiminen. Tilakoneella on lisäksi vastuu huolehtia siitä, että toiminta on todenmukaista ja mahdollista. Tilakoneen tehtäviin sisältyy myös hoitaa siirtymien aikaiset asiat, jotka suoritetaan ennen tilan alkamista ja tilan jälkeen. Näiden lisäksi, tilakone olisi kiinnitettävä suoraan hahmoon. Unityllä tämä on mahdollista luomalla komponentti, joka lisätään hahmoon.

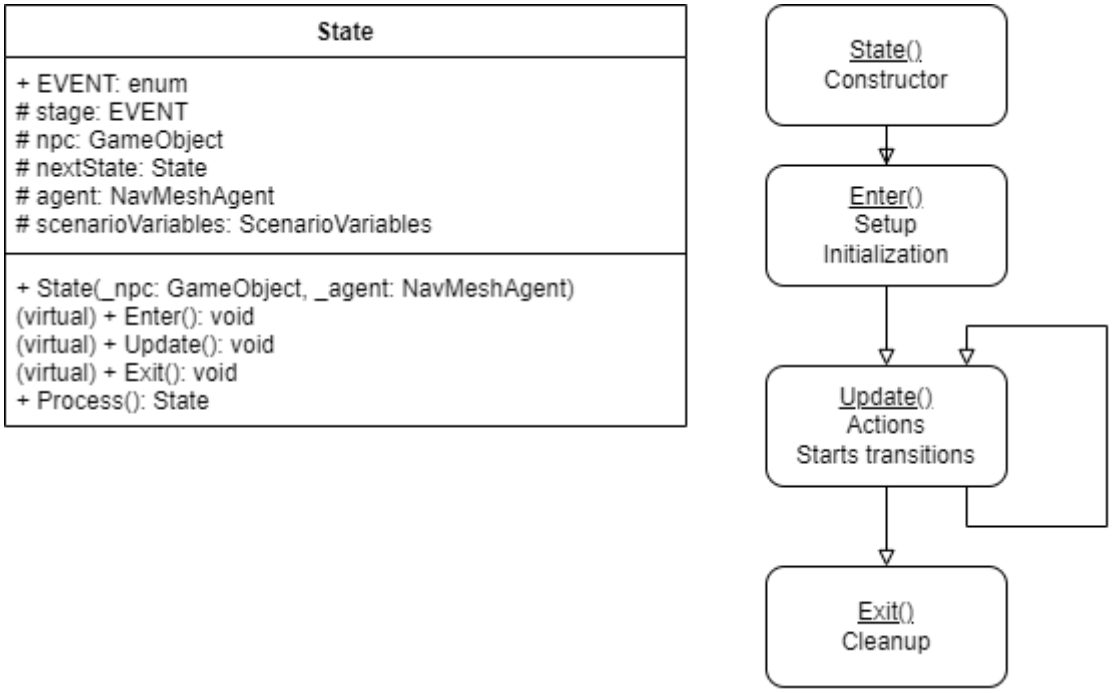
Koska komponentti toimii yhteytenä hahmon ja tilakoneen välillä, päätettiin, että komponentti tarjoaisi tilalle sen tarvitsemat tiedot ja yhdistäisi tilakoneen pääsilmukan Unityn kanssa. Komponentin on siten tiedettävä mikä tila on aktiivinen ja mikä on tilakoneen aloitustila. Komponentin tehtävät ovat aloitustilan luominen ja tilan etenemisen seuranta. Etenemisen seurantaa varten tilalle luotiin kolme vaihetta, jotka vastaavat asioita, jotka tilan on tehtävä ennen alkamista, tilan aikana ja tilan jälkeen ennen seuraavan tilan alkamista. Komponentti huolehtii siitä, että tilan vaihe päivittyy Unityn pääsilmukan aikana. Kuva 7 esitetään esimerkki yksinkertaisimmasta projektin aikana toteutetusta tilakonekomponentista. Witness-tilakoneen lähdekoodiin voi tutustua liitteessä 2. Luokka AIStateMachineWitness periytyy Unityn luokasta MonoBehaviour, joten sitä voidaan käyttää komponenttina. Attribuutti currentState pitää sisällään aktiivisen tilan, joka asetetaan Start-metodissa osoittamaan aloitustilaa. Update-metodi kutsuu aktiivisen tilan Process-metodia, jonka tehtävänä on päivittää tilan sen hetkinen vaihe ja palauttaa tieto seuraavasta tilasta, kun siirtymä tapahtuu.



Kuva 7. Esimerkki hahmon tilakonekomponentista

Tilakone toteutettiin omana kokonaisuutenaan, joka ei yksin käytä Unityn pelisilmukkaa. Se on riippuvainen edellä mainitusta komponentista, joka toimii rajapintana Unityn ja tilakoneen

välillä. Tilakonetta varten luotiin kantaluokka State, jonka rakenne ja toiminta on nähtävissä Kuva 8. Kantaluokan attribuuteista tilakoneen kannalta tärkeimmät ovat stage, joka käyttää EVENT-attribuuttia määrittelemään tilan aktiivisen vaiheen, ja nextState, joka sisältää tiedon seuraavasta tilasta.

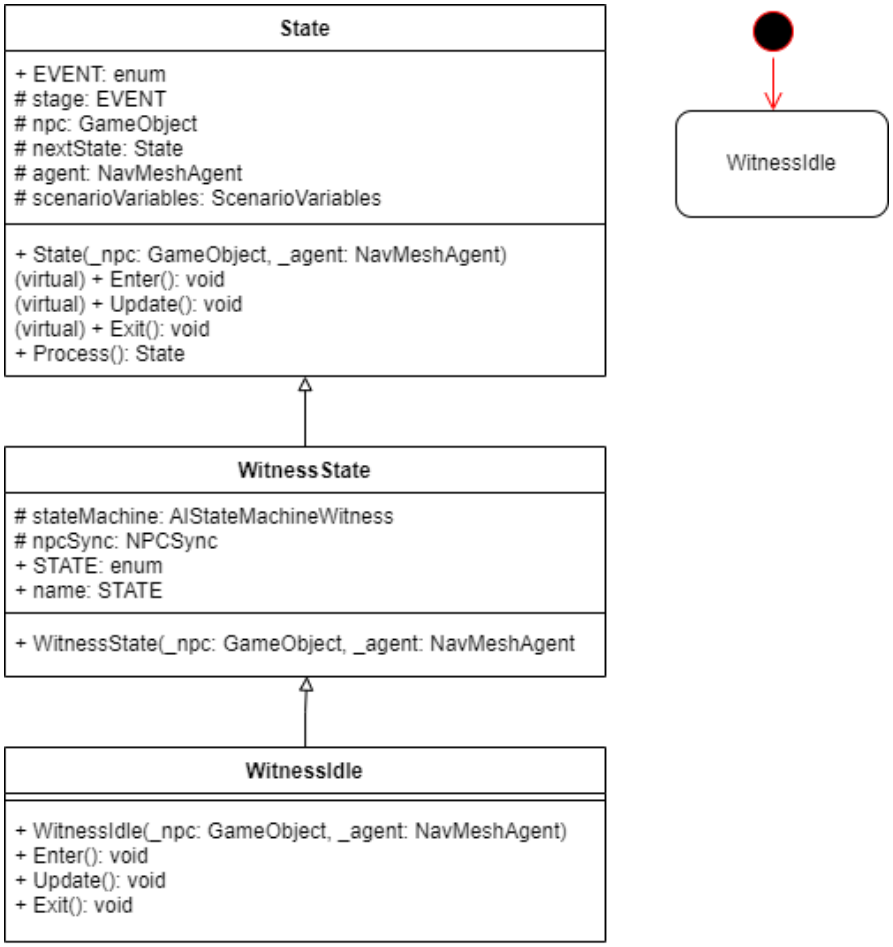


Kuva 8. Tilakoneen kantaluokka State

Kantaluokassa määritellään komponentin tarvitsema Process-metodi, mikä jakaa jokaisen tilan kolmeen vaiheeseen ja siten ohjaa tilan toimintaa. Ilman tämän metodin kutsua komponentissa, tilakone ei toimisi, sillä tilakone ei itse kutsu Process-metodia aloittaakseen toimintansa. Tilakoneen vaiheita vastaa kolme metodia. Enter-metodia kutsutaan, kun tila luodaan. Sen tarkoitus on asettaa tilan lähtötilanne ja tilan tarvitsemat tiedot, kuten viittaukset peliobjekteihin ja komponentteihin, tarvittavien attribuuttien arvot ja muut tilan toiminnan kannalta tärkeät asiat. Kun Enter-metodi on suoritettu, tila siirtyy varsinaiseen tilan toimintaan, josta vastaa Update-metodi. Tila pysyy tässä vaiheessa, kunnes jokin sen siirtymän laukaisevista ehdoista täyttyy. Siirtymän alkaessa tila siirtyy ensin viimeiseen vaiheeseen, jossa on mahdollista siivota tilan tarvitsemat asiat, joita ei enää tarvita, ja alustaa seuraavan tilan tarvitsemia asioita. Kantaluokka tarjoaa nämä kolme tilan vaiheesta vastaavaa metodia virtuaalisina, jolloin jokainen tilaa vastaava luokka voi määritellä niiden toiminnan itsenäisesti.

Witness-tilakone on projektin yksinkertaisin tilakone, koska siihen ei kuulu kuin yksi tila, WitnessIdle. Tila kuvaa hahmon paikallaan seisomista, mikä on yleisin aloitustila projektissa

toteutetuille tilakoneille. Kuva 9 esittää Witness-tilakoneen luokkakaavion sekä toiminnan tilakonekaaviona.



Kuva 9. Witness-tilakone

Kantaluokka State on kaikille tilakoneille yhteinen kantaluokka, mutta se ei sisällä kaikkea tilakoneen tarvitsemaa yhteistä tietoa. Jokainen tilakoneen tila pitää olla tunnistettavissa. Sen sijaan, että tunnistetieto olisi lisätty kaikkien tilakoneiden kantaluokkaan, periytettiin State-luokasta jokaiselle tilakoneelle oma kantaluokka. WitnessState-luokka vastaa Witness-tilakoneelle ominaisista tiedoista, joista tärkeimpinä ovat viittaukset tilakone- ja NPCSync-komponentteihin. Attribuutti name vastasi tilan tunnisteesta, kunnes se siirrettiin NPCSync-komponentille. Attribuutti kuitenkin jätettiin mahdollisten regressio-ongelmien varalta.

Silminnäkijälle luotiin tilakone, sillä yksi tilakoneen vastuista oli animaatioiden kontrolloiminen. Animaatioita ei luokkakaavioissa näy, koska tämä vastuu siirrettiin myöhemmin tilakoneen ulkopuolelle omaksi järjestelmäkseen. Silminnäkijän tilakone jätettiin käyttöön, jos jatkokehityksen aikana sitä tarvittaisiin.

4.1.1 Pelaajan syöte siirtymän laukaisijana

Projektin perusajatuksena on yksiköiden johtaminen ja kommunikaatio. Pelaaja toimii yksikön johtajana, ja hänen pitäisi pystyä jakamaan miehistölle käskyjä. Yksi tekoälyn ensimmäisistä haasteista olikin näiden käskyjen toteutus. Toisin sanoen, miten pelaajan syöte saadaan laukaisemaan tilakoneen siirtymä? Pelaajan antama käsky on yksittäinen tapahtuma, jonka pelaaja välittää käyttöliittymän kautta. Tekoälyn tilakone puolestaan on erillinen järjestelmä, jonka ainoa olemassa oleva linkki Unityyn on tilakonekomponentti. Mahdollisuutena oli kytkeä nämä kaksi toisiinsa, mutta se ei ollut hyvä ajatus. Pelaajan käyttöliittymää ja tekoälyä kehittivät eri ihmiset, ja niiden sitominen toisiinsa suoraan olisi vaikeuttanut kehitystä ja ylläpitoa jatkossa. Ratkaisuksi valittiin tapahtumapohjainen malli.

Pelaajan käyttöliittymä lähettää tapahtumakutsun, ja tilakone kuuntelee niitä. Tämä toimi, muttei riittävän hyvin. Ensimmäisenä ongelmaksi nousi tapahtumakutsujen synkronointi pelaajien välillä. Projektissa oli jo käytössä Photon, joka tuo oman vaihtoehdon Unityn ja C#:n tapahtumakutsuille, ja on sitäkin yksinkertaisempi käyttää. PhotonNetwork.RaiseEvent-metodi kaikessa yksinkertaisuudessaan vaatii vain muutaman määrittelyn, joiden avulla viesti saadaan jaettua kaikkien pelaajien kesken. Kuva 10 esittää metodia, jota käytetään yksikön käskyjen välittämiseen.

```
1  /// <summary>
2  /// AI command event
3  /// </summary>
4  /// <param name="leader">The unit leader</param>
5  /// <param name="order">The command string</param>
6  /// <param name="target">The target position for the command</param>
7  public void SendOrderEvent(string leader, string order, Vector3 target)
8  {
9      object[] content = new object[] { leader, order, target };
10     RaiseEventOptions raiseEventOptions = new RaiseEventOptions
11     {
12         Receivers = ReceiverGroup.All
13     };
14     PhotonNetwork.RaiseEvent(ScenarioVariables.CBSafeSimulationAIEvent,
15         content, raiseEventOptions, SendOptions.SendReliable);
16 }
```

Kuva 10. SendOrderEvent-metodi yksikön johtamiseen

PhotonNetwork.RaiseEvent-metodin parametrit ovat

- tapahtumakutsun koodi: byte-tyyppinen luku välillä 0–199

- viesti: yleensä object-taulukko, joka sisältää jokaisen lähetettävän datan
- RaiseEvent-asetukset: voidaan määritellä vastaanottajat ja tallennetaanko tapahtumakutsu välimuistiin
- lähetysasetukset: lähetystapa ja salauksen valinta.

Tapahtumakutsun lähettäminen on vasta ensimmäinen puoli tapahtumasta. Sille on määriteltävä kuuntelija (event listener). Tekoälyn toteutuksessa erityyppiset hahmot jaettiin omiin tilakoneisiinsa. Pelaaja on ainoa, joka tarvitsee mahdollisuuden antaa komentoja tekoälyn kontrolloimille hahmoille, joten käskyjen kuuntelija lisättiin aluksi FirefighterState-luokkaan.

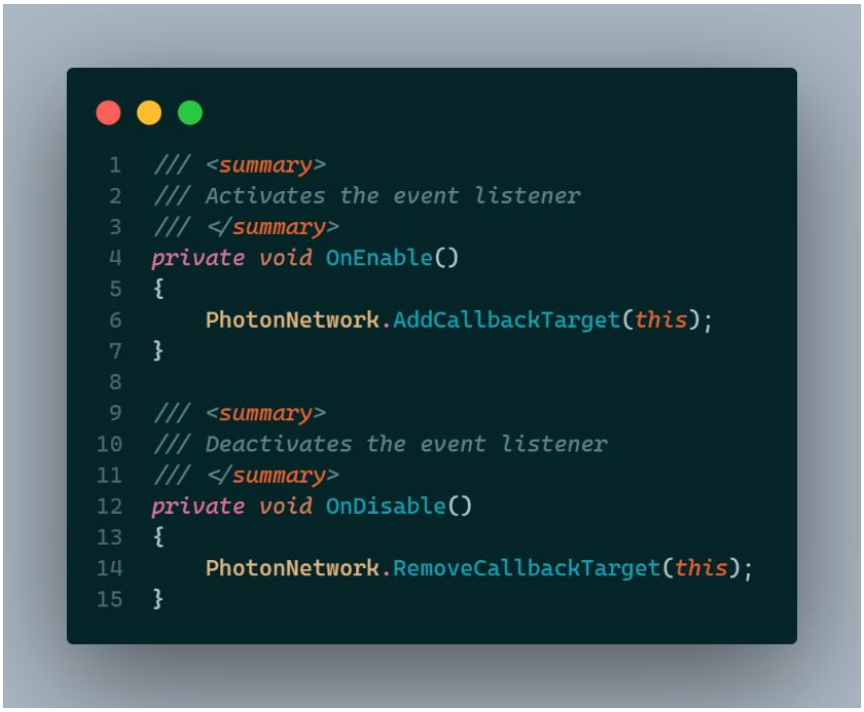
Tapahtumakutsun kuuntelijan on toteutettava IOnEventCallback-rajapinta ja määritettävä rajapinnan OnEvent-metodin toiminta. Kuva 11 esittää käskytapahtuman kuuntelijan toteutuksen.

```
/// <summary>
/// The event listener
/// </summary>
/// <param name="_photonEvent">EventData</param>
public void OnEvent(EventData _photonEvent)
{
    byte eventCode = _photonEvent.Code;

    if (eventCode == ScenarioVariables.CBSafeSimulationAIEvent)
    {
        if (gameObject.GetComponent<IsABoat>() &&
            gameObject.GetComponent<IsABoat>().BoatType == IsABoat.BOATTYPE.DC)
        {
            return;
        }

        object[] data = (object[])_photonEvent.CustomData;

        Leader ??= (string)data[0]; // If leader is null, set leader by event data

        if (Leader == (string)data[0])
        {
            if (SetTask((string)data[1]) == TASK.SURFACE ||
                (SetTask((string)data[1]) == TASK.SEARCH &&
                SetSearchDefinition((string)data[1]) == SEARCHPATTERN.SECTOR))
            {
                if (!UtilNavigation.InWater((Vector3)data[2]))
                {
                    aiEvent.SendNPCResponseEvent("Order", "Event handler"
                        , "Invalid target");
                    return;
                }
            }

            Order = (string)data[1];
            Target = (Vector3)data[2];

            aiEvent.SendLogEvent(Leader, gameObject.name, Order + " " + Target);

            if (Order == "Drive")
            {
                CarStatus = CAR.DRIVE;
                CarTarget = Target;
            }

            GivenTask = SetTask(Order);
            GivenMethod = SetMethod(Order);
            GivenSearchPattern = SetSearchDefinition(Order);
            GivenShorelineDirection = SetShorelineDirection(Order);
            GivenSectorDirection = SetSectorDirection(Order);
        }
        else
        {
            Debug.Log("Invalid leader information: " + (string)data[0]);
        }
    }
}
```

Kuva 11. Käskytapahtuman kuuntelija

Ensimmäisenä verrataan tapahtumakutsun koodin avulla, onko tapahtumakutsu osoitettu tälle kuuntelijalle. Tapahtumakutsun kuljettama koodi on sijoitettu EventData.Code-attribuuttiin Jos näin on, voidaan tapahtumakutsun sisältämä data lukea EventData.CustomData-attribuutista. Tässä tärkeintä on muistaa varmistaa datan tyyppi, sillä Photon ei sitä tietoa välitä. Kun data on luettu, se voidaan käsitellä. Käskytapahtuman kohdalla varmistetaan, että komento on oikea, ja sen mukana tulleen tiedon perusteella annettu käsky jaetaan useampiin osiin. Jokainen tapahtumakuuntelija lähettää oman tapahtumakutsun, jonka avulla tieto saapuneesta tapahtumakutsusta voidaan tallentaa lokiin.

Tässä vaiheessa pelaajan käyttöliittymä ja tekoälyn siirtymien laukaisijat voidaan yhdistää toisiinsa. Tilakoneen tarvitsee vain tarkkailla komennon perusteella tallennettua tietoa ja aloittaa siirtymä sen perusteella. Tapahtumakutsun kuuntelija tarvitsee vielä yhden tärkeän lisäyksen toimiakseen. Se on rekisteröitävä, jotta sitä voidaan käyttää. Rekisteröimiseen on kaksi vaihtoehtoa, jotka todellisuudessa ovat täysin sama asia. Kuva 12 näyttää tapahtumakutsun kuuntelijan rekisteröimisen ”callback”-metodina. Vaihtoehtoinen tapa olisi rekisteröidä tapahtumakutsun kuuntelija PhotonNetwork.NetworkingClientille. Tällöin luokan ei tarvitse toteuttaa IOnEventCallback-rajapintaa.



```
1  /// <summary>
2  /// Activates the event listener
3  /// </summary>
4  private void OnEnable()
5  {
6      PhotonNetwork.AddCallbackTarget(this);
7  }
8
9  /// <summary>
10 /// Deactivates the event listener
11 /// </summary>
12 private void OnDisable()
13 {
14     PhotonNetwork.RemoveCallbackTarget(this);
15 }
```

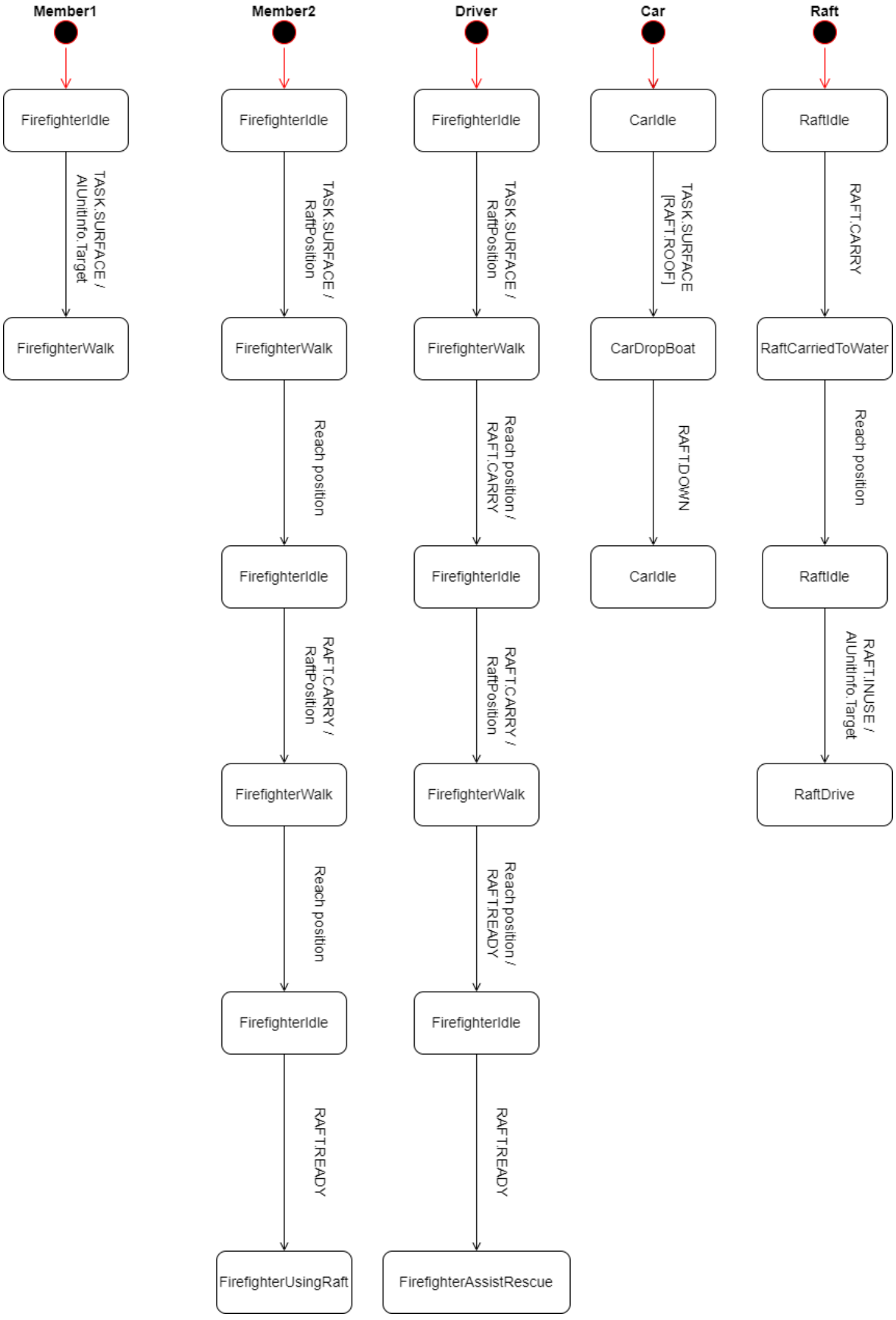
Kuva 12. Kuuntelijan rekisteröiminen

Tapahtumapohjainen malli toimii hyvin liittämään pelaajan käyttöliittymän ja tekoälyn ohjaamisen toisiinsa, mutta tapahtumakutsun kuuntelijan sijoittaminen itse tilakoneeseen ei ole paras vaihtoehto. Kävi nopeasti ilmi, että pelaaja ei komenna vain miehistöä, vaan myös yksikköä kokonaisuutena. Miehistö oli jo olemassa, mutta varsinainen yksikkö ei ollut kuin

ajoneuvo. Ajoneuvo vaatisi oman tilakoneen, jonka pitäisi myös kuunnella pelaajan antamina käskyjä. Käskyjä, jotka annetaan koko yksikölle.

Yksikölle luotiin oma komponentti AIUnitInfo, joka liitettiin ajoneuvoon. Komponentin tarkoitus oli sisältää kaikki yksikön yhteinen data, mukaan lukien käskyjen tapahtumakutsun kuuntelija. Käskyjen sisällön siirtäminen hahmojen tilakoneen ulkopuolelle avasi mahdollisuuden teettää yksikön jäsenillä eri toimintoja, saattaa heidät eri tilaan toistensa kanssa. Esimerkiksi pintapelastuskäskyn aikana kaikki kolme yksikön jäsentä ovat vain hetken samassa tilassa, ennen kuin he alkavat tekemään eri asioita. Yksi pelastajista lähtee veteen uiden, toinen ottaa kuljettajan kanssa lautan alas ja lähtee sen jälkeen lautalla ensimmäisen perään. Tämänlaisen toiminnan toteuttaminen olisi ollut huomattavasi haastavampaa, jos jokainen tilakone olisi toiminut itsenäisesti.

Lisähaasteena edellä mainitussa tilanteessa on se, että pelastajien AIStateMachineFirefighter ei ole ainoa tilakone, joka vastaa pintapelastustoiminnasta. Yksikön ajoneuvolla ja lautalla on omat tilakoneensa: AIStateMachineCar ja AIStateMachineRaft. Yksi käsky, joka sisältää vain yksikön tunnisteen, käskysanan ”Surface” ja käskylle annetun sijainnin vedessä saa miehistön, ajoneuvon ja lautan toimimaan yhdessä Kuva 13 mukaisesti.



Kuva 13. Pintapelastuksen aloittaminen tilakonekaaviona

Isolla kirjoitetut TASK ja RAFT ovat yksikön ja sen osien tilasta kertovia statusmuuttujia, jotka on sijoitettu AIUnitInfo-komponenttiin. Kuvaajassa oletetaan, että pintapelastuskäsky on tullut ennen yksikön saapumista, jolloin ensimmäinen ja toinen pelastaja ovat valmiiksi uimavarustuksessa.

Miehistön ensimmäinen pelastaja aloittaa FirefighterIdle-tilasta. Hän siirtyy FirefighterWalk-tilaan, koska laukaisija ”TASK.SURFACE” on asetettu. Hän saa tilakoneensa kohteeksi yksikölle annetun sijainnin ja kulkee sitä kohden.

Miehistön toinen pelastaja aloittaa samasta tilasta kuin ensimmäinen, koska FirefighterIdle-tila on määritelty AIStateMachineFirefighter-komponentissa pelastajien aloitustilaksi. Sama laukaisija ”TASK.SURFACE” johtaa myös toisen pelastajan FirefighterWalk-tilaan, mutta asettaa tälle eri kohteen. Toinen pelastaja saa kohteekseen sijainnin lautan vierestä. Lautta on piilossa ajoneuvon takana ja katolla oleva kopio on visuaalinen harhautus. Kohteeseen saavuttuaan, hän siirtyy FirefighterIdle-tilaan odottamaan seuraavaa laukaisijaa, joka on ”RAFT.CARRY”.

Samanaikaisesti kuljettaja seuraa toisen pelastajan toiminnan mukaisesti. Ainoana erona tähän mennessä on hänelle annettu kohde, joka on heti ajoneuvon takana. Hän siirtyy FirefighterIdle-tilaan jääden odottamaan laukaisijaa ”RAFT.DOWN”, jonka jälkeen hän siirtyy lautan viereen FirefighterWalk-tilan kautta, vastakkaiselle puolelle toisesta pelastajasta.

Kuljettaja asettaa ”RAFT.CARRY” statuksen yksikölle, joka laukaisee lautan tilakoneen ensimmäisen siirtymän. Lautta siirtyy RaftIdle-aloitustilasta RaftCarriedToWater-tilaan. ”RAFT.CARRY” siirtää toisen pelastajan ja kuljettajan jälleen FirefighterWalk-tilaan, tällä kertaa kohteinaan lautan vierellä olevat sijainnit. Kun lautta saavuttaa rantaviivan, se siirtyy takaisin RaftIdle-tilaan ja asettaa ”RAFT.READY” statuksen. Tämä kertoo toiselle pelastajalle ja kuljettajalle seuraavan siirtymän, joka on takaisin FirefighterIdle-tilaan.

FirefighterIdle-tilasta ”TASK.SURFACE” ja ”RAFT.READY” siirtävät kuljettajan FirefighterAssistRescue-tilaan, jossa kuljettaja siirtyy rannalla yksikölle annettua sijaintia lähimpänä olevaan paikkaan ja jää odottamaan seuraavaa laukaisijaa. Toinen pelastaja puolestaan siirtyy FirefighterUsingRaft-tilaan samoilla laukaisijoilla. Tässä tilassa toinen pelastaja piiloutuu näkyvistä, mutta jatkaa lautan vieressä olevan kohteen seuraamista, mutta vasta, kun hän on asettanut ”RAFT.INUSE” statuksen. Kyseinen status on lautalle merkki siitä, että se siirtyy RaftDrive-tilaan, jossa lautta saa yksikölle annetun kohteen ja lähtee kulkemaan sitä kohden.

Kaikki yksikön hahmot ovat nyt päässeet siihen tilaan, joka toimii heidän perustilanaan pintapelastuksen aikana. Seuraava tila voi olla pelastuksen aloittaminen tai tehtävän lopettaminen suoritettuna.

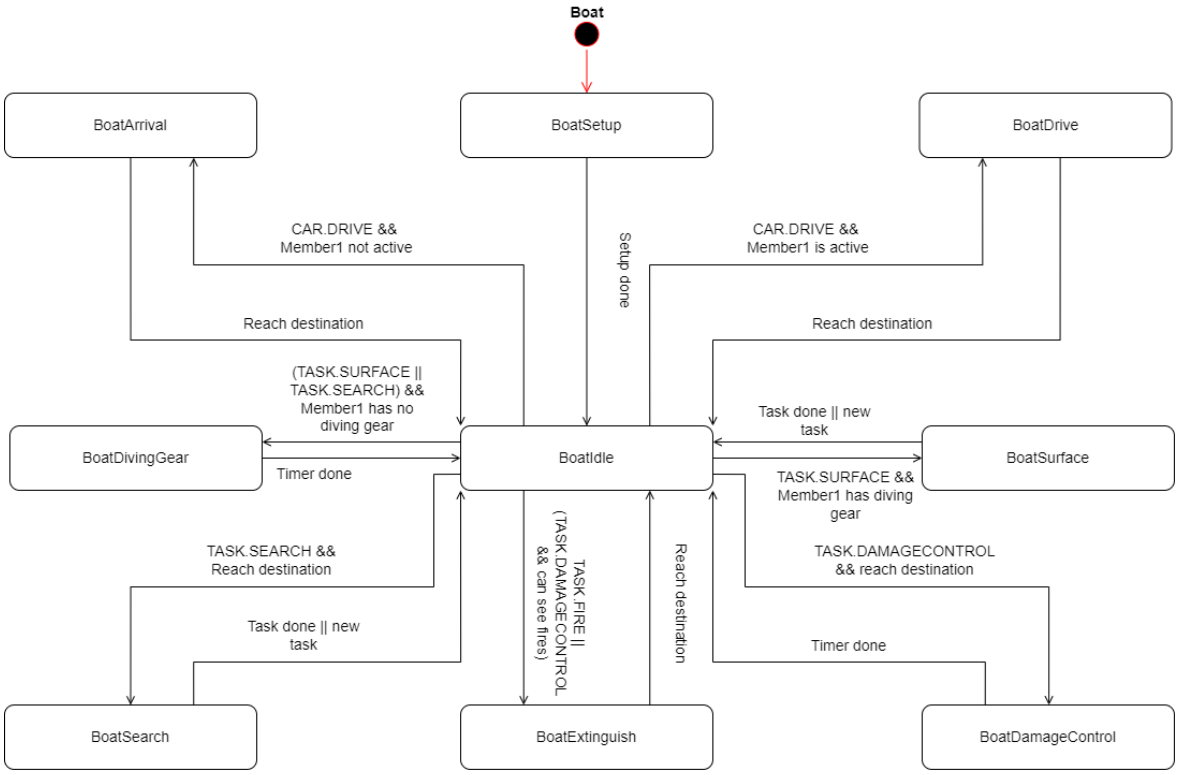
4.1.2 Useita tilakoneita eri tarkoituksiin

Projektissa tekoälyn liikuttamia hahmoja on useita, ja jokainen niistä toimii eri tavalla, siksi onkin helpompaa, jos jokaiselle hahmotyypille luodaan oma tilakone. Vaikka kaikkien erilaisten hahmojen kontrolloiminen olisi mahdollista yhdellä tilakoneella, se ei ole tarkoituksenmukaista. Eri hahmotyypit jakavat paljon samoja toimintoja, mutta yksikään niistä ei toimi samalla tavalla. Tilakoneiden jakaminen hahmotyypin perusteella helpottaa niiden luomista ja ylläpitoa. Tilakoneiden suunnittelua helpottaa myös tilojen pitäminen yksinkertaisina. On helppoa lisätä uusi tila, kun sitä tarvitaan, mutta uuden toiminnan lisääminen olemassa olevaan tilaan on huomattavasti vaikeampaa. Esimerkkinä käytetty PacMan-peli kertoo samaa tarinaa. Yksinkertaisella tavalla voi saavuttaa monipuolisen tuloksen, jos sen toteuttaa hyvin. Tämä kävi erittäin selkeäksi kehitystyön aikana. Ongelmien löytäminen ja ratkaiseminen osoittautui helpommaksi, jos tilojen toiminnan piti yksinkertaisena ja kasvatti tilojen määrää. Samalla kävi ilmi, että käytetyistä kehitystyökaluista ei ollut apua toteutetun tilakoneen ongelmien löytämiseen. Tämä vain korosti edellä mainittua hyötyä yksinkertaisista tiloista, sillä ainoa tapa löytää tilakoneesta tai tiloista ongelmia, oli konsoliin kirjoittaminen tilan aikana.

Aikaisemmin tilakoneista esiteltiin Witness-tilakone ja edellisessä kappaleessa kolme lisää. Yhteensä eri tilakoneita luotiin yhdeksän, joista seitsemää pelaajat voivat ohjailla antamalla ryhmälleen käskyjä. Kaikkia tilakoneita yhdistää se, että niiden perustila kuvaa toimetonta, odottavaa tilannetta. Siksi jokaisen tilan perustila on kaiken toiminnan keskiössä. Aina, kun toiminta on suoritettu loppuun, palataan perustilaan odottamaan uuden toiminnan aloittamista. Perustiloja yhdistää myös se, että siirtymien alkaessa, ne määrittelevät toiminnan kohteen tilakoneelle, mutta eivät itsessään sisällä mitään toimintaa.

AIStateMachineBoat

AIStateMachineBoat-tilakone ohjaa pelastajien venettä. Veneellä ei ole paikasta toiseen liikkumisen lisäksi varsinaista toimintaa, mutta toteutuksessa eri tehtävät eroteltiin omiksi tiloiksi. Kuva 14 näkyy, miten jokainen tila johtaa takaisin BoatIdle-tilaan, josta seuraava toiminta alkaa.

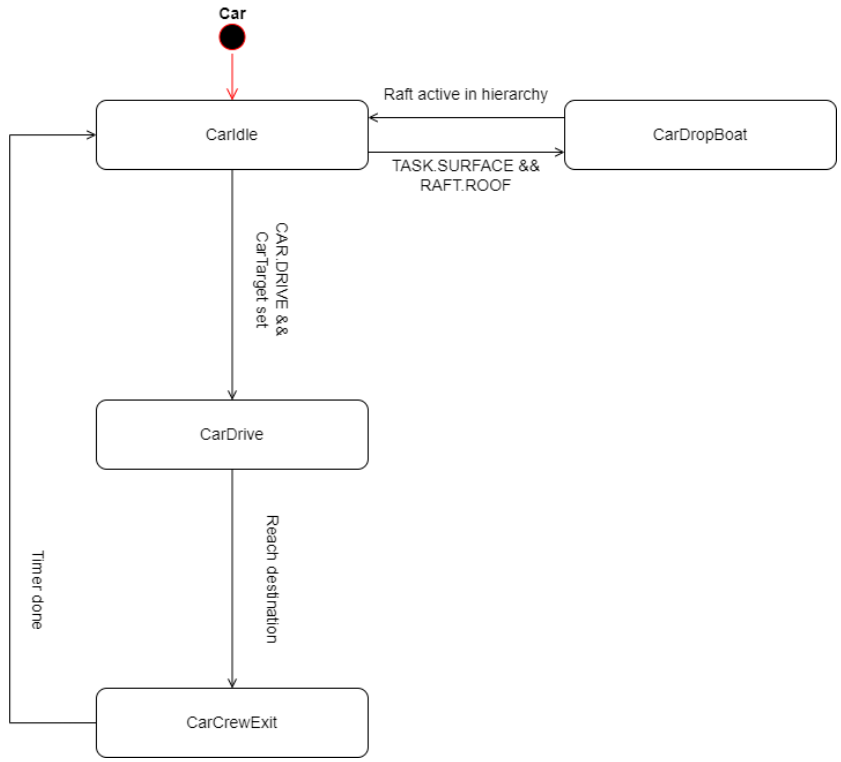


Kuva 14. AIStateMachineBoat-tilakonekaavio

Tehtävien erotteleminen mahdollisti veneen miehistön kontrolloimisen veneen kautta. Miehistö aloittaa toimintansa vain, kun vene on oikeassa paikassa ja valmiina toiminnan aloittamiselle. Miehistö ei voi toteutuksessa liikkua veneen kannella, joten vene siirtää miehistön jäsenet oikeisiin paikkoihin toiminnan mukaisesti. Veneen tilakone ei varsinaisesti yhdistä kahta tilakonetta, vaikka veneitä on kahta eri tyyppiä. Veneen tilakone kuitenkin huomioi veneen tyypin ja käyttäytyy sen mukaisesti. SAR-vene on pelaajan kontrolloitavissa, mutta vahingontorjuntavene toteutettiin autonomisena. Syynä oli se, että tilakonetta luodessa vahingontorjuntaveneellä ei ollut kuin yksi tehtävä: Öljyvuotojen torjunta. Myöhemmin veneelle lisättiin sammutusmahdollisuus ja öljyvuotoihin lisättiin mahdollisuus syttyä tuleen, joten vahingontorjuntaveneen muuttaminen pelaajan kontrolloitavaksi on yksi jatkokehitysmahdollisuuksista.

AIStateMachineCar

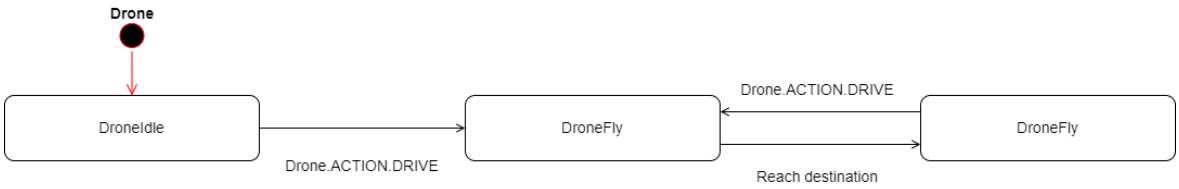
Paloautojen AIStateMachineCar-tilakone on varsin yksinkertainen. Ajaminen saapumispaikasta kohteeseen on sen ensisijainen tehtävä, ja suurimman osan ajasta tilakone viettääkin CarIdle-tilassa. Paikalle saapumisen lisäksi autojen tilakone vastaa lautan laskemisesta ja siitä, että miehistö nousee autosta. Tilakonekaavio on nähtävissä Kuva 15.



Kuva 15. AIStateMachineCar-tilakonekaavio

AIStateMachineDrone

AIStateMachineDrone kontrolloi dronea. Drone osaa kolme eri asiaa, joten sillä on vain kolme eri tilaa, jotka on esitetty Kuva 16.



Kuva 16. AIStateMachineDrone-tilakonekaavio

Drone aloittaa paikoiltaan DroneIdle-tilassa, josta se lähtee lentoon, kun sitä kontrolloiva johtoauton kuljettaja on valmis. Drone lentää joko annettuun sijaintiin tai onnettomuuspaikkaa kohden. Saapuessaan määränpäähän, drone jää leijumaan paikoilleen. Leijumistilasta drone voidaan ohjata toiseen sijaintiin, jos johtoauton kuljettaja on valmiina. Dronen tilakone ei kontrolloi sen kameraa, vaan kameraa ohjataan omalla komponentilla, joka suuntaa kameran annettuun sijaintiin tai onnettomuuspaikkaa kohden. Dronelle toteutettiin oma NavMeshAgent sekä oma NavMeshSurface, jotta sen lentäminen saatiin toteutettua.

AIStateMachineFirefighter

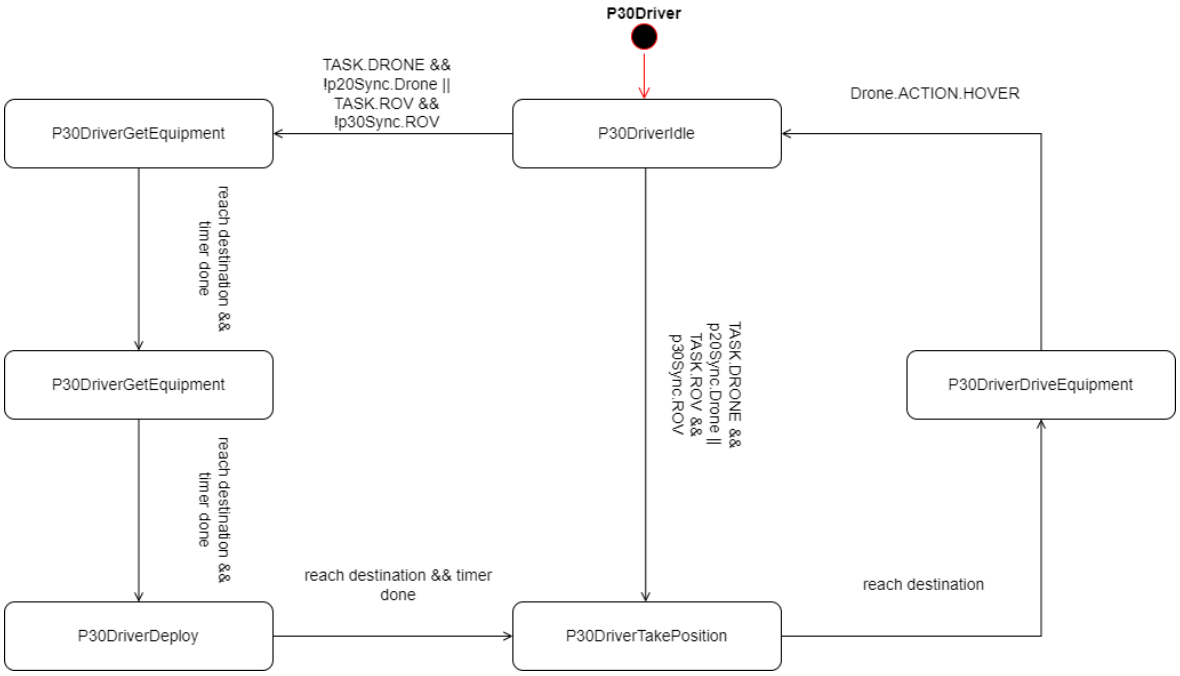
Pelastajien AIStateMachineFirefighter-tilakone on pikemminkin kaksi tilakonetta yhdessä. Rannalta toimivilla pelastajilla on omat tilansa, kuten on veneessä toimivilla pelastajilla. Tiloja ei kuitenkaan eroteltu kahdeksi tilakoneeksi, sillä ne molemmat tarvitsevat samat tiedot. Erottelu hoidettiin siten, että ennen Idle-tilaa lisättiin uusi tila, jonka tehtävänä on tarkistaa kumpaa tyyppiä hahmo edustaa ja siirtää tämä sitä vastaavaan aloitustilaan.

Veneessä toimivien pelastajien tilakone on näistä kahdesta yksinkertaisempi. Miehistön kolmesta jäsenestä vain ensimmäinen pelastaja toteuttaa tämän tilakoneen. Vene itse huolehtii toisesta pelastajasta ja kuljettajasta. Käytännössä tosin, toinen pelastaja ja kuljettaja siirtyvät FirefighterBoatIdle-tilaan, jossa he pysyvät loputtomiin. Tämä mahdollistaa heidän animaatioiden kontrolloimisen kaikkien hahmojen yhteisellä kontrollerilla. Ensimmäinen pelastaja puolestaan toimii vain pintapelastuksen sekä etsinnän aikana.

Pelastajat, jotka toimivat rannalta, puolestaan käyttävät projektin laajinta tilakonetta toimintansa ohjaajana. Tämä on luonnollista, sillä heidän vastuullaan on suurin osa simulaation toiminnasta. Nämä pelastajat osaavat pintapelastuksen yhteistyössä lautan kanssa, he osaavat kaksi pinnan alta -etsintämallia, he kykenevät sammuttamaan tulipaloja, ja heille aloitettiin vahingontorjunnan toteutus, mikä myöhemmin jätettiin jatkokehitysmahdollisuuksiin.

AIStateMachineP30Driver

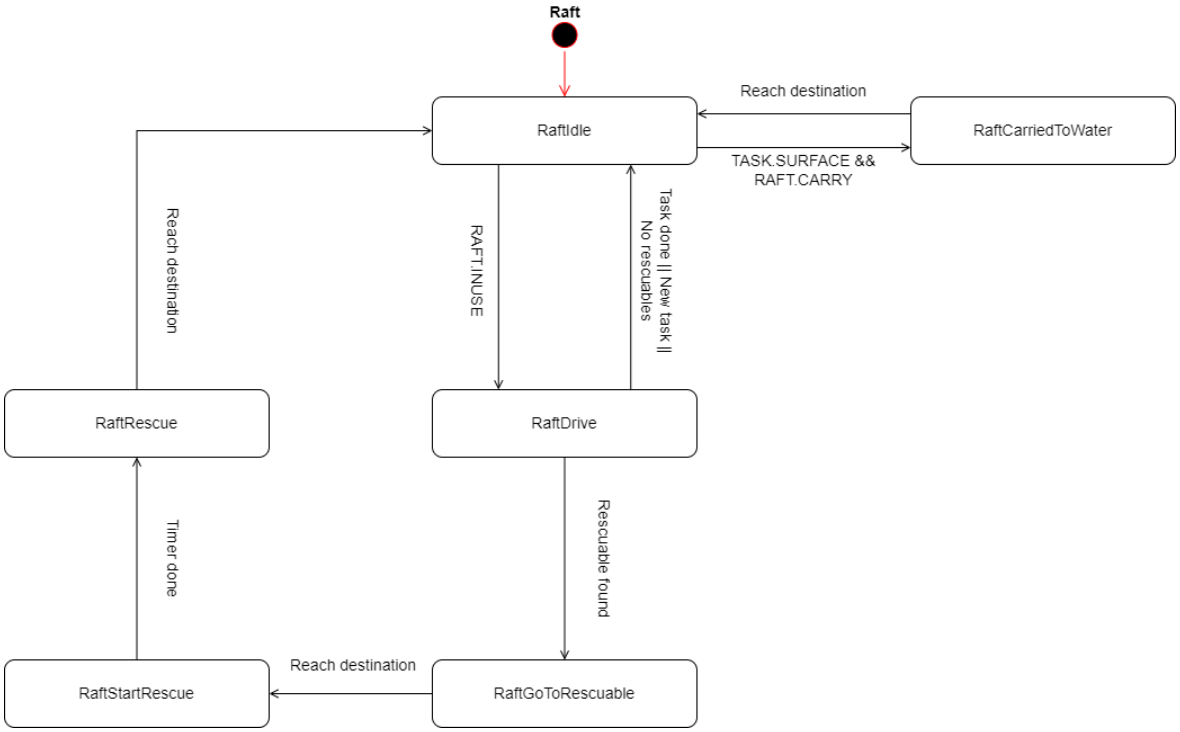
AIStateMachineP30Driver-tilakone ohjaa johtoauton kuljettajaa, joka toimii niin dronen kuin ROV-sukellusveneen ohjaajana. Seuraava kuva (Kuva 17) esittää johtoauton kuljettajan tilakoneen toimintaa. Käskyn saatuaan, hän noutaa laitteen johtoauton takaosasta ja laskee sen joko lähelleen maahan tai lähimpään sijaintiin vedessä. Hän siirtyy ohjaamista varten johtoauton takaosan näytön lähelle ja ohjaa laitteen perille. Tämän jälkeen hän jää odottamaan uutta käskyä. Kuljettaja voi myös siirtää käytössä olevia laitteita käskyn perusteella, jos hän on vapaa ja laite on leijuntatilassa.



Kuva 17. AIStateMachineP30Driver-tilakonekaavio

AIStateMachineRaft

Lauttaa kontrolloi AIStateMachineRaft-tilakone. Suurimmilta osin lautta käyttäytyy kuin pintapelastaja, joka ui. Lautan tilakonekaavio on nähtävissä Kuva 18.

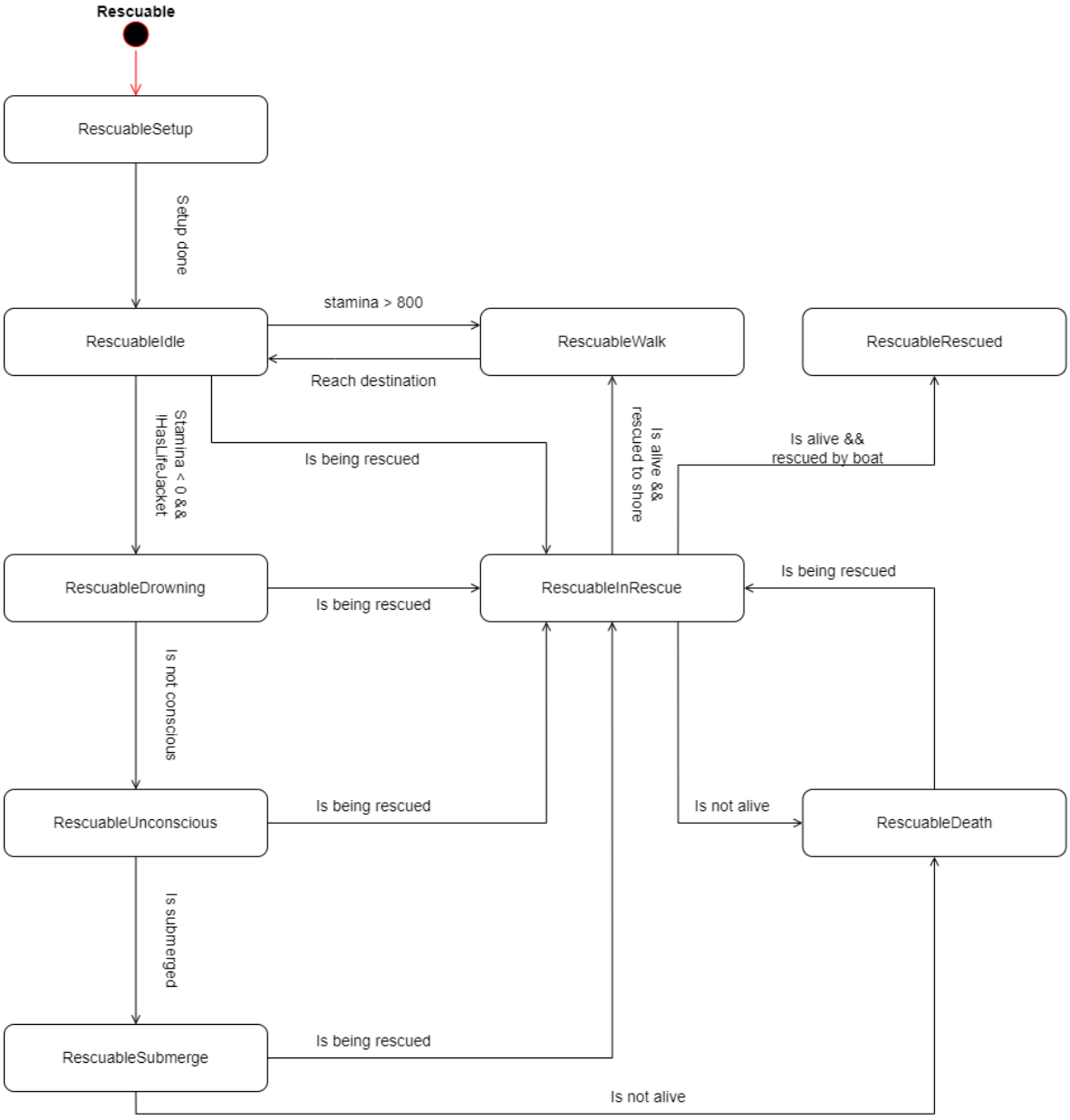


Kuva 18. AIStateMachineRaft-tilakonekaavio

Se liikkuu käskyn mukaiseen sijaintiin ja etsii pelastettavan, jonka pyrkii pelastamaan. Erona kuitenkin on se, että lautta aktivoituu vasta, kun se on laskettu ajoneuvon katolta alas ja siirtyy itse rannalle. Lautalle toteutettiin myös etsintä pinnan alta, mikä myöhemmin poistettiin. Veneen miehistö kuitenkin käyttää edelleen lautalle luotua etsintämallia.

AIStateMachineRescuable

Pelastettavien AIStateMachineRescuable-tilakone on muista tilakoneista erillinen tilakone. Se käyttää pelastettavan statusta ohjaajanaan ja pysähtyy vain, jos pelastettavaa pelastetaan. Kuva 19 esittää pelastettavan tilakoneen toimintaa.



Kuva 19. AIStateMachineRescuable-tilakonekaavio

Pelastettavan statuksen perusarvona toimii hänen jäljellä oleva jaksaminen, stamina, joka kuvataan sekunteina. Stamina kuluu jatkuvasti, kunnes pelastettava on joko pelastettu tai menehtynyt. Stamina määrittelee pelastettavan tilan, kun häntä ei olla pelastamassa. Tila alkaa vedessä kellumisesta ja siirtyy siitä hukkumiseen, tajunnan menetykseen ja pinnan alle vajoamiseen päättyen menehtymiseen. Staminan kulumisnopeuteen vaikuttavat pelastettavan uimataito ja paniikki. Paniikki on satunnaistettu ja sen mahdollisuus kasvaa mitä vähemmän staminaa pelastettavalla on jäljellä.

Pelastettavien tilakone on toteutetuista tilakoneista keskeneräisin, vaikkakin toteutukseltaan täydellisin. Syynä on simulaation toteutuksesta puuttuva ensihoito. Pelastamisen tai pelastautumisen jälkeen pelastettavat joko siirtyvät ensihoitopaikalle tai jäävät elottomana makaamaan rannalle. Ensihoidon lisääminen simulaatioon on siten erinomainen jatkokehitysehdokas.

AIStateMachineROV

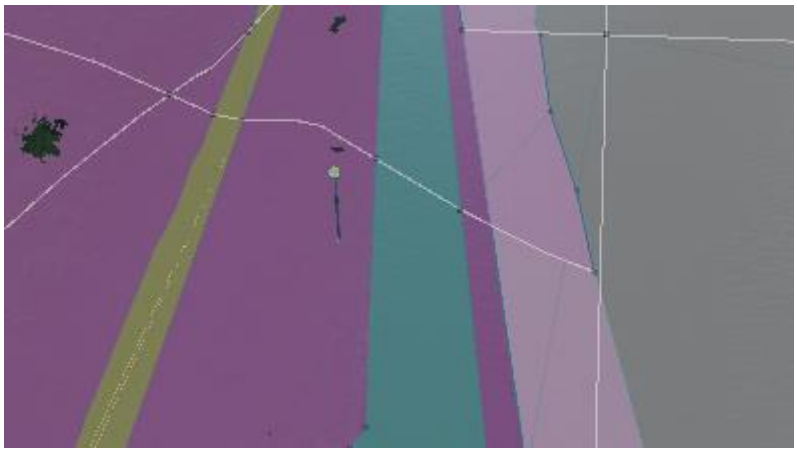
AIStateMachineROV-tilakone on pitkälti sama kuin dronella. Erona on luonnollisesti eri liikkumistapa. ROV-sukellusvene ui määränpäähänsä veden pinnalla ja sukeltaa vasta sen saavutettuaan. Jos ROV saa uuden määränpään, se aloittaa nousemalla pintaan ja liikkuu vasta pinnalla. Kuten dronella, ROV:n kameralla on oma kontrollerinsa, tosin ROV käyttää samaa NavMeshSurfacea muiden agenttien kanssa. ROV:n agentti on rajoitettu vain Shore- ja Water-aluetyypeille.

AIStateMachineWitness

Silminnäkijän AIStateMachineWitness-tilakonetta käytettiin aikaisemmin esimerkkinä. Tilakone sisältää vain yhden tilan, jonka aikana silminnäkijä seisoo paikallaan. Tilakonetta voidaan laajentaa tulevaisuudessa.

4.2 Navigointi

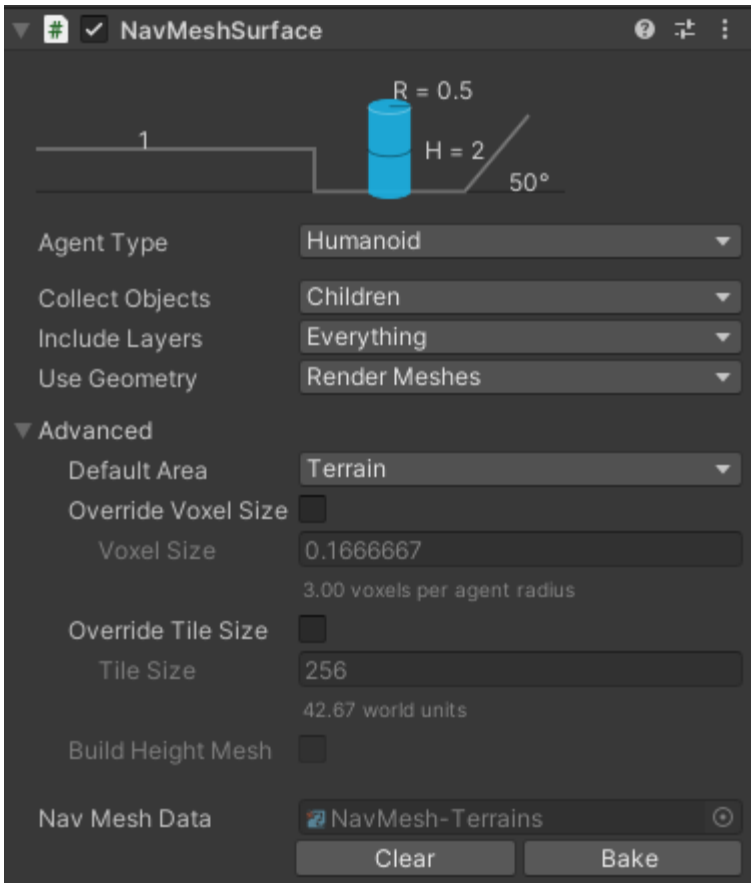
Unity tarjoaa sisäänrakennetun työkalun navigointiverkon luomiseen. Tämä työkalu ei ollut riittävän monipuolinen projektin tarpeisiin nähden. Unity ylläpitää NavMeshComponents-työkalulaajennusta, joka on erikseen ladattavissa heidän GitHub-sivultaan. Laajennus tarjoaa työkaluja, joiden avulla navigointiverkon luominen on helpompaa editorissa. Lisäksi on mahdollista luoda useita navigointiverkkoja sekä alueita. Kuva 20 esittää, miltä valmis navigointiverkko näyttää editorissa.



Kuva 20. NavMesh

Kuvassa maaston päälle on luotu värien avulla havaittava navigointiverkko, jossa jokainen väri kuvastaa eri aluetyyppiä. Violetti vastaa maastoa, jossa voi vapaasti kävellä. Tie on merkitty keltaisena ja ranta vihreänä. Vaaleampi raita vihreän oikealla puolella on veden pinnalla, ja sen alta heijastuu joen pohjalla oleva alue, jolla ei voi kulkea.

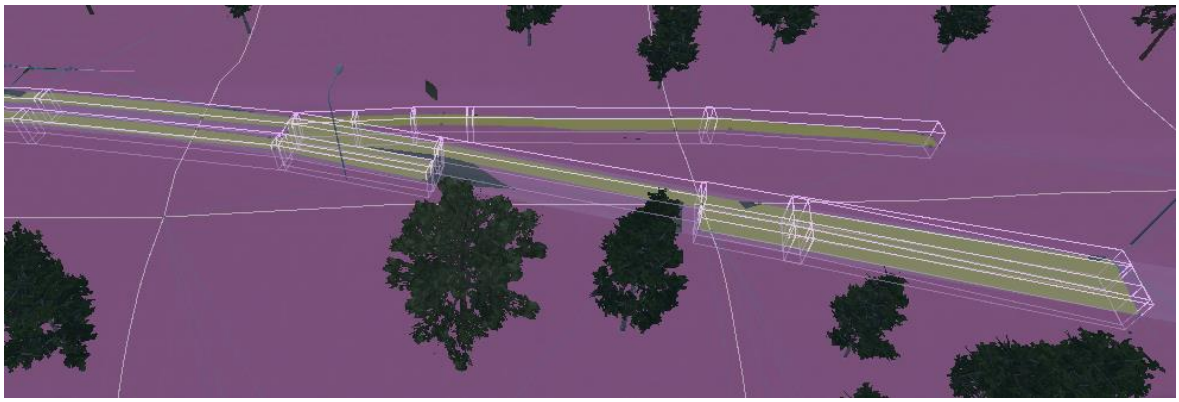
NavMeshComponents-työkalujen avulla navigointiverkon määrittely on mahdollista luomalla pintoja, jotka sisältävät halutun alueen navigointiverkon. Tämä tapahtuu lisäämällä NavMeshSurface-komponentti haluttuun peliobjektiin. NavMeshSurface korvaa Unityn sisäänrakennetun navigointiverkon luomiseen käytetyn Navigation-työkalun. Kuva 21 näyttää tämän komponentin mahdolliset määrittelyt.



Kuva 21. NavMeshSurface

Projektissa toteutettiin kolme NavMeshSurfacea, joista ensimmäinen kattaa kaiken maa-alueen. Vesi muodostaa toisen NavMeshSurfacen. Nämä eroteltiin toisistaan, jotta hahmot voivat liikkua veden pintaa pitkin ja sukeltamisen toteutus helpottui. Kolmas NavMeshSurface luotiin dronea varten, ja se on tasainen yhtenäinen alue, joka kattaa koko luodun maailman.

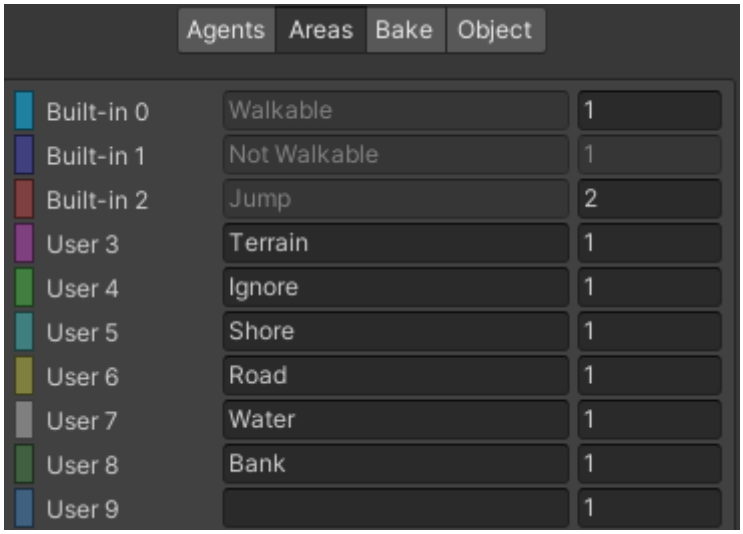
Peliobjekti, johon NavMeshSurface on liitetty, määrittelee sen alueen, jolle navigointiverkko voidaan luoda. Tämä alue kuvaisi NavMeshSurfacessa määriteltyä aluetyyppiä. NavMeshSurfacen kattamaa aluetta voidaan jakaa pienempiin osiin käyttämällä NavMeshModifierVolume-työkalua. Kyseessä on komponentti, jonka avulla NavMeshSurfacelle voidaan lisätä muita aluetyyppejä haluttuihin paikkoihin. Työkalu toimii siten, että se liitetään toiseen objektiin, joka voi olla tyhjäkin, ja sille määritellään aluetyyppi. Kun navigointiverkko luodaan, tämän objektin alue korvaa aluetyypin sen muodon osoittamalla alueella. Kuva 22 esittää keltaisten tiealueiden luomista NavMeshModifierVolume-objekteilla.



Kuva 22. NavMesh Modifier Volume

NavMeshModifierVolume-komponentti on liitetty suorakulmaiseen särmiöön, ja sille on määritelty oma aluetyyppinsä. Muuttamalla tämän särmiön sijaintia ja kokoa, voidaan muuttaa sen ”maalaaman” alueen tyyppiä valmiissa navigointiverkossa.

Aluetyyppien määrittely onnistuu Unityn sisäänrakennetun Navigation-työkalun kautta. Kolme ensimmäistä aluetyyppiä on lukittu, ja niitä ei voi muuttaa, mutta uusien aluetyyppien lisääminen on mahdollista tyhjien rivien kautta. Seuraavassa kuvassa (Kuva 23) on esitetty projektissa käytetyt aluetyypit. Aluetyypeille on myös mahdollista määritellä kerroin, jota käytetään reittien laskemiseen. Tätä ominaisuutta ei projektissa hyödynnetty, sillä tekoäly kontrolloi hahmojen liikkumista tarkoin kohteiden määrittelyn kautta.



Kuva 23. Määritetyt NavMesh-aluetyypit

NavMeshSurfacet ovat erillisiä alueita, joten liikkuminen niiden välillä ei ole suoraan mahdollista. NavMeshComponents tarjoaa alueiden välisen liikkumisen mahdollistavan työkalun, NavMeshLink. Linkkien lisääminen toteutettiin skriptillä, joka ajetaan pelin alkaessa. Linkkejä ei tarvita jokaisessa skenaariossa, joten niiden luominen vain tarvittaessa on

perusteltua. Linkki toimii siten, että hahmon saapuessa NavMeshSurfacen reunalle, hahmo siirtyy linkkiä pitkin toiselle NavMeshSurfacelle. Tämä ei ole sulava siirtyminen, vaan teleport-tyyppinen siirtymä. Linkkien käyttö oli kuitenkin helpompi toteuttaa, kuin rampin luominen rannalta veden pinnalle ja veden pinnan käyttäminen navigointiverkon osana. Suurin osa eri ratkaisuyrityksistä käytti joko joen pohjaa hahmojen kulkemana pintana tai ei antanut määritellä ranta-aluetta veden pinnalle. Ranta-alue on vahvasti käytössä tekoälyn ohjaajana, joten sen pois jättäminen olisi ollut työlästä.

Navigointiverkon luominen toimii pohjana hahmojen liikkumiselle. Varsinaisesta liikkumisesta huolehtii NavMeshAgent-komponentti, joka liitetään jokaisen hahmon peliobjektiin. Komponentin lisääminen muuttaa hahmon agentiksi. Agentille voidaan määritellä kohde, joka on sijainti navigointiverkolla. Kohteen määrittelemisen jälkeen, agentti yrittää laskea itselleen reitin, jota seuraamalla se pääsee kohteeseen. Kohteen antamisen lisäksi agentille voidaan laskea reitti erikseen, ja tämä reitti voidaan antaa agentille kohteen sijasta. Reitin laskeminen mahdollistaa sen tarkistuksen ennen sen antamista agentille.

Tekoäly käyttää kohteen antamista ja laskee reitin erikseen vain silloin, kun se on tarpeellista. Esimerkiksi, jos löydetty tulipalo on veneessä, pelastajan on mentävä lähemmäs tulipaloa voidakseen sammuttaa sen. Pelastaja ei kuitenkaan saa kulkea vedessä, joten pelastaja ei voi laskea reittiä suoraan tulipalon luokse. Pelastajan on tämän vuoksi tiedettävä, ettei sillä ole reittiä tulipalon luokse, vaan sen on laskettava reitti vettä lähimpänä olevalle rannalle.

Reitin heijastaminen tietylle navigointiverkon alueelle on yksi eniten tarvittavista tekoälyä ohjaavista asioista. NavMesh-luokka tarjoaa tähän kaksi metodia: SamplePosition ja FindClosestEdge. SamplePosition-metodin avulla voidaan laskea lähin sijainti haluttuun aluetyyppiin ja FindClosestEdge-metodilla lähin sijainti alueen reunalla. Ensimmäisen vaihtoehdon käyttö oli helppoa ja selkeää. Sen haasteena oli kuitenkin se, ettei se palauttanut normaalivektoria lasketulle sijainnille. Toisen vaihtoehdon ymmärtäminen puolestaan oli vaikeaa. Sitä ei osattu käyttää oikein ennen kuin vasta projektin loppuvaiheessa, joten tekoäly käyttää vain SamplePosition-metodia.

4.3 Synkronointi

Synkronointi on tärkeä osa, jonka avulla eri tilakoneiden toiminta voidaan yhdistää yksikön sisällä. Vene on yksinkertaisin yksikkö, joka kattaa kaksi eri tilakonetta, joiden on toimittava yhdessä. Vastaavasti pelastettavien on osattava reagoida pelastamiseen, samoin kuin pelastajien on oltava tietoisia pelastettavien tilasta. Kaikkien on siis toimittava yhdessä, jotta hahmojen toiminta vaikuttaa itsenäiseltä ja todenmukaiselta.

Tilakoneiden välisen synkronoinnin lisäksi on huomioitava, että kyseessä on moninpeli. Tämä tarkoittaa sitä, että peli on synkronoitava myös pelaajien välillä, jotta kaikki pelaajat näkevät samat tapahtumat samaan aikaan.

4.3.1 Usean hahmon tila yksikön sisällä

Yksikkö koostuu useammasta hahmosta, ja jokaisella hahmolla on oma tilakone. Jotta yksikkö voi toimia yhdessä, hahmojen on tiedettävä, mitä muut yksikön hahmot tekevät. Jokaisen hahmon tilan kertominen muille olisi ollut yksi vaihtoehto. Se olisi toiminut ja ollut samalla helppo ylläpitää. Tutkittaessa pelastajan tilakonetta huomattiin, että hahmot eivät tarvitse tarkkaa tietoa siitä, mitä toiset hahmot tekevät. Tarvittavan tiedon pystyi supistamaan kolmeen eri tilanteeseen, jotka ovat: tehtävä on suoritettu, uusi käsky on annettu ja hahmo on suorittanut oman toimintansa ja on valmis jatkamaan seuraavaan vaiheeseen. Nämä kolme olisi voitu lisätä AIUnitInfo-komponenttiin, mutta ne päädyttiin pitämään erillään. Ne toteutettiin luomalla jokaista tilannetta vastaava statuskomponentti.

Statuskomponentit eivät sisällä mitään toimintaa tai tietoa. Ne ovat tyhjiä luokkia, jotka periytetään Unityn MonoBehaviour-luokasta, joten niitä voi käyttää komponentteina. Koska ne ovat komponentteja, niitä voi lisätä hahmolle, niiden olemassaolon voi tarkistaa, ja ne voidaan poistaa tarvittaessa. Yleisesti valittiin yksi hahmo, joka seuraa tilannetta ja laukaisee statuskomponenttien tarkistuksen. Ensimmäinen pelastaja tekee tarkistuksen ja lähettää siitä viestin eteenpäin lisäämällä kuljettajalle oikean statuskomponentin. Jos kuljettaja itse on valmis ja lukee statuskomponentin, hän lisää saman komponentin molemmille pelastajille. Näin kaikki kolme yksikön jäsentä saavat tiedon tilanteesta ja voivat yhdessä jatkaa seuraavaan toimintaan.

Hahmojen määrän kasvaessa, tarvittiin lisää eri statuskomponentteja eri tilanteisiin. Statuskomponentteja lisättiin toiset kolme, jotka kertovat sammutuksesta, merkitsevät pelastettavan pelastuksen alkamista ja kertovat lautan saavan työntöapua uimarilta. Statuskomponenttien luominen osoittautui oikeaksi ratkaisuksi, kun selvitettiin, miten tekoäly ja sen ulkopuoliset järjestelmät saadaan toimimaan yhdessä. Huomattiin, että statuskomponentin avulla voidaan myös siirtää dataa, vaikkei sitä ollut aikaisemmin tarvittu yksikön sisällä, koska kaikki data oli AIUniInfo-komponentissa. Paras esimerkki dataa sisältävästä statuskomponentista on AIExtinguishingFire-komponentti, jonka sammuttamassa oleva pelastaja lisää tuliobjektille. Tämän komponentin kautta sammuttaja kertoo valitun sammutusaineen tuliobjektin kontrollikomponentille. Kuva 24 näyttää miten sammuttajan tilakoneen sammutustila lisää tuliobjektille tiedon sammutuksesta ja sammutustavasta.

```
if (stateMachine.TargetObject &&
    !stateMachine.TargetObject.GetComponent<AIExtinguishingFire>())
{
    string substance = "";

    stateMachine.TargetObject.AddComponent<AIExtinguishingFire>();

    if (aiUnitInfo.GivenMethod == AIUnitInfo.METHOD.WATER)
        substance = "Water";
    if (aiUnitInfo.GivenMethod == AIUnitInfo.METHOD.FOAM)
        substance = "Foam";

    stateMachine.TargetObject.GetComponent<AIExtinguishingFire>().Substance = substance;
}
```

Kuva 24. Sammuttaja lisää tuliobjektille statuskomponentin ja tiedon sammutustavasta

4.3.2 Synkronointi pelaajien välillä

Moninpelitestauksen yhteydessä huomattiin varhaisessa vaiheessa, että jokainen pelaaja ajaa omia kopioitaan tilakoneista. Vaikka tilakoneet eivät vie paljoa resursseja, resurssien käyttö turhaan on tarpeetonta. Varsinaiseksi ongelmaksi tämä kuitenkin muodostui vasta, kun pelastettaville lisättiin heidän statuksestansa huolehtiva komponentti, jonka arvoja satunnaistettiin. Koska satunnaisuus tapahtui yksilöllisesti jokaisen pelaajan kopiossa, pelastettavat yrittivät toimia eri tavalla eri pelaajan laitteella. Käytössä ollut Photon tarjosi ratkaisuja pelaajien väliseen synkronointiin, tosin kaikki valmiit ratkaisut eivät toimineen halutulla tavalla.

Photon jakaa pelaajat kahteen ryhmään, joista ensimmäinen on ”MasterClient”. MasterClient on pelaaja, jonka peli toimii samanaikaisesti asiakkaana ja palvelimena. PhotonView käyttää komponentille määritellyn omistajan peliä vertailukohteena synkronoitavalle datalle ja kopioi tämän datan muille pelaajille. Tästä syystä tilakoneet päädyttiin aktivoimaan vain MasterClientilla, ja muut pelaajat eivät suorita tilakoneita ollenkaan. Näin ollen MasterClient on määritelty omistajaksi jokaiselle PhotonView-komponentille ja toimii siten vertailukohteena objektisynkronoinnille.

Tekoälylle annettiin tehtäväksi hahmojen liikuttaminen ja animaatioiden kontrolloiminen. Ilman tilakonetta, asiakkaat eivät näitä tietoja saa, ellei niitä välitetä MasterClientilta. Photon tarjoaa PhotonView-komponentin, joka huolehtii kaikista samaan peliobjektiin liitetyistä Photonin synkronointikomponenteista. Näistä ensimmäinen lisäys oli PhotonTransformView-komponentti, joka synkronoi Transform-komponentin kaikkien pelaajien välillä. PhotonTransformView riittää huolehtimaan hahmojen liikkumisen synkronoinnista kaikille pelaajille.

Photon tarjoaa myös PhotonAnimatorView-komponentin, joka on tarkoitettu Animator-komponenttien synkronoimiseen. Animator-komponentti puolestaan määrittelee hahmolle liitetyn animaattorin, joka sisältää kaikki hahmon animaatiot. Animaattori toimii tilakoneena, ja sille voidaan määritellä laukaisijoita tai boolean-arvoja, joiden avulla voidaan määritellä, miten animaatioiden välillä siirrytään. Yrityksistä huolimatta PhotonAnimatorView-komponenttia ei saatu toimimaan, joten tilalle oli löydettävä toinen ratkaisu.

Objektisynkronointi on Photonin tarjoama synkronointitapa, jonka avulla on mahdollista synkronoida animaattorin tarvitsema laukaisija. Vaihtoehto tuntui toimivalta, mutta vaati seurakseen komponentin, joka antaisi laukaisijan arvon animaattorille. Kuva 25 näkyy ensimmäinen versio animaattorin synkronoimista varten luodusta objektisynkronointikomponentista.

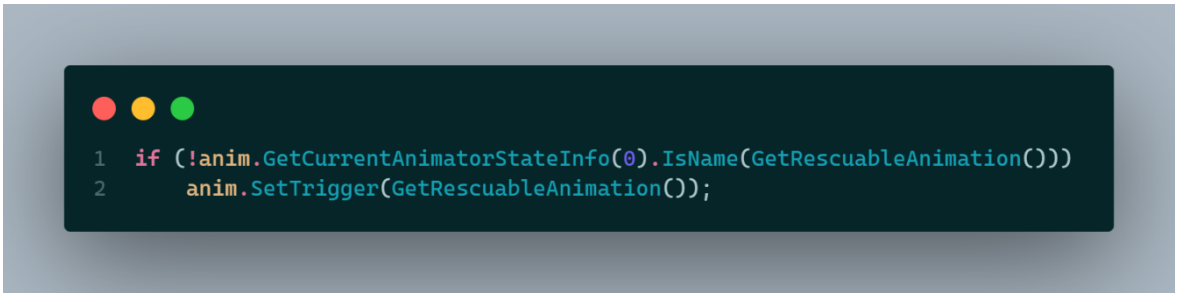
```
using UnityEngine;
using Photon.Pun;

namespace CBSafe.Simulation.AI
{
    public class NPCSync : MonoBehaviourPunCallbacks, IPunObservable
    {
        [SerializeField]
        private string animationTrigger;

        public string AnimationTrigger { get => animationTrigger; set => animationTrigger = value; }

        public void OnPhotonSerializeView(PhotonStream stream, PhotonMessageInfo info)
        {
            if (stream.IsWriting)
            {
                stream.SendNext(AnimationTrigger);
            }
            else if (stream.IsReading)
            {
                AnimationTrigger = (string)stream.ReceiveNext();
            }
        }
    }
}
```

Kuva 25. NPCSync.cs

Komponentin on toteutettava IPunObservable-rajapinta sekä määriteltävä OnPhotonSerializeView-metodi, joka vastaa datan kirjoittamisesta ja lukemisesta synkronointia varten. Tämä komponentti on liitettävä peliobjektiin, johon on liitetty myös PhotonView-komponentti, joka huolehtii synkronoinnista. Data siirtyy PhotonStream-luokan avulla tyypittömänä. Tästä syystä on huolehdittava siitä, että data luetaan oikean tyyppisenä.

Synkronoitu laukaisija välitettiin animaattorille omassa komponentissa, joka seuraa NPCSync-komponentin AnimatorTrigger-attribuutin arvoa ja päivittää animaattorin tilan, jos

se on muuttunut. Kuva 26 esittää kuinka animaattorin tilan vertailu laukaisijaan on mahdollista, ja kuinka uusi laukaisija asetetaan. Animaattorin aktiivista tilaa ei voi lukea, joten tämä tapa on tarpeellinen.



```
1  if (!anim.GetCurrentAnimatorStateInfo(0).IsName(GetRescuableAnimation()))
2      anim.SetTrigger(GetRescuableAnimation());
```

Kuva 26. Animaattorin tilan vertailu ja uuden tilan laukaiseminen

Animaattorin tilan päivittäminen toimi muutamalla koodirivillä, kun Animator-komponentille oli määritelty oikea AnimatorController, ja hahmolla oli vain yksi asu, joka oli aktiivinen koko ajan. Kun hahmoille lisättiin oma asu uimista varten, jouduttiin tämä huomioimaan myös animaatioiden synkronoimisessa. Oli tarkistettava, että animaatioiden synkronointi tietäisi koko ajan, mikä hahmon asuista oli aktiivinen. Lisäksi oli varmistettava, että asun Animator-komponentilla oli oikea kontrolleri. Animaattorin kontrolleri olisi voitu asettaa editorissa, mutta oli tiedossa, että kontrollerit päivitetään uusien animaatioiden valmistuessa. Oli siten järkevämpää antaa synkronointikomponentin hoitaa kontrollerin määrittäminen.

5 Yhteenveto ja pohdinta

Tavoitteena oli toteuttaa tekoäly VR-harjoitteluympäristön hahmoille. Toteutukseen valittu tilakone-malli osoittautui jo aluksi hyväksi valinnaksi sen yksinkertaisen toiminnan ja laajennettavuuden vuoksi. Käskyjen antaminen tapahtumakutsujen avulla, ja sen tallentaminen yksikön omaan dataluokkaan, mahdollisti tavan, jolla pelaaja voi hallita yksikkönsä toimintaa. Statuskomponenttien lisääminen ratkaisi hyvin niin yksiköiden sisäiset synkronointihaasteet kuin esimerkiksi tulen sammuttamisen.

Unityn navigointitoteutus tuli nopeasti tutuksi, joten hahmojen liikkumisen kanssa suurimmat ongelmat johtuivat virheellisen määränpään antamisesta. Tätä pyrittiin välttämään siten, että määränpää projisoitiin navigointiverkolle. Projisointi helpotti varsinkin rajatapausten kohdalla.

Moninpelin haasteet vaikuttivat aluksi suurilta. Photonin suppea dokumentointi ei tässä auttanut, vaan ratkaisut oli luotava pitkälti itse. Tästä huolimatta ne onnistuttiin ratkaisemaan, ja lopputuloksesta saatiin mielestäni varsin hyvä.

Tuloksena saavutettiin toimiva malli, jonka avulla VR-harjoitteluympäristön tuomia mahdollisuuksia pääsee alustavasti selvittämään. Samalla on hyvä alkaa suunnittelemaan, miten ympäristöä halutaan laajentaa jatkokehityksen aikana.

Mieleen jäi muutamia asioita, jotka toteuttaisin toisin seuraavalla kerralla. En ollut projektissa alusta alkaen, joten en päässyt hyödyntämään tilaajan asiantuntemusta niin paljon, kuin olisin halunnut. Tästä syystä keskittyminen todenmukaiseen toimintaan toteutettiin yleensä selvittämällä ensin asiaan liittyvät fysiikan lait ja laskut, ja etsimällä tietoa Internetistä. Olen erittäin mielissäni niistä tiedoista, jotka saimme tilaajalta. Hyvänä esimerkkinä pintapelastuksen uusi toimintaohje, jonka onnistuimme toteuttamaan.

Suurin osa tulevaisuuden parannuksista liittyy omaan osaamiseeni Unityn kanssa. Liittyessäni projektiin kehittäjänä, osasin Unitystä vain sen, mitä olin oppinut kursseilta. Navigointi oli suurimmilta osin itselleni uutta, joten en osannut hyödyntää kaikkia Unityn tarjoamia mahdollisuuksia. Esimerkiksi reittien laskeminen etukäteen ja hahmojen liikuttaminen niiden avulla edustavat näitä mahdollisuuksia. Päädyin toteutuksessa vertaamaan etäisyyttä hahmon ja tämän päämäärän välillä. Tämä aiheutti haasteita, jotka olisi voinut välttää reittien laskemisella. Vastaavasti ajoneuvojen agenttien liikkuminen on kaukana todenmukaisesta, koska en osannut määritellä agenttia oikein.

Navigointiin liittyen ongelmien ratkaiseminen olisi ollut huomattavasti helpompaa, jos päämääriä olisi visualisoitu esimerkiksi lisäämällä 3D-objekti (primitiivi) ilman törmäytintä

(collider) hahmolle annettuun päämäärään. Tämä olisi osoittanut heti, jos päämäärä on saavutettavissa ja laskettu oikein. Unityn ja editorin välinen yhteistyö on hyvää, mutta varsinainen virheiden etsintä on haastavaa ilman oikeita työkaluja. Koodin yksikkö- ja integraatiotestaus oli mahdollista, mutten löytänyt sopivaa työkalua, joka olisi osannut huomioida myös Unityn.

Tilojen refaktorointi on loputon haaste, joka korostuu niiden toiminnan ja määrän kasvaessa. Jälkeenpäin ajateltuna, olisin halunnut pilkkoa ainakin pelastajien Idle- ja Walk-tilat pienempiin osiin. Tein alussa sen virheen, että liitin paljon toimintaa yhteen tilaan. Tämä vaikeuttaa ongelmien löytämistä ja ratkaisua. Mitä yksinkertaisempana tila toteutetaan, sitä helpompi sen ylläpito on jatkossa.

En ollut suoraan vastuussa animaatioista, mutta tekoälyn tavoitteisiin luettiin animaatioiden kontrolloiminen. Ensimmäisen animaattorin kohdalla ei ollut suurempia haasteita. Haasteita ilmeni uusien animaatioiden lisäämisen jälkeen, sillä uusi animaattori oli toteutettu kerroksia (layer) käyttämällä. Animaatioiden päivitys olisi ollut helpompaa, jos olisin ollut aikaisemmin osana animaatioiden toteutusta ja siten tuntenut Unityn animaattorin paremmin.

Seuraavana vuorossa olisi uusien tilanteiden lisääminen ja toiminnan laajentaminen. Pääsimme vasta raapaisemaan pintaa, joten mahdollisuuksia on paljon. Toisena tilanteena aloitettiin luomaan kaatunutta säiliöautoa, jonka tankki oli alkanut vuotamaan. Tilannetta varten pelastajien tekoälyyn lisättiin runko vahingontorjuntaan, mistä olisi helppo jatkaa. Toinen mahdollisuus olisi laajentaa tulen leviämistä materian kautta, jonka avulla voitaisiin toteuttaa rakennus- tai maastopalo seuraavaksi.

Näen toteutetulla VR-harjoittelualustalla paljon potentiaalia tulevaisuudessa.

Lähteet

Brodkin, J. 2013. How Unity3D Became a Game-Development Beast. Dice Insights. Viitattu 14.10.2022. Saatavissa https://insights.dice.com/2013/06/03/how-unity3d-become-a-game-development-beast/

Kyaw, A., Peters, C. & Swe, T. 2013. Unity 4.x Game AI Programming. Birmingham: Packt Publishing Ltd.

MacroTrends. 2022. Unity Software: Number of Employees 2019-2022. Viitattu 14.10.2022. Saatavissa https://www.macrotrends.net/stocks/charts/U/unity-software/number-of-employees

Millington, I. 2019. AI for Games. 3. painos. Boca Raton: CRC Press.

Millington, I. 2022. AI for Games. AI for Everything. Boca Raton: CRC Press.

Patel, A. 2022a. Introduction to A*. Viitattu 17.10.2022. Saatavissa http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html

Patel, A. 2022b. Introduction to the A* Algorithm. Viitattu 7.11.2022. Saatavissa https://www.redblobgames.com/pathfinding/a-star/introduction.html

Patel, A. 2022c. Heuristics. Viitattu 17.10.2022. Saatavissa http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html

Photon. 2022a. Introduction. Viitattu 22.10.2022. Saatavissa https://doc.photonengine.com/en-us/pun/current/getting-started/pun-intro

Photon. 2022b. Ownership & Control. Viitattu 22.10.2022. Saatavissa https://doc.photonengine.com/en-us/pun/current/gameplay/ownershipandcontrol

Photon. 2022c. Synchronization and State. Viitattu 22.10.2022. Saatavissa https://doc.photonengine.com/en-us/pun/current/gameplay/synchronization-and-state

Photon. 2022d. RPCs and RaiseEvent. Viitattu 22.10.2022. Saatavissa https://doc.photonengine.com/en-us/pun/current/gameplay/rpcsandraiseevent

Schardon, L. 2022. What is Unity? – A Guide for One of the Top Game Engines. GameDev Academy. Viitattu 14.10.2022. Saatavissa https://gamedevacademy.org/what-is-unity/

Tepfenhart, W. & Wang, J. 2020. Formal Methods in Computer Science. Boca Raton: CRC Press.

Unity. 2022a. System requirements for Unity 2021 LTS. Viitattu 14.10.2022. Saatavissa https://docs.unity3d.com/Manual/system-requirements.html

Unity. 2022b. GameObjects. Viitattu 15.10.2022. Saatavissa https://docs.unity3d.com/Manual/GameObjects.html

Unity. 2022c. Important Classes – MonoBehaviour. Viitattu 15.10.2022. Saatavissa https://docs.unity3d.com/Manual/class-MonoBehaviour.html

Unity. 2022d. Inner Workings of the Navigation System. Viitattu 22.10.2022. Saatavissa https://docs.unity3d.com/Manual/nav-InnerWorkings.html

Unity. 2022e. Script lifecycle flowchart. Viitattu 14.10.2022. Saatavissa https://docs.unity3d.com/uploads/Main/monobehaviour_flowchart.svg

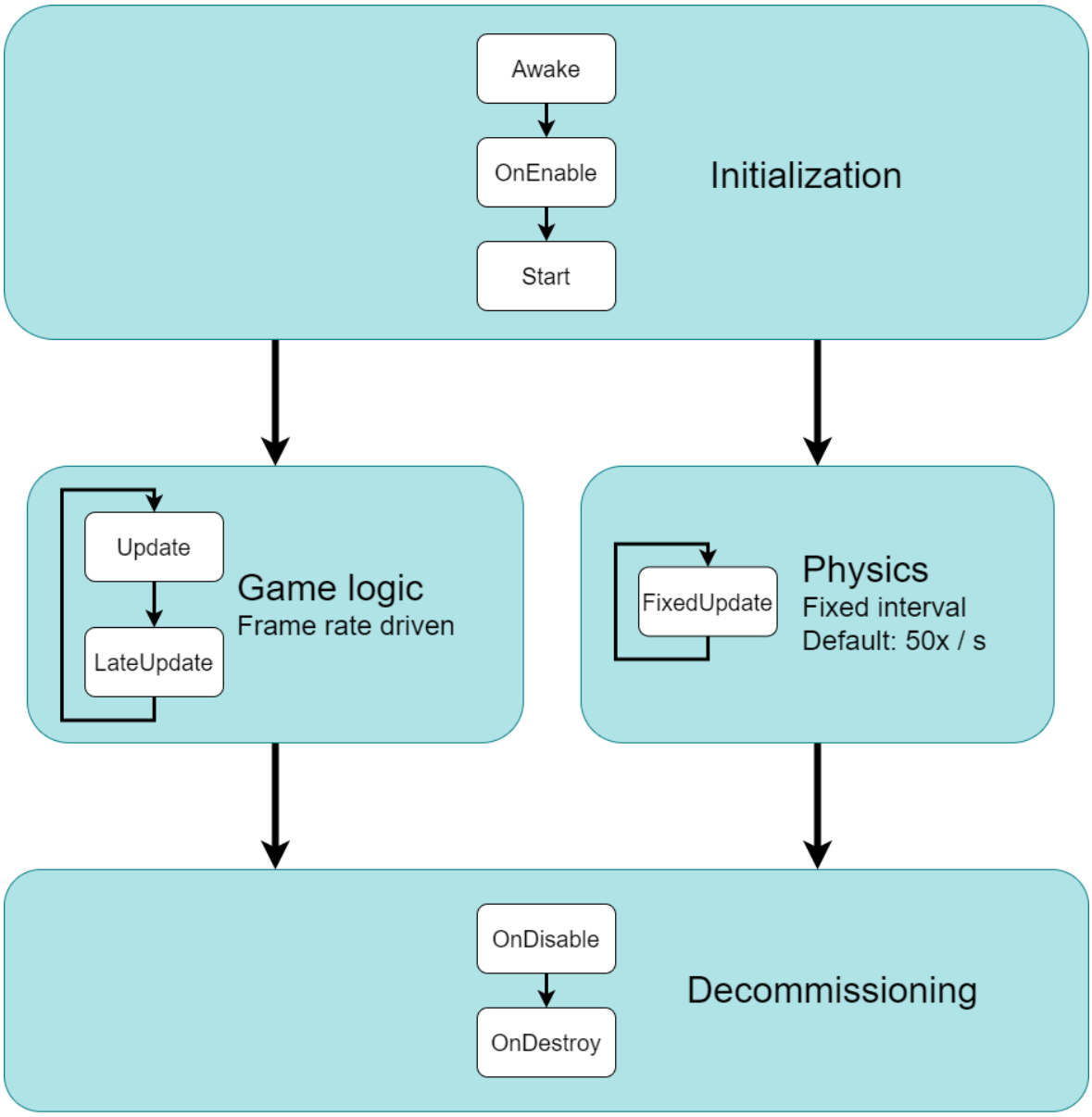
Unity Scripting API. 2022a. MonoBehaviour.Awake(). Viitattu 15.10.2022. Saatavissa https://docs.unity3d.com/ScriptReference/MonoBehaviour.Awake.html

Unity Scripting API. 2022b. MonoBehaviour.FixedUpdate(). Viitattu 15.10.2022. Saatavissa https://docs.unity3d.com/ScriptReference/MonoBehaviour.FixedUpdate.html

William, E. 2020. Learning C# by Developing Games with Unity. 2. Painos. Yksityinen kustantaja.

Wright, D. 2005. Finite State Machines. Viitattu 23.10.2022. Saatavissa https://web.archive.org/web/20140327131120/http://www4.ncsu.edu/~drwrigh3/docs/courses/csc216/fsm-notes.pdf

Liite 1. Unityn skriptin elinkaari



Kuva 27. Unityn skriptin elinkaari (mukaillen Unity 2022e)

Liite 2. Witness-tilakoneen lähdekoodi

```
using UnityEngine;
using UnityEngine.AI;

namespace CBSafe.Simulation.AI
{
    3 references
    public class AIStateMachineWitness : MonoBehaviour
    {
        2 references
        NavMeshAgent agent;
        3 references
        State currentState;

        2 references
        public Vector3 target = Vector3.zero;

        0 references
        public Vector3 Target
        {
            get { return target; }
            set {
                NavMesh.SamplePosition(value, out NavMeshHit hit, 5.0f, NavMesh.AllAreas);
                target = hit.position;
            }
        }

        0 references
        void Start()
        {
            agent = GetComponent<NavMeshAgent>();
            currentState = new WitnessIdle(gameObject, agent);
        }

        0 references
        void Update()
        {
            currentState = currentState.Process();
        }
    }
}
```

Kuva 28. AIStateMachineWitness.cs

Liite 2. Witness-tilakoneen lähdekoodi

```
using UnityEngine;
using UnityEngine.AI;

namespace CBSafe.Simulation.AI
{
    31 references
    public class State
    {
        213 references
        public enum EVENT
        {
            2 references | 3 references | 207 references
            ENTER, UPDATE, EXIT
        };

        212 references
        protected EVENT stage;
        876 references
        protected GameObject npc;
        0 references
        protected Animator anim;
        206 references
        protected State nextState;
        372 references
        protected NavMeshAgent agent;

        297 references
        protected ScenarioVariables scenarioVariables;

        10 references
        public State(GameObject _npc, NavMeshAgent _agent)
        {
            npc = _npc;
            agent = _agent;
            stage = EVENT.ENTER;
            scenarioVariables = GameObject.Find("ScenarioVariables").
            GetComponent<ScenarioVariables>();
        }

        79 references
        public virtual void Enter() { stage = EVENT.UPDATE; }
        1 reference
        public virtual void Update() { stage = EVENT.UPDATE; }
        79 references
        public virtual void Exit() { stage = EVENT.EXIT; }

        9 references
        public State Process()
        {
            if (stage == EVENT.ENTER) Enter();
            if (stage == EVENT.UPDATE) Update();
            if (stage == EVENT.EXIT)
            {
                Exit();
                return nextState;
            }
            return this;
        }
    }
}
```

Kuva 29. State.cs

Liite 2. Witness-tilakoneen lähdekoodi

```
using UnityEngine;
using UnityEngine.AI;

namespace CBSafe.Simulation.AI
{
    2 references
    public class WitnessState : State
    {
        1 reference
        protected AIStateMachineWitness stateMachine;
        4 references
        protected NPCSync npcSync;

        2 references
        public enum STATE
        {
            1 reference
            IDLE
        }

        1 reference
        public STATE name;

        1 reference
        public WitnessState(GameObject _npc, NavMeshAgent _agent)
            : base(_npc, _agent)
        {
            stateMachine = npc.GetComponent<AIStateMachineWitness>();
            npcSync = npc.GetComponent<NPCSync>();
            npcSync.StateMachine = "Witness";
        }
    }
}
```

Kuva 30. WitnessState.cs

Liite 2. Witness-tilakoneen lähdekoodi

```
using UnityEngine;
using UnityEngine.AI;

namespace CBSafe.Simulation.AI
{
    1 reference
    public class WitnessIdle : WitnessState
    {
        1 reference
        public WitnessIdle(GameObject _npc, NavMeshAgent _agent)
            : base(_npc, _agent)
        {
            name = STATE.IDLE;
            npcSync.State = "Idle";
        }

        79 references
        public override void Enter()
        {
            Debug.Log(npc.name + " " + "Witness Idle State");

            npcSync.AnimationTrigger = "isIdle";

            scenarioVariables.Speed.TryGetValue("walk", out float value);
            agent.speed = value;

            base.Enter();
        }

        1 reference
        public override void Update()
        {
        }

        79 references
        public override void Exit()
        {
            base.Exit();
        }
    }
}
```

Kuva 31. WitnessIdle.