Osaamista
ja oivallusta
tulevaisuuden
tekemiseen

Ella Luukkonen

Hahmojoukon tekoälytoteutus tarvejärjestelmän avulla

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

6.11.2019

Tekijä Otsikko Sivumäärä Aika	Ella Luukkonen Hahmojoukon tekoälytoteutus tarvejärjestelmän avulla 38 sivua 6.11.2019
Tutkinto	Insinööri (AMK)
Tutkinto-ohjelma	Tieto- ja viestintätekniikka
Ammatillinen pääaine	Pelisovellukset
Ohjaaja	Lehtori Antti Laiho

Insinöörityössä tutkittiin yleisimpiä tekoälymenetelmiä videopeleissä etenkin pelihahmotekoälyn toteutuksessa. Tarkoituksena oli luoda peliprojektiin tekoäly, joka ohjaa pelihahmojoukon toimintaa. Peliprojekti ja tekoälytoteutus toteutettiin Unity-pelimoottoria ja C#-ohjelmointikieltä käyttäen. Tekoälyjärjestelmäksi valittiin niin kutsuttu ”tarvejärjestelmä”.

Insinöörityön tutkintavaiheessa selvitettiin tekoälyn määritelmää videopeleissä ja niiden ulkopuolella. Tarkemmin perehdyttiin muutamiin käytetyimpiin menetelmiin pelitekoälyn alalla, kuten tilakoneeseen, polunetsintä- ja tiedonoppimisalgoritmeihin.

Tutkinta- ja suunnitteluvaiheessa havaittiin eri tekoälymenetelmien sopivan erilaisiin ratkaisuihin, ja tämän perusteella saatiin rajattua sopivaa menetelmää, jota lopulta käytettäisiin insinöörityössä. Tekoälyn pitäisi pystyä vaihtamaan pelihahmon toimintaa pelin aikana ja tarpeiden evaluointi tulisi tapahtua tavalla, joka saisi aikaan järkevän pelikokemuksen pelaajalle.

Lopulliseksi toteutustavaksi työhön valittiin tarvejärjestelmä, joka mahdollistaa pelihahmolle useita erilaisia toimintatapoja, jotka määräytyvät asetettujen tarpeiden mukaan. Pelihahmolle määriteltiin neljä eri tarvetilaa: normaali, ravitsemus, lepo ja pakoon juoksu. Toteutuneet tarpeet toimivat lähes suunnitellulla tavalla, muutamaa toimintahäiriötä lukuun ottamatta.

Insinöörityössä saatiin rakennettua toimiva pohja tekoälyratkaisulle, jossa pelihahmolla on neljä eri toimintatilaa. Toimintatiloja ohjaava tekoäly määrittää pelihahmon toiminnolle tarpeet, ja suoritettavaksi toiminnoksi valitaan toiminto, jolla on korkein tarvelukema. Tarpeita lasketaan Unity-pelimoottorissa olevien animaatiokäyrien avulla. Kolme neljästä tarpeesta toimii suunnitelman mukaisesti. Lepotarpeen evaluointiin jäi laskentavirhe, jota ei ehditty korjaamaan. Toteutunut ratkaisu on kuitenkin toimiva pohja tekoälyn toimintojen kehittämiselle.

Avainsanat	tekoäly, pelihahmo, Unity

Author Title Number of Pages Date	Ella Luukkonen Game AI Implementation for Game Characters Using Utility System 38 pages 6 November 2019
Degree	Bachelor of Engineering
Degree Programme	Information and Communication Technology
Professional Major	Game Applications
Instructor	Antti Laiho, Senior Lecturer

The thesis researches the most commonly used artificial intelligence systems used in the videogame industry. Special concentration is put on the use of artificial intelligence with videogame character behavior. The game project and the artificial intelligence were created in Unity, using C# programming language. The chosen system for the artificial intelligence was utility system.

In the thesis' research part, the definition for artificial intelligence was examined in and outside of the videogame genre. A closer examination was done on some of the most used artificial intelligence implementations. Examples are state machine and algorithms for pathfinding and different data-learning-mechanisms.

In the research and designing part it was discovered, that different artificial intelligence solutions suit different types of projects better than others. This allowed different option to be narrowed down for solutions that would best suit the thesis. The artificial intelligence would need to be able to handle different states for the game character and chance the behavior of the character in question. The evaluation for the chosen action should be done in a way that makes sense in a gameplay perspective.

The chosen solution ended up being the utility system, that allows the game character to act according to different needs that correspond to the need utility. The game character has four utility states: normal, health, rest and running away.

The most of these utilities work as designed, all the utility states were created and at some point, were proven to work. Every utility works as intended, except for the rest state. The evaluation of the rest utility has some errors and thus prevents the state to work properly. The current state of the rest utility never becomes to highest utility for the character to perform that action. Otherwise the solution created works as a good base for further improvement.

Keywords	Game AI, Game Character, Unity

Sisällys

1 Johdanto

Tekoäly on nimi tietokonetekniikalle, jossa tarkoitus on simuloida älyllistä toimintaa. Yleensä älyllisellä toiminnalla tarkoitetaan tässä yhteydessä toimintaa, joka simuloi tapaa, miten ihminen tai eläin käyttäytyisi. Tätä voidaan lähestyä eri näkökulmista, ja tekoälyllä yleensä pyritään saamaan aikaan vain joitakin ihmiskäyttäytymisen ominaisuuksia, kuten päätöksentekoa tai tunnistusta. [1, s. 4.]

Videopeleissä on ollut yksinkertaisia tekoälyratkaisuja lähes yhtä pitkään, kuin videopelejä on ollut. Videopelien toiminnan kannalta ei aina ole edes toivottavaa täysin ihmismäinen käyttäytyminen tai oppiminen, mutta tietyillä ominaisuuksilla voidaan lisätä mielekkäämpää toimivuutta peliin. Yleinen esimerkki on esimerkiksi ei-pelattavien hahmojen käyttäytyminen. Käyttäytyminen luodaan yleensä liikkumisella ja päätöksenteolla. [2.]

Insinöörityössä on tarkoitus suunnitella ja toteuttaa yksinkertainen tekoälytoteutus videopeliin, jossa tekoäly ohjaa ei-pelattavien hahmojen toimintaa pelimaailmassa. Työssä selvitetään yleisimpiä toteutustapoja pelitekoälyn toteuttamiseen ja tutustutaan yleisimpiin pelitekoälyssä käytettäviin algoritmeihin ja periaatteisiin. Lopuksi verrataan toteutustapojen ominaisuuksia toteuttaa toivotut ominaisuudet.

Tavoitteena on saada pelihahmojen toimintalogiikka toimimaan suunnitellulla tavalla ja luoda pohja mielekkäälle pelikokemukselle, jossa hahmojoukon tarpeet vaikuttavat hahmon tekoälyn toimintaperiaatteisiin eri tavoin. Pelihahmolla tulee olla erilaisia tarpeita, ja riippuen kunkin tarpeen voimakkuudesta tekoälyn on osattava muuttaa hahmon toimintaa. Toteutus luodaan Unity-pelimoottorin sisällä käyttäen C#-ohjelmointikieltä ja tarvetekoälyjärjestelmää.

Aihe valittiin aikaisemman peliprojektin vuoksi, kun peliprojektin toteutus ei vastannut lopulta enää alkuperäistä suunnitelmaa. Ideaa tahdottiin jatkaa erilaisella toteutustavalla, jossa hahmotekoälyn toiminnalla on suurempi osuus.

2 Tekoälyn käyttö videopeleissä

2.1 Tekoälyn määritelmä

Tekoälyllä tarkoitetaan karkeasti tietokoneen käyttäytymistä tavalla, joka noudattaa ihmisen tai eläimen ajattelun tasoa. Tekoälyn avulla tietokoneita on opetettu pelaamaan pelejä paremmin kuin kenenkään ihmisen olisi edes mahdollista pelata sekä suoriutumaan monista muista ongelmista. Tietokoneen on kuitenkin vaikea hallita ihmisille yksinkertaiselta tuntuvia tehtäviä, kuten oman kielen puhuminen tai luova ajattelu. Tekoälyteknologian onkin tarkoitus löytää algoritmeja, joilla näitä ominaisuuksia voisi ilmentää. [1, s. 4.]

Tekoälylle on useita eri määritelmiä historiassa, mutta yhtenäistä niillä on usein vertaus ihmismäiseen käyttäytymiseen tai päätöksentekoon. Kyky ymmärtää ja tehdä opitun tiedon pohjalta päätöksiä ja muuttaa käyttäytymistä sen mukaan on pääroolissa tekoälystä puhuttaessa. Alan Turingin luomassa testissä tekoäly kirjoitti vastauksia ihmisen esittämiin kysymyksiin. Testi katsottiin onnistuneeksi, kun ihmiskuulustelija ei kyennyt enää päättelemään, oliko vastauksen kirjoittanut kone vai ihminen. Tässäkin testissä koneen oli pystyttävä prosessoimaan kieltä, sisäistämään ja säilyttämään informaatiota ja käyttämään tätä tietoa johtopäätöksien tekemiseen. Turingin testi käsittää tekoälyn kykyä käyttäytyä tavalla, jota mielletään ihmismäiseksi käyttäytymiseksi. [3, s.1–2.]

Tekoälyä voi tutkia myös ihmismäisen ajattelun kannalta sekä rationaalisen ajattelun ja käyttäytymisen kannalta. Ihmismäisen ajattelun määrittäminen vaatii tarkan katsauksen ihmismielen ja ajatusten toimintaan, sillä vasta tarkan teorian esittämisen jälkeen on mahdollista lähteä yrittämään ohjelmoida vastaavaa ihmismäistä käyttäytymistä. Ihmismielen ajatusten tutkiminen ja mallintaminen kuuluu enemmän kognitiotieteen puolelle, mutta näiden mallien perusteella on mahdollista lähteä selvittämään yhtäläisyyksiä ihmismielen ja tekoälyn toiminnan välillä. [3, s.2–3.]

Jos tekoälyn on tarkoitus simuloida ihmismieltä, pitää ensin määritellä ihmisen mieli. Määritelmää voi rakentaa eri tieteenalojen kannalta esittämällä niille ominaisia kysymyksiä mielen toiminnasta. Filosofian kannalta voidaan pohtia, kuinka mieli erottuu fyysisistä aivoista, mistä tieto tulee ja kuinka tieto johtaa toimintaan. Matematiikassa kysytään, mitä

voidaan laskea ja miten perustellaan epävarmaa informaatiota. Taloustieteen kannalta kysytään, kuinka toimia tuoton maksimoinniksi ja miten toimia, jos tuotto on saavutettavissa vasta kaukana tulevaisuudessa. Asiaa voi pohtia myös neurotieteen kannalta: miten aivot prosessoivat informaatiota tai psykologian näkökulmasta: miten ihmiset ja eläimet ajattelevat. Kuvassa 1 on jaettu tekoälyn määritelmää neljään eri kategoriaan ja kaksi eri määritelmää kustakin kategoriasta.

Thinking Humanly	**Thinking Rationally**
"The exciting new effort to make computers think . . . *machines with minds*, in the full and literal sense." (Haugeland, 1985)	"The study of mental faculties through the use of computational models." (Charniak and McDermott, 1985)
"[The automation of] activities that we associate with human thinking, activities such as decision-making, problem solving, learning . . ." (Bellman, 1978)	"The study of the computations that make it possible to perceive, reason, and act." (Winston, 1992)
Acting Humanly	**Acting Rationally**
"The art of creating machines that perform functions that require intelligence when performed by people." (Kurzweil, 1990)	"Computational Intelligence is the study of the design of intelligent agents." (Poole *et al.*, 1998)
"The study of how to make computers do things at which, at the moment, people are better." (Rich and Knight, 1991)	"AI . . . is concerned with intelligent behavior in artifacts." (Nilsson, 1998)

Kuva 1. Tekoälyn määritelmiä neljässä kategoriassa [3].

Ihmismielen toiminnan määrittely itsessään tuottaa haasteita. Vaikka mielen ajateltaisiin toimivan edes osittain loogisiin sääntöihin perustuen, ihmismielen mieltäminen fyysiseksi järjestelmäksi, jota voidaan emuloida keinotekoisessa fyysisessä järjestelmässä voi luoda ongelmia. Tämä jättää vähän tilaa vapaalle ajattelulle, sille, kuinka vaihtoehdot koetaan. Filosofisessa näkökulmassa tekoälyä ajatellessa on otettava huomioon myös tiedon ja toiminnan yhteys. Tekoäly vaatii toimiakseen suoritettavan toiminnon ja ajattelun toiminnon takana. Oikeutetun toiminnan aikaansaamiseksi on siis ymmärrettävä, miten toiminta oikeutetaan. Aristoteles perustelee teoksessaan Eläinten liikkeestä (lat. De motu animalium) tekojen olevan perusteltuja, kun on saatu looginen yhteys teosta aiheutuvalle seuraukselle ja tavoitellulle päämäärälle. [3, s.6–7.]

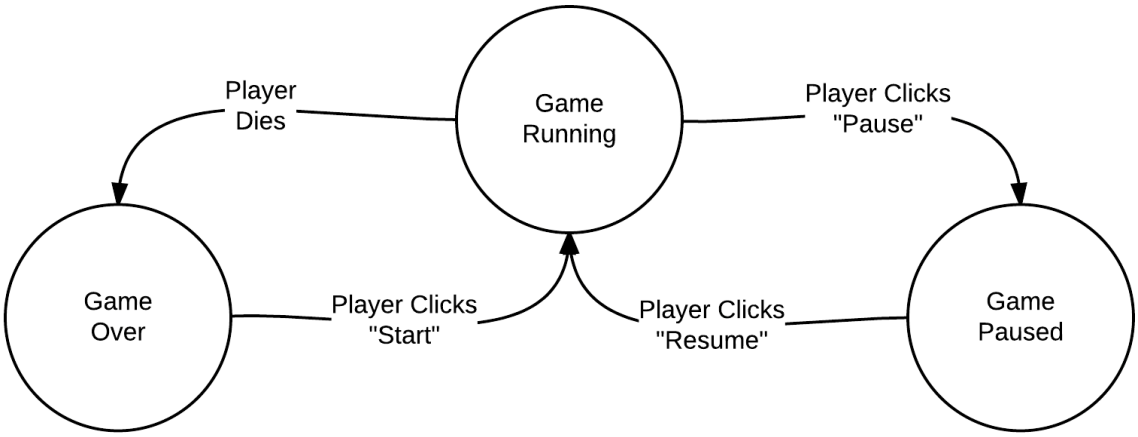
Videopeleissä tekoäly on yleensä taustalla toimiva ominaisuus, joka ohjaa esimerkiksi pelihahmojen liikkumista, päätöksentekoa ja mahdollisia strategioita. Jokaiseen niistä on omia erilaisia toteutustapoja ja algoritmeja. Useat pelit hyödyntävät vain hahmon liikkumiseen ja päätöksentekoon tekoälyä, sillä ne voivat oikein toteutettuina luoda tarpeeksi hyvin toimivan ratkaisun peliin haluttuun toimintaan. [1, s. 10–11].

2.2 Käytetyimmät tekoälymenetelmät

Tekoälyn luomiseen on useita erilaisia toteutustapoja, jotka sopivat eri käyttötarkoituksiin. Eri algoritmeilla ja toteutuksilla on omat vahvuutensa ja niistä riippuen niitä voidaan suosia joillakin aloilla enemmän kuin toisilla. Seuraavaksi tarkastellaan tarkemmin muutamia yleisimmin käytettäviä tekniikoita.

Tilakone

Alun perin insinöörityöprojektin toteutukseen suunniteltiin tilakonetta. Kuvassa 2 on visualisoitu esimerkkitilanne tilakoneen toiminnasta, jossa pelin tila muuttuu eri toimintoja suoritettaessa. Englanninkielinen termi on finite-state machine, ja varsinainen suomennettu termi on äärellinen automaatti. Työssä käytetään kuitenkin termiä tilakone. Tilakone on algoritmien suunnitteluun käytetty käsite, jossa tilakone lukee eri syöttöjä ja vaihtaa niiden mukaan tilaansa. Jokaisessa tilassa on määritetty, millä syötöllä mihinkin tilaan vaihdetaan. [4.]



Kuva 2. Visualisoitu tilakoneen toiminta [5].

Tilakoneen avulla voidaan jaotella isoja monimutkaisia kokonaisuuksia itsenäisiksi pienemmiksi tehtäviksi. Nämä tehtävät ovat kytkettyinä toisiinsa eri tapahtumilla, ja tilan vaihtuessa yleensä suoritetaan jokin toiminto. Jakamalla monimutkaiset toiminnot pienemmiksi yksiköiksi on helpompi hallita näitä yksiköitä abstraktimmalla tavalla. Tarvitsee vain konfiguroida, milloin tila voi siirtyä toiseen tilaan, ja keskittyä määrittämään, mitä tapahtuu siirtymän sattuessa. Voidaan keskittyä ohjelman toiminnon kannalta vain kysymyksiin, milloin ja mitä, eikä miten. [6.]

A*-algoritmi

A*-algoritmi on yleisesti polunetsinnässä käytetty tietokonealgoritmi, joka pyrkii hakemaan puurakenteisesti parhaan reitin haluttuun päämäärään. Paras ratkaisu voi tilanteesta riippuen olla esimerkiksi matkallisesti lyhin tai ajallisesti nopein. [7.] Funktiossa 1 esitetään A*-etsintäalgoritmin toimintaa, jossa yhdistetään solmuun pääsyyn kuluva hinta $g(n)$ ja solmusta haluttuun päämäärään kuluva hinta $h(n)$.

$$f(n) = g(n) + h(n) \tag{1}$$

Funktiossa n on seuraava solmu polulla, $g(n)$ on polun hinta aloitussolmusta n:ään ja $h(n)$ on heuristinen funktio, joka arvioi kulutuksen halvimmalle reitille n:stä päämäärään. Tämä laskenta tehdään A*-algoritmin jokaisessa silmukassa. Kun yritetään löytää tällä periaatteella halvin reitti, aloitetaan ensin solmusta, jossa $g(n) + h(n)$ antaa pienimmän arvon, olettaen, että $h(n)$ täyttää tarvittavat ehdot. [3, s. 93.]

Ensimmäinen ehto on, ettei funktio saa koskaan yliarvioida hintaa päämäärään pääsylle. Toinen ehto on, että funktio on johdonmukainen. Heuristinen $h(n)$ on johdonmukainen, kun jokainen n:n generoiman seuraajan odotettu hinta aloitussolmusta päämäärän saavuttamiselle ei ole korkeampi kuin askeleen hinta seuraajasolmulle ja tähän summattuna oletettu hinta päämäärään seuraajasolmusta. [3, s. 94–95.]

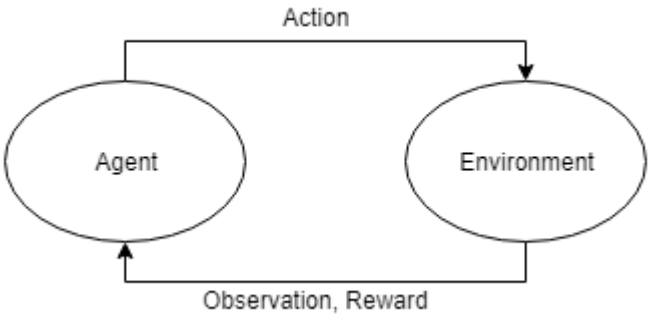
Koneoppiminen

Koneoppiminen on yksi tekoälyjen osa-alue. Koneoppimisessa on kyse datan oppimisesta; kone tai ohjelma saa käsiteltäväkseen dataa, jonka pohjalta ohjelma muodostaa itsenäisesti toimintatapoja erilaisiin tilanteisiin.

Koneoppimisessa on useita oppimistapoja, joilla algoritmi opettelee sille annettua dataa. Oppimistapoja ovat esimerkiksi ohjattu oppiminen (engl. supervised learning), ei-ohjattu oppiminen (engl. unsupervised learning), vahvistusoppiminen (engl. reinforcement learning) ja ominaisuusoppiminen (engl. feature learning). Ohjatussa oppimisessa koneelle annetussa oppimisdatassa on tiedossa sekä annettu informaatio että haluttu lopputulos. Tällöin ohjelma käy läpi opetusdataa, jossa on useita erilaisia alku- ja loppulukemia. Niiden opettelu luo logiikan, jossa tietty alussa saatu informaatio johtaa tiettyyn lopputulokseen. Oppimisen jälkeen ohjelma pystyy itsenäisesti päätymään uusiin lopputuloksiin noudattamalla opetusdatasta oppimaansa logiikkaa. Ei-ohjatussa oppimisessa annetaan vain aloitusinformaatio, josta ohjelma etsii kaavoja ja yhtäläisyyksiä datasta. [8.]

Vahvistusoppimisessa ohjelmalle ei anneta aloitusinformaatiota, eikä varsinaista haluttua lopputulosta. Ohjelman toimija, esimerkiksi pelihahmo, suorittaa jonkin toiminnan peliympäristössä ja peliympäristö palauttaa toiminnasta saadun havainnon ja ”palkinnon”. Saadun palkinnon perusteella pelihahmon toiminta päättää joko jatkaa tätä samaa toimintaa, tai välttää sen tekemistä uudelleen. Toimija pyrkii saamaan mahdollisimman suuren palkinnon toiminnallaan ja jatkaa toimintaansa saamansa palautteen mukaan. [9.]

Esimerkki vahvistusoppimisesta taistelupelissä voisi olla pelihahmo tilanteen toimijana, vihollinen ja taisteluareena muodostaisivat tilanteen peliympäristön. Kuvassa 3 nähdään esimerkki, jossa kuvataan pelihahmon toiminnan vaikutus ympäristöön ja tästä aiheutuva seuraus.



Kuva 3. Diagrammi vahvistusoppimisen toimintamallista [9].

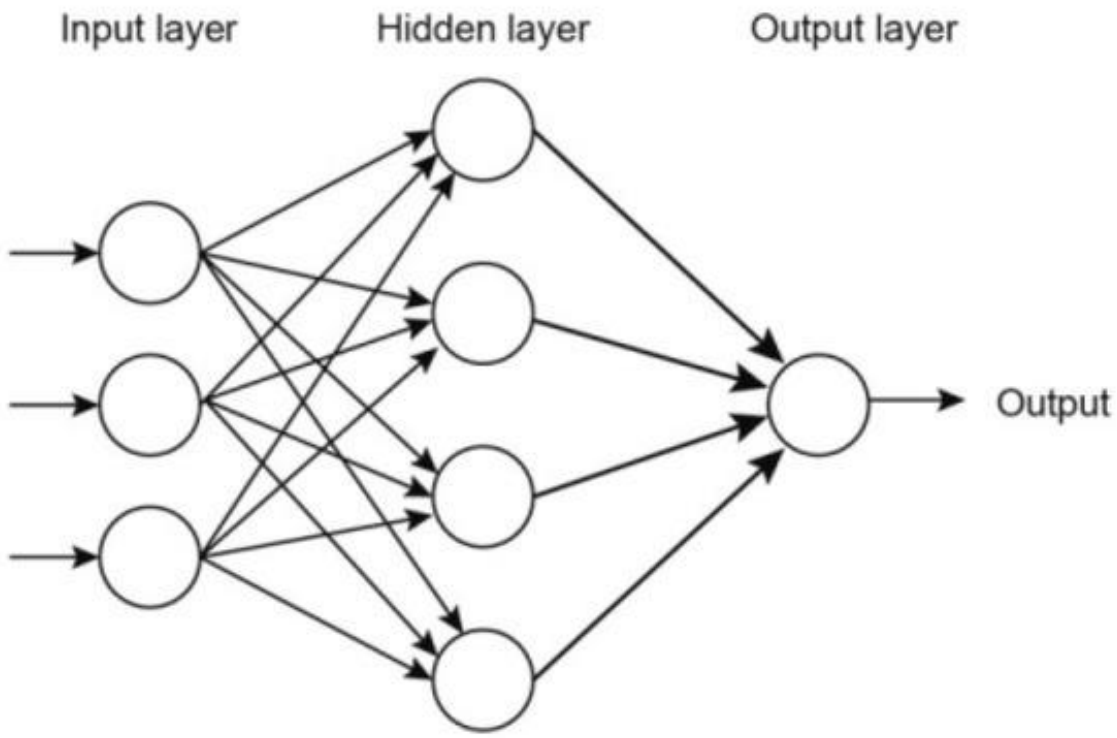
Pelihahmo hyökkää vihollista kohti ja osuttuaan laskee vihollisen elämäpisteitä ja pelihahmo saisi tästä toiminnosta ”palkinnon”. Vaihtoehtoisesti, jos pelihahmo yrittää juosta pakoon ja vihollinen pääsee tällöin hyökkäämään pelihahmon kimppuun, tehdystä toiminnasta rangaistaisiin pelihahmoa. [9.]

Neuroverkot

Neuroverkot imitoivat toiminnallaan ihmisen aivojen hermosolujen, neuronien, toimintaa. Neuroverkot käyttävät toiminnassaan koneoppimisen periaatteita neuroverkon muodostamiseen. Neuroverkot ovat yksi ohjatun oppimisen malleista. [10, s. 238–239.]

Neuroverkkojen toimintaa voi ajatella niin, että jokaisella sisään tulevalla arvolla on oma painonsa. Nämä arvot menevät sisälle ”neuroniin”, jossa arvot kerrotaan omalla painollaan. Funktio tämän ”hermon” sisällä summaa uudet painolla kerrotut arvot. Tämä summattu arvo toimii ”aktivointiarvona”, ja mikäli aktivointiarvo ylittää tietyn kynnyksen, esimerkkinä vaikka luku yksi, ”neuronista” lähtee signaali ja ulossyöttönä luku yksi. Mikäli aktivointiarvo on alle luvun yksi, tai mikä tahansa luku kynnykseen on asetettu, tulee neuronista ulos nolla. [10, s. 238–239.]

Kuvassa 4 on visualisoitu neuroverkkojen toiminta ja toiminnan eri tasot.



Kuva 4. Esimerkki neuroverkkojen toiminnasta [11].

Kun neuroverkon ja päätöspuun toimintaa verrattiin saman opetusdatan avulla, huomattiin neuroverkon oppivan tietoa hyvin, vaikka ei ehkä yhtä nopeasti kuin jotkin toiset algoritmit. Neuroverkot ovat kuitenkin kykeneviä monimutkaisempiin tehtäviin, vaikka verkon rakenteen saaminen oikeanlaiseksi voikin vaatia erityistä säätelyä. [3, s. 736.]

Päätöspuut

Päätöspuu on tehokas ja yleinen ratkaisu luokitteluun ja ennakointiin. Päätöspuu toimii prosessikaaviona, joka muodostaa puumaisen mallin. Puussa on solmukohtia, joissa ilmaistaan jotain vertailua jollekin ominaisuudelle. Solmusta lähtevät oksat esittävät vertailun lopputuloksia, ja jokainen lehti esittää seurausta ominaisuudelle. [12.]

Päätöspuun vahvuuksia on luoda selkeät ja ymmärrettävät säännöt toiminnalle, eikä luokittelu usein vaadi paljon laskentatehoa. Rakenteensa vuoksi päätöspuusta on myös helppo havainnoida tärkeimmät osa-alueet ennakoinnille ja luokittelulle. Tämän toiminnan heikkouksia taas on esimerkiksi sopimattomuus arvioiden tekemiseen, kun tavoite on ennustaa arvo ominaisuudelle, jolla voi olla lukemattoman monta eri arvoa.

Päätöspuu voi myös tehdä herkästi virheitä, mikäli opetusesimerkkejä on annettu vähän. Vastaavasti opetus voi käydä laskentatehon kannalta kalliiksi. [12.]

2.3 Tekoälyn käyttö muissa videopeleissä

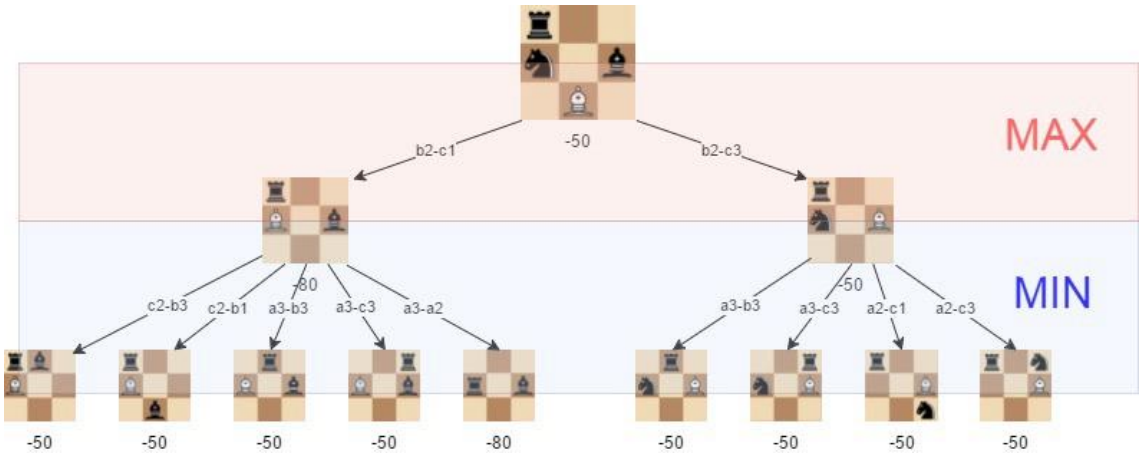
Videopeleissä käytetään harvoin todellista tekoälytekniikkaa, sillä se ei usein ole tarpeen videopeleissä halutun toiminnan aikaansaamiseksi. Monia tekoälyn toimintaperiaatteita voidaan silti hyödyntää ja mukauttaa eri tarkoituksiin videopeleissä. Näistä esimerkkejä ovat muun muassa ei-pelattavien hahmojen toiminta ja polunetsintä. [2.] Tekoälyä voi myös käyttää simuloimaan vastustajaa digitaalisissa lautapeleissä, kuten shakki.

Shakki

Shakki on yleinen esimerkki puhuttaessa tekoälyistä. Tietokone toimii vastustajana monessa tietokoneella pelattavassa lautapelissä, ja tekoäly on avainasemassa näissä peleissä. Toimivan shakkitekoälyn luomisessa on aloitettava itse pelin ymmärtämisestä: miten pelinappulat voivat liikkua pelilaudalla, mitkä pelinappulat ovat arvokkaampia pelin kannalta ja mikä pelin päämäärä on. [13.]

Sen sijaan, että tekoäly valitsisi satunnaisesti minkä tahansa sallitun siirron pelissä, on pelinappulat syytä painottaa. Näin eri nappuloiden tärkeydestä saadaan arvo, jota voidaan hyödyntää siirtojen valinnassa. [13.]

Parhaan siirron selvittäminen tehdään Minimax-algoritmilla. Kuva 5 näyttää kuvitettuna Minmax-algoritmin toimintaa. Tässä algoritmissa tarkastetaan rekursiivinen puurakenne kaikista mahdollisista siirroista valittuun syvyyteen asti. Jokainen positio evaluoidaan puurakenteen lopussa olevissa lehdissä, minkä jälkeen palautetaan joko pienin tai suurin arvo, riippuen siitä,kumman pelaajan vuoro on. [13.]



Kuva 5. Minimax-algoritmin visualisointi [13].

Algoritmin toiminnan optimointiin käytetään alfa-beetakarsintaa, jossa ohjelma karsii tietyt oksat puurakenteesta. Tällöin algoritmin syvyyttä voidaan lisätä käyttämättä enempää laskentaresursseja. Alfa-beetakarsinnassa voidaan jättää tietty osa puurakenteesta evaluoimatta, mikäli huomataan sen johtavan aikaisempaa huonompaan lopputulokseen. Näillä tekniikoilla saadaan luotua suhteellisen hyvin toimiva tekoäly shakin pelaamiseen. [13.]

Pac-Man

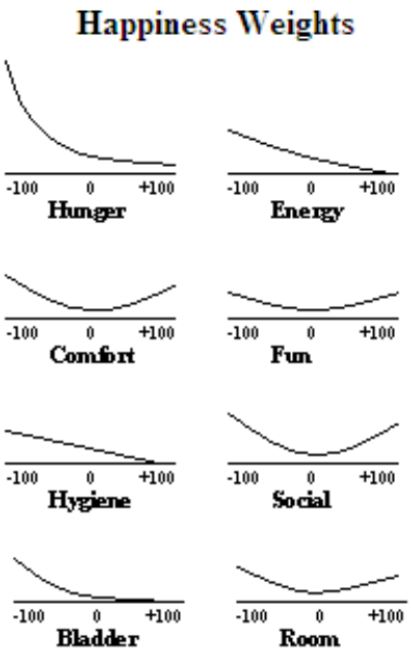
Pac-Man-pelissä vihollishahmot, neljä aavetta, toimivat pelaajaa vastaan pelissä jahtaamalla pelaajaa. Välillä viholliset taas jättävät pelaajan rauhaan. Aaveilla on eri toimintavaiheita, jotka vaihtuvat pelin aikana. Tämän lisäksi jokaisella aaveella on oma yksilöllinen toimintatapansa.

Punainen aave jahtaa suoraan pelaajan perässä, mutta vaaleanpunainen aave yrittää jahdatessaan päästä pelaajan edelle. Muiden aaveiden toiminnasta ei ole peliyhteisössä päästy täyteen yhteisymmärrykseen. On kuitenkin havaittavissa yhteisymmärrys siitä, että aaveiden tekoälytoiminta on olennainen, ellei olennaisin, osa koko pelikokemusta. [14.]

The Sims

The Sims simulaatiopelisarjassa ohjataan ihmispelihahmoja, Simejä, joilla on pelistä riippuen eri määrä tarpeita, joiden mukaan hahmot tekevät itsenäisiä päätöksiä pelissä. Pelaaja voi myös itse ohjata hahmot tekemään haluamiansa asioita ja poistaa pelihahmoilta ”vapaan tahdon” eli tekoälyn ohjaaman toiminnan. [15.] Tekoälytoteutuksen kannalta tämä lähestymistapa on lähinnä sitä, mitä tässä insinöörityöprojektissa pyritään luomaan.

Simeillä on fyysisiä ja henkisiä tarpeita ja kaikkia tarpeita täyttämään eri toimintoja ja ominaisuuksia. Kuvassa 6 näytetään pelaajalle tiedossa olevat tarpeet The Sims 3 -pelissä, joita pelin aikana voi seurata. Nämä tarpeet vaikuttavat Simin mielialaan ja onnellisuuteen. Tarpeita ovat esimerkiksi nälkä, hygienia, mukavuus, sosiaalisuus ja hupi. Tarpeet vaikuttavat mielialaan eri tavalla, esimerkiksi pieni näläntunne ei vielä saa aikaan suuria muutoksia, mutta nälän kasvaessa se alkaa vaikuttaa negatiivisesti Simin mielialaan. [15.]



Kuva 6. Tarpeiden vaikutus mielentilaan käyrinä The Sims 3 -pelissä [15].

Toiminnan ei haluta olevan liian täydellistä, ja siksi ei suoraan valitakaan tehtäväksi toimintoa, joka parantaisi mielentilaa eniten. Valinta tehdään satunnaisesti neljän eniten mielihyvää tuottavan toiminnan välillä. Tätä voidaan kehittää vielä laskemalla hyöty todennäköisyydeksi. [15.]

3 Tekoälytoteutuksen suunnittelu peliprojektiin

3.1 Toteutustapojen vertailu ja valinta

Tekoälyn toteutustavan valinnassa on olennaista, että halutut ominaisuudet ovat tiedossa. Tämä vaatii tarkan ja loppuun asti viedyn suunnittelun, sillä toivotut ominaisuudet voivat vaikuttaa suuresti siihen, mitä pelitekoälytoteutusta on tilanteessa tehokkainta käyttää. Projektissa oli tarkoitus luoda suhteellisen kevyt, mutta toimiva ratkaisu, jolloin sen ei ole tarpeen olla monimutkainen tai ohjelmana kuluttava.

Toteutustapaa valittaessa tarkastellaan erityisesti, miten eri tekoälyratkaisut kykenevät toteuttamaan eri toimintoja. Pelihahmolle on tyypillistä suorittaa erilaisia toimintoja, ja tässä projektissa erityisesti hahmon oli oltava mahdollista kulkea pelimaailmassa, kerätä ruokaa ja paeta vihollista. Projektissa ohjattava hahmo oli zombie ja tekoälyn ohjaamat hahmot ihmisiä.

Tarvejärjestelmä

Tarvejärjestelmän (engl. utility system tai utility AI) toimintaperiaate perustuu tarpeeseen tehdä eri toimintoja. Tarpeet eivät suoraan ole numeerisia arvoja, vaan ne voidaan määrittää edustamaan eri toimintoja tilanteen mukaan. Tarve voisi olla esimerkiksi syöminen. Juuri ruokailun jälkeen olemme tavallisesti kylläisiä, eikä silloin ole uudestaan tarvetta syödä. Mikäli ruokailusta taas on pitkä aika, tarve on suurempi. Arvoja tarvitaan silti päätöksentekoon, sillä arvot antavat meille mahdollisuuden verrata eri toimintoja keskenään ja valita niiden väliltä. [16, s. 114.]

Päätöstenteko tarvejärjestelmässä tapahtuu kuten lähes jokaisessa tekoälytoteutuksessa: vertailemalla kaikkia mahdollisia toimintoja, jotka tekoälylle on asetettu. Toiminnot

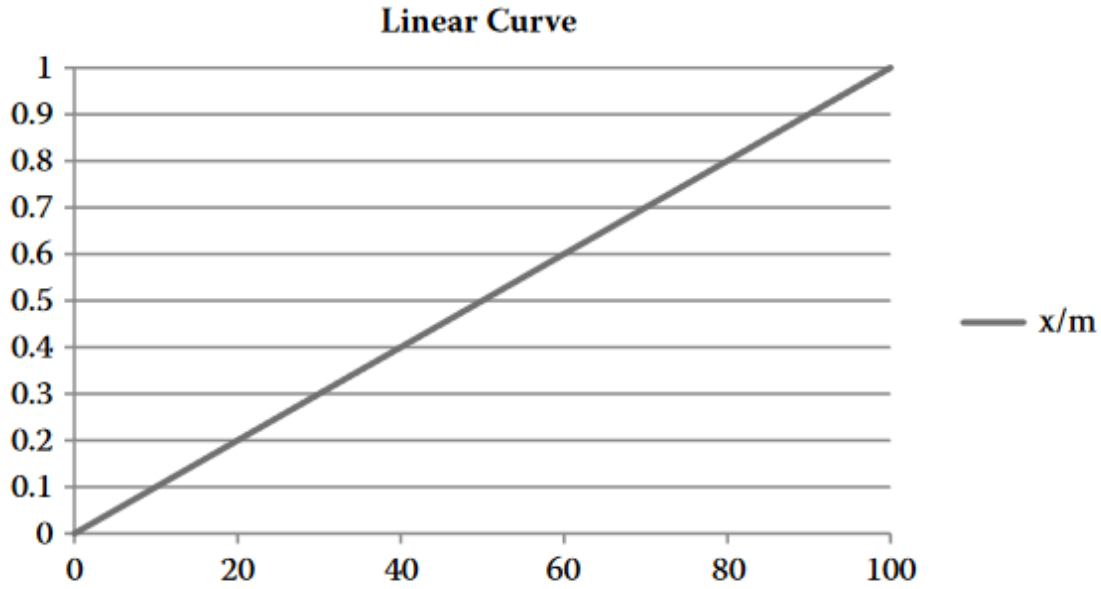
on usein painotettu ja laskemalla ja vertaamalla toimintojen arvoja valitaan suoritettavaksi toiminnoksi korkein saatu arvo. Yksi yleinen tapa painottaa tarvetta on kertoa se jokaisen mahdollisen lopputuloksen todennäköisyydellä ja summata painotetut luvut. [16, s. 114.] Tämän voi laskea esimerkiksi yhtälöllä 2.

$$EU = \sum_{i=1}^{n} D_i P_i \tag{2}$$

Yhtälössä D on lopputuloksen tarve ja P todennäköisyys, jolla lopputulos tapahtuu. Tämä todennäköisyys on normalisoitu, jotta kaikkien todennäköisyyksien summa on 1. Tätä sovelletaan kaikkiin mahdollisiin toimintoihin, jotka voi valita ja toiminta, jolla on korkein odotettu tarve, valitaan. [16, s. 115.]

Päätöstenteko on harvoin riippuvainen vain yhdestä tekijästä. Jos otetaan aikaisempi esimerkki syömisestä, syömisen tarpeeseen ja haluun ei aina vaikuta pelkästään kulunut aika viime ruokailusta. Haluun voi vaikuttaa tarjolla olevan ruoan miellyttävyys, ruoan valmistukseen kuluva aika ja ruoan terveellisyys. Esimerkkinä kuvitellaan tarjolla olevan ranskalaisia perunoita ja kasvissosekeittoa. Kahdella eri henkilöllä voi olla erilaiset mieltymykset näihin vaihtoehtoihin, ja tällöin heidän tarpeensa painottuisivat eri tavalla. Toinen henkilöistä saattaa arvostaa enemmän ruoan terveellisyyttä eikä välitä, jos ruoan valmistukseen kuluu enemmän aikaa. Toinen henkilö haluaa taas ruoan mahdollisimman nopeasti ja sattuu myös pitämään ranskalaisista.

Tarpeita voidaan mitata myös käyrien avulla. Kuvassa 7 on yksinkertainen lineaarinen käyrä, jonka x-akselilla on arvot nollasta sataan ja y-akselilla arvot nollasta yhteen. Käyrän voi luoda eri matemaattisten kaavojen avulla, mutta usein videopeliominaisuuksia luotaessa tarvitaan hienosäätöjä, joita voi olla hankala tai mahdoton luoda pelkästään matemaattisesti laskemalla. Tällöin mukautettu käyrä, jonka voi yksityiskohtaisesti muokata haluamakseen, on usein paras vaihtoehto. [16, s. 117.]



Kuva 7. Lineaarinen käyrä [16].

Tarpeiden arvot eivät yksinään tarkoita mitään, vaan on laskettava jokaisen muunkin vaihtoehdon tärkeys. Nämä laskennat on suoritettava ohjelman ja itse pelin ollessa käynnissä, sillä kaikki, mitä pelissä tapahtuu, vaikuttaa hahmon tarpeisiin. Tällä tavalla saadaan aikaan tekoäly, joka pystyy arvioimaan tilannetta hienovaraisesti ja toimimaan myös niin, ettei käyttäytyminen ole täysin mustavalkoista. [17.]

Kaksi tyypillistä tapaa lähestyä tarvepohjaista päätöksentekoa on ehdoton hyöty ja suhteellinen hyöty. Ehdoton hyöty toimii niin, että kaikki vaihtoehdot evaluoidaan ja valitaan aina kaikista korkein hyöty. Suhteellisessa hyödyssä valitaan hyöty satunnaisesti, käyttäen jokaisen vaihtoehdon hyötyä määrittämään todennäköisyys, jolla se valittaisiin. [18.] Yhtälössä 3 lasketaan satunnainen hyöty painotuksen avulla.

$$P_O = \frac{U_O}{\sum_{i=1}^{n} U_i} \tag{3}$$

Todennäköisyys (P) vaihtoehdon (O) valintaan pohjautuu jakamalla kyseisen vaihtoehdon hyödyn (U) kaikkien vaihtoehtojen hyötyjen loppusummalla. Tätä lähestymistapaa kutsutaan yleensä painotetuksi satunnaistamiseksi (engl. weight-based random tai weighted random).

Dual Utility Reasoner (DUR) yhdistää molemmat näistä lähestymistavoista. Siinä jokaiseen vaihtoehtoon liitetään kaksi hyötyarvoa: sija ja paino. Sija toimii ehdottoman hyödyn lailla ja paino suhteellisen hyödyn toiminnan mukaan. Käsitteellisesti sijaa käytetään jakamaan vaihtoehdot kategorioihin, ja vaihtoehtoja valitaan vain parhaasta kategoriasta. Painon avulla evaluoidaan vaihtoehdot kategorian sisällä. Vaihtoehdon painolla on tällöin merkitystä vain suhteessa muihin vaihtoehtoihin saman sijan kategoriassa. Tällöin vain parhaan kategorian vaihtoehtojen painolla on merkitystä. [17.]

Päätöksenteko aloitetaan laskemalla jokainen vaihtoehdon sija ja paino. Jokainen vaihtoehto, jonka paino on nolla, poistetaan valinnoista. Seuraavaksi etsitään korkein sija jäljellä olevista vaihtoehdoista, ja kaikki alemmalla sijalla olevat vaihtoehdot poistetaan valinnoista. Ajatuksena on löytää sopivin kategoria vaihtoehtoja. Tämän jälkeen poistetaan kaikki vaihtoehdot, joiden paino on huomattavasti alempi kuin parhaiden jäljellä olevien valintojen. Näin poistetaan mahdollisuus valita vaihtoehto, jonka valinta pelissä tuntuisi typerältä siinä tilanteessa. Tarkka suhde poistettavien valintojen välillä riippuu täysin tilanteesta. Lopulta painotetun satunnaisuuden avulla valitaan jäljellä olevista vaihtoehdoista jokin. [17.]

Halutuin mahdollinen toiminto voi riippua videopelin tyylistä. Joskus pelin kannalta paras ratkaisu on aina matemaattisesti korkein arvo, mutta joihinkin peleihin tämä ei välttämättä sovi. Kuvassa 8 näkyy The Sims 2 -pelin tarpeet ja niiden senhetkinen tilanne. Erityisesti simulaatiopeleissä voi tuntua keinotekoiselta, jos hahmo valitsee aina parhaan mahdollisen tavan toimia. Tällöin voi olla sopivampaa valita satunnaisesti eri toiminnoista, mitä suoritetaan. [16, s. 120.]

Kuva 8. The Sims 2 -pelin tarpeet [18].

Tarvejärjestelmä tarjoaa suhteellisen yksinkertaisella tavalla monimuotoisen käyttäytymismallin, jota voi lähteä syventämään monimutkaisemman tekoälyn luomiseksi. Insinöörityöprojektin toteutuksessa päädyttiin lopulta tarvejärjestelmään, sillä se tarjoaa hieman sulavamman toiminnan kuin tilakone. [17.]

3.2 Valitun toteutustavan tuomat rajoitukset

Tarvejärjestelmässä, kuten lähes missä tahansa muussakin pelitekoälytoteutuksessa, on kaikki halutut tarpeet ja toiminnot määriteltävä ohjelmaan erikseen. Tämä tarkoittaa, että mikäli hahmon toimintaan haluttaisiin myöhemmin lisätä uusia toimintatapoja ja tiloja, ne on lisättävä erikseen useaan eri ohjelmakoodiin. Tämä tekee toiminnan muuttamisen ja laajentamisen suhteellisen vaivalloiseksi.

Toinen mahdollinen ongelma, joka voi ilmetä lähes missä tahansa tekoälytoteutuksessa, on kykenemättömyys pitäytyä valinnassa. Mikäli kaksi tarvetta ovat samansuuruiset, tekoäly voi jumiutua jatkuvasti vaihtamaan näiden kahden valinnan välillä. Tätä on mahdollista välttää lisätyllä painotuksella valitussa tarpeessa tai esimerkiksi ajastamalla, milloin hahmo voi taas vaihtaa tilaansa. [16, s. 122.]

4 Tekoälyn toteutus hahmojoukolle

4.1 Työkalut ja algoritmit

Insinöörityön projektiosuus toteutettiin Unity-pelimoottorilla C#-ohjelmointikielellä. Tekoälyn luomiseen käytettiin tarvejärjestelmää ja tarvekäyrien luomiseen hyödynnettiin Unityn animaatiokäyriä.

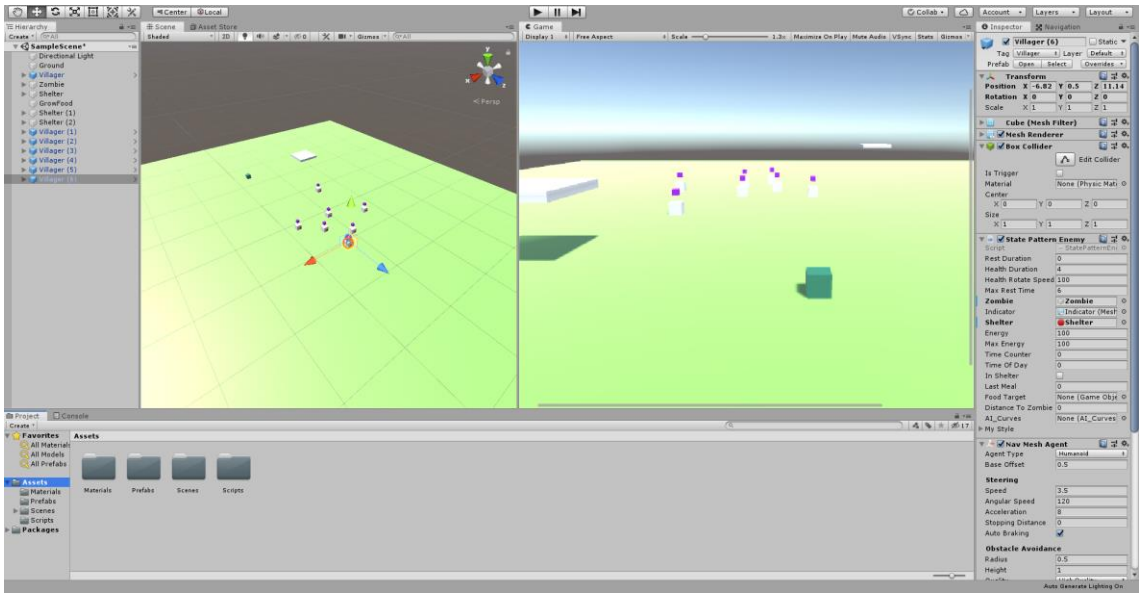
Unity-pelimoottori

Unity (aik. Unity3D) on vuonna 2005 Unity Technologies -yrityksen julkaisema pelimoottori, joka on noussut etenkin itsenäisten pelinkehittäjien suosioon saatavuutensa ja

laajojen ominaisuuksiensa ansiosta. Unity-pelimoottorilla on mahdollista luoda 2D- ja 3D-peliprojekteja sekä hyödyntää lisättyä todellisuutta ja virtuaalitodellisuutta. [19.]

Unity-moottoria kehitetään jatkuvasti, ja uusien versioiden myötä siihen on lisätty uusia ominaisuuksia, jotka monipuolistavat ja helpottavat pelinkehitystä Unitylla. Erityisesti Unity on mahdollistanut taiteellisia ja luovia peliprojekteja, kun pelinkehittäjien ei ole tarvinnut luoda teknologiaa itse. Unity-ohjelman sisällä voi luoda kaiken mitä peliprojekti vaatii: animaatiot, pelin fysiikat, äänet sekä tekoälyn. [19.]

Kuva 9 on kuvakaappaus Unity-pelimoottorin visuaalisesta käyttöliittymästä, jossa suurin osa pelin luomisesta tapahtuu. Vaikka pelin luominen Unitylla ilman ohjelmointia on teoriassa mahdollista moottorin ”raahaa ja pudota” -ominaisuuden ansiosta, monet tarkemmat ominaisuudet on ohjelmoitava erikseen. Peliobjektit voi luoda visuaalisella käyttöliittymällä muutamaa nappia painamalla, minkä jälkeen objektiin voidaan lisätä skripti, ohjelmakoodi, joka kertoo objektille sen toiminnan. [20.]



Kuva 9. Unity-pelimoottorin käyttöliittymä.

C#-ohjelmointikieli

C#-ohjelmointikieli on Microsoftin vuonna 2000 julkaisema ohjelmointikieli, ja se on suunniteltu toimimaan osana .NET -viitekehysjärjestelmää. Ohjelmointikielenä C# on olio-orientoitunut. Olio-ohjelmoinnissa tieto on sidottu kokoelmaan, ”olioon”, joka voivat käsitellä myös toisien olioiden hallinnoimaa tietoa. Tyypillisiä piirteitä olio-ohjelmoinnille, joita C# myös tukee, ovat perintä, kapselointi ja polymorfismi eli monimuotoisuus. [21.]

C# muistuttaa syntaksiltaan paljon C/C++ -ohjelmointikieliä. C#-kielen suunnittelussa pyrittiin vain tarpeen mukaan laajentamaan C++-kielen ominaisuuksia. C# käyttää ohjelmointiluokkia. Luokka on eräs oliotyyppi, ja oliot ovat luokasta tehtyjä käytettäviä instansseja. Olion luomista kutsutaan ilmentämiseksi (engl. instantiation). [22.] Esimerkkikoodissa 1 esitetään luokan alustaminen C#-kielellä.

```
class EsimerkkiLuokka
{
}
```

Esimerkkikoodi 1. Luokan määrittäminen C#-kielellä.

C#-ohjelmakoodeja käytetään Unityssä peliobjektien toiminnan määrittelyyn, sillä peliobjekti itsessään ei tee mitään, mitä sitä ei ohjelmoida tekemään. C#-ohjelmointi Unityssa eroaa hieman puhtaasta ohjelmoinnista, sillä Unity tekee osan työstä itse. Pelinkehittäjän ei esimerkiksi tarvitse kirjoittaa koodia, joka suorittaa ohjelmaa, vaan hän voi keskittyä täysin pelin toiminnan kannalta olennaiseen ohjelmointiin. [23.]

Yleisimmin käytetyt ominaisuudet ohjelmoitaessa ovat muuttujat, luokat ja funktiot. Nämä ovat myös Unityn C#-ohjelmoinnissa tärkeimmät ja käytetyimmät ominaisuudet. Muuttujat pitävät sisällään arvoja ja referenssejä olioihin. Muuttujien nimet kirjoitetaan pienellä alkukirjaimella. Funktiot tai metodit ovat koodilajitelma, jossa verrataan tai muutetaan ohjelman muuttujia. Koodin lajitteleminen funktioihin tekee ohjelman rakenteesta selkeämmän, ja funktioita on helppo kutsua uudestaan koodissa. Luokat ovat myös tapa järjestää ohjelmakoodia. Luokkaan saa paketoitua lajitelman muuttujia ja funktioita uudelleenkäyttämistä varten. [23.]

Tarvejärjestelmä

Insinöörityön tekoäly toteutettiin aikaisemmin mainitulla tarvejärjestelmällä. Tarpeiden käyrien muodostamiseen hyödynnetään Unityn sisältämää animaatiokäyrää, joiden arvoihin voi viitata ohjelmakoodin sisällä.

Tarvejärjestelmän luomisella on tyypillisesti kolme toimenpidettä:

- vaihtoehtojen listan rakentaminen
- vaihtoehtojen evaluointi ja vaihtoehtoa kuvaavan arvon laskeminen
- vaihtoehdon valinta laskettujen arvojen perusteella.

Olennaisinta on, että evaluointi tehdään ohjelman, eli tässä tapauksessa pelin, suorituksen aikana. Arvoja ei tule määrittää etukäteen vaan ne on laskettava ajon aikana saatujen tietojen mukaan. Tarpeita edustavat arvot muuttuvat jatkuvasti ja tällöin myös sopivin toiminta. Arvot on syytä määrittää liukulukumuuttujille eikä esimerkiksi boolean-muuttujille, sillä liukuluku tarjoaa enemmän rakeisuutta. [17.]

4.2 Hahmojoukon toiminta

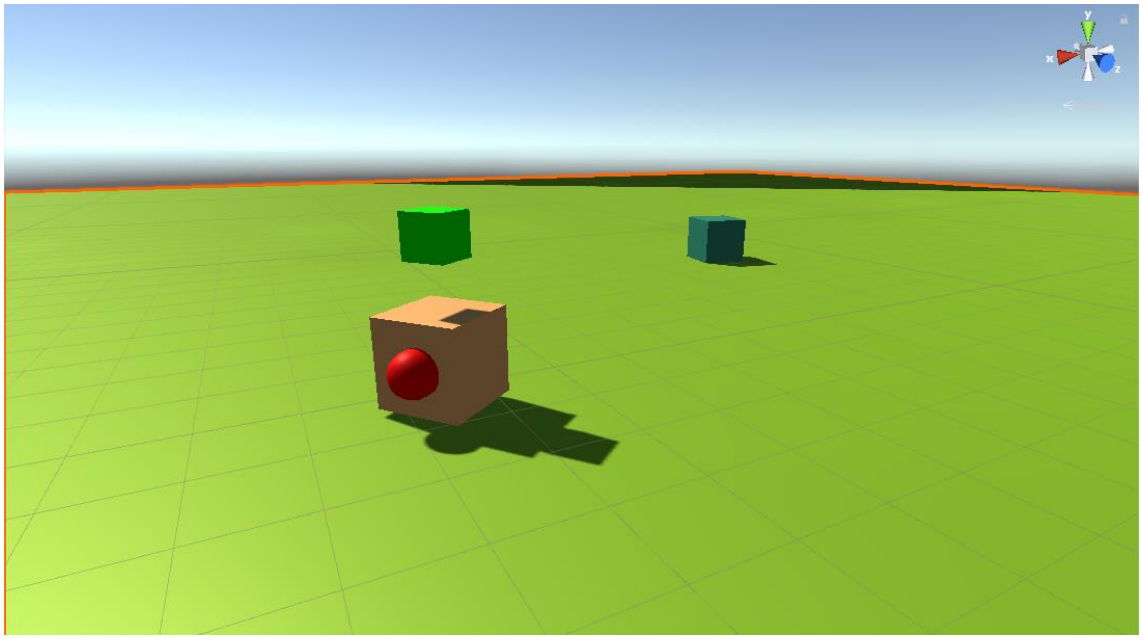
Hahmojoukon toiminta perustuu neljään eri tilaan, jossa hahmo voi olla pelin aikana. Jokainen tila määritetään omassa luokassaan, joista jokainen perii saman rajapinnan. Rajapinnassa on alustettu funktiot, joita jokainen tila tarvitsee toimiakseen. Näiden tilojen välisiä siirtymiä hallinnoi oma, tälle prosessille omistettu ohjelmakoodi.

Normaalitila

Pelihahmo on normaalitilassa, kun mikään muu tarve ei ole noussut kovin ylös. Normaalitilassa hahmo liikkuu satunnaisesti ympäri pelikenttää, kunnes siirtyy johonkin toiseen tilaan. Hahmon energia ja ravitsemus vähentyvät tasaisesti koko ajan myös normaalitilassa. Hahmon päälle on lisätty kuutio indikaattoriksi. Indikaattorin väri muuttuu aina eri tilassa, jota tilamuutokset ovat selkeämpiä havainnoida visuaalisesti. Normaalitilassa tämä indikaattori on vihreä.

Normaalitilan luokka perii IEnemyState-rajapinnan, jossa on alustettu funktiot tilojen päivittämiselle ja siirtymisiin tilojen välillä sekä OnTriggerEnter(Collider other) -metodi, jossa tarkistetaan osutaanko jonkin peliobjektin törmäyttimeen (engl. collider). Jokainen tilaluokka perii tämän saman rajapinnan ja sen sisällä olevat funktiot.

Kuvassa 10 nähdään, miltä tekoälyllä ohjatun pelihahmon normaalitila näyttää itse pelitilanteessa. Normaalitilaluokan alussa alustetaan yksityinen muuttuja StatePatternEnemyn luokasta, joka antaa pääsyn kyseisen luokan muuttujiin. Näitä tarvitaan pelihahmon, tässä projektissa kyläläisen hallintaan. Tässä kohtaa alustetaan myös muut normaalitilassa käytettävät muuttujat: liukulukumuuttuja eatTimer ja boolean-muuttujat haveTarget ja eating. Seuraavana luokassa on tyhjä OnTriggerEnter-metodi. Normaalitilassa ei suoriteta mitään toimintoja muihin peliobjekteihin törmätessä, mutta tähän voitaisiin haluttaessa esimerkiksi ohjelmoida hahmo saamaan tai tekemään vahinkoa.



Kuva 10. Pelihahmo (vas.) normaalitilassa.

Tämän jälkeen on metodit kaikkiin muihin tiloihin siirtymiselle sekä UpdateState-metodi, jossa kutsutaan normaalitilan toimintoa: kävelyä. Esimerkkikoodissa 2 esitetään pelihahmon kävelyfunktio.

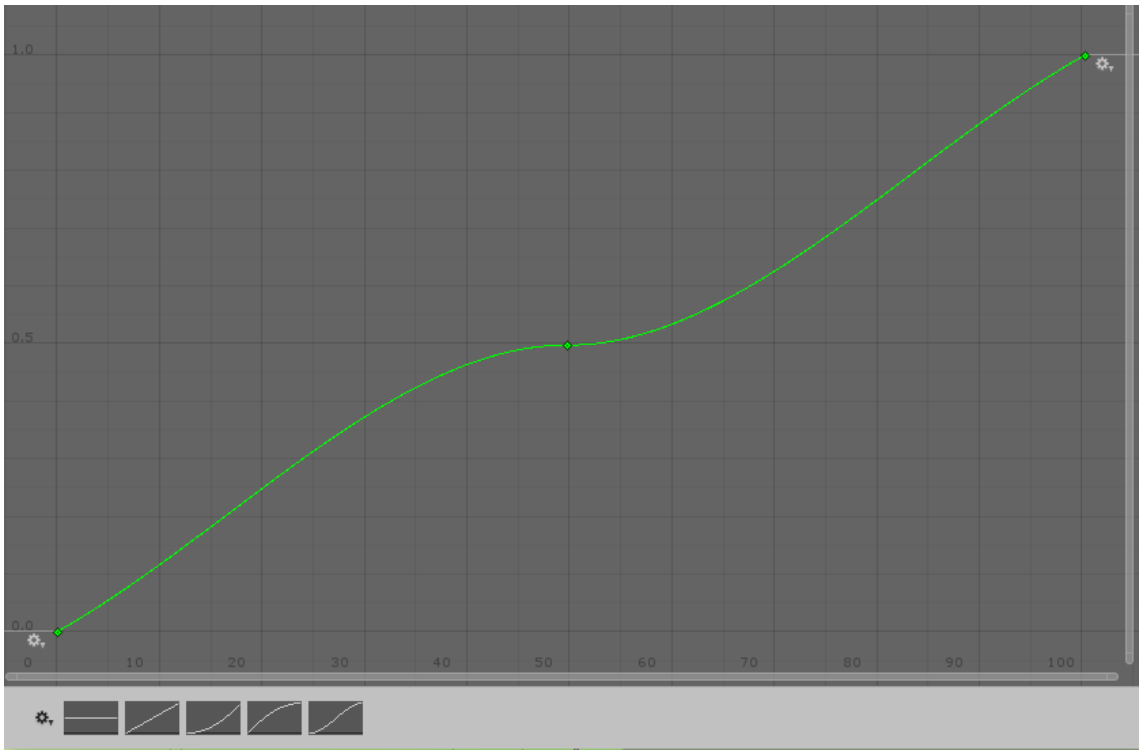
```
void Walk()
{
      villager.indicator.material.color = Color.green;
      villager.navMeshAgent.speed = 1;

      if (villager.navMeshAgent.remainingDistance <=
      villager.navMeshAgent.stoppingDistance &&
      !villager.navMeshAgent.pathPending)
      {
            villager.navMeshAgent.destination = villager.transform.position +
            new Vector3(Random.Range(-5, 5), 0, Random.Range(-5, 5));
      }

      villager.navMeshAgent.isStopped = false;
}
```

Esimerkkikoodi 2. Normaalitilassa suoritettava kävelyfunktio.

Kävelyfunktiossa ensin muutetaan hahmon indikaattorin materiaalin väri vihreäksi. Tällä ei ole pelin toiminnallisuuden kannalta mitään merkitystä, mutta se helpottaa tilojen vaihtumisen huomaamista testausvaiheessa.

Seuraavalla rivillä hahmolle annetaan nopeus. Hahmoon on liitetty Unityn Navigation Mesh Agent, Unityn oma sisäinen navigointijärjestelmä, joka auttaa hahmoja liikkumaan oikealla alueella ja tarvittaessa väistämään muita peliobjekteja. Seuraava if-lause tarkistaa, onko hahmon päämäärään jäljellä oleva matka pienempi tai yhtä suuri kuin pysähtymismatka päämäärästä. Samanaikaisesti varmistetaan, ettei hahmolle prosessoida polkua minnekään. Kun nämä ehdot täyttyvät, hahmolle määritetään uusi satunnainen päämäärä. Lopuksi todetaan vielä, että hahmo ei ole pysähtynyt. Tätä kävelyfunktiota suoritetaan jatkuvasti hahmon ollessa normaalitilassa.

Käyrät tarpeiden mittaamiseen toteutettiin Unityn animaatiokäyrillä, joita evaluoidaan ja näin saadaan ajan määrittämä arvo käyrältä. Kuva 11 näyttää hahmon normaalitilalle luodun käyrän. Arvoja verrataan korkeimman tarpeen saamiseksi.



Kuva 11. Käyrä pelihahmon normaalitarvetilasta.

Ravitsemustila

Ravitsemustila mittaa pelihahmon tarvetta kerätä ruokaa pelimaailmassa. Ravitsemuksen tarvetta lasketaan eri tiloja hallinnoivassa luokassa. Esimerkkikoodi 3 näyttää laskentakaavan pelihahmon ravitsemustarpeelle.

```
float urge = (100 - health) / 2 + lastMeal;
```

Esimerkkikoodi 3. Ravitsemustarpeen laskenta.

Ravitsemustarpeen laskennassa ensin lasketaan x:n arvo ravitsemuskäyrältä, 100 on maksimi arvo käyrän x-akselilla ja energy-muuttuja on ajan kuluessa vähentyvä arvo. Tällä saadaan silloisen ajan arvo x-akselilta ravitsemuskäyrällä. Tämä arvo jaetaan kahdella, ja sitten summataan aika viimeisestä syödystä ateriasta. Tämän laskun tilalla voisi olla mikä tahansa muukin lasku, jolla parhaiten saadaan laskettua kyseisen tilanteen tarve. Tämä laskenta suoritetaan StatePatternEnemy-luokassa.

Ravitsemustilaluokan alussa alustetaan myös StatePatternEnemyn luokka sekä OnTriggerEnter ja kaikkien muiden tilojen siirtymät. Luokan alussa alustetaan myös muuttujat eatTimer, haveTarget ja eating. Ravitsemustilassa OnTriggerEnter-metodissa on toiminta. Metodi saa parametrina jonkin toisen peliobjektin törmäyttimen ja tässä kohtaa tarkastetaan, onko tämän törmäyttimen merkki (engl. tag) ”Food”. Seuraava toiminto suoritetaan vain, kun kyseessä on Food-merkitty peliobjekti. Esimerkkikoodi 4 näyttää ravitsemuskohteen valintalogiikan.

```
if(other.CompareTag("Food"))
{

      GameObject[] villagers = GameObject.FindGameObjectsWithTag("Villager");
      foreach(GameObject singleVillager in villagers)
      {
            if(singleVillager.GetComponent<StatePatternEnemy>().foodTarget ==
                  other.gameObject)
            {
                  singleVillager.GetComponent<StatePatternEnemy>().foodTarget =
                  SearchNewEnergyTarget();
            }
      }
}
```

Esimerkkikoodi 4. Ohjelmakoodi ravinnonkohteen valintaan.

Ensimmäiseksi tehdään lista kaikista ”Villager”-merkityistä peliobjekteista. Tämän jälkeen iteroidaan jokainen yksittäinen peliobjekti tällä listalla ja tarkistetaan hahmon StatePatternEnemy-luokan muuttujista, onko ravintokohde sama kuin se, jonka törmäyttimeen on osuttu. Mikäli on, ajetaan metodi, joka etsii uuden ravinnonkohteen.

Ravitsemusluokan UpdateState-metodi ajaa Eat-metodin, jossa suoritetaan ravitsemustarpeen täyttäminen. Kuten normaalitilassakin, ravitsemustilassa muutetaan pelihahmon indikaattorin väriä tilan vaihtuvuuden selkeyttämiseksi. Toiminta pelihahmon ruokailulle esitetään esimerkkikoodissa 5.

```
if(haveTarget == false)
{
      villager.foodTarget = SearchNewEnergyTarget();
      villager.navMeshAgent.destination
      = villager.foodTarget.transform.position;

      villager.navMeshAgent.speed = 3;
      villager.navMeshAgent.isStopped = false;
      haveTarget = true;
}

if(villager.navMeshAgent.remainingDistance
<= villager.navMeshAgent.stoppingDistance
&& !villager.navMeshAgent.pathPending && eating == false)
{
      villager.navMeshAgent.isStopped = true;
      eatTimer = 0;
      eating = true;
}

if(eating == true)
{
      if(eatTimer >= villager.healthDuration)
      {
            villager.lastMeal = villager.lastMeal / 4;
            villager.health += 20;
            eating = false;
            haveTarget = false;
            if (villager.health > villager.maxHealth)
            {
                  villager.health = villager.maxHealth;
            }
      }

      villager.transform.Rotate(0, villager.healthRotateSpeed
      * Time.deltaTime, 0);

      eatTimer += Time.deltaTime;
      Object.Destroy(villager.foodTarget);
}
```

Esimerkkikoodi 5. Eat-metodin toiminta.

Aluksi varmistetaan, onko ravinnolle kohdetta. Kun kohdetta ei ole, ajetaan metodi sen etsimiselle. Tämän jälkeen asetetaan pelihahmon päämääräksi ravinnonkohteen positio pelimaailmassa. Pelihahmon liikkumisen nopeutta myös nostetaan hieman, ja samalla määritetään pelihahmo liikkuvaksi. Vahvistetaan myös, että pelihahmolla on kohde.

Seuraavassa if-lausekkeessa katsotaan, että pelihahmo on saapunut kohteelle, hahmolla ei ole enää liikkumiselle määränpäätä, eikä hahmo tällä hetkellä syö. Kun ehdot ovat totta, pysäytetään hahmo, nollataan eatTimer-muuttuja ja muutetaan eating-muuttuja todeksi. Jos eating-muuttuja on totta ja eatTimer-muuttuja pienempi tai yhtä suuri kuin hahmon healthDuration-muuttuja, muutetaan lastMeal-muuttujaa jakamalla sen

neljällä, ja lisäämällä hahmon ravitsemusta ja muutetaam eating- ja haveTarget muuttuja epätodeksi. Varmistetaan vielä if-lauseessa, ettei hahmon ravitsemus voi mennä yli siitä, mitä on asetettu ravitsemuksen enimmäisarvoksi. Viimeisenä pelihahmo asetetaan pyörimään syömisen ajaksi, ja ihan viimeisenä tämä ravinnonkohdeobjekti poistetaan.

Ravitsemusluokan viimeinen metodi on SearchNewEnergyTarget, joka suoritetaan muutamaan otteeseen luokassa aikaisemmin. Tämä metodi etsii pelimaailmasta lähinnä pelihahmoa olevan ravinnonkohteen ja palauttaa tämän peliobjektin. Esimerkkikoodi 6 esittää toimintaperiaatteen lähimmän ravinnonlähteen hakemiselle pelimaailmassa.

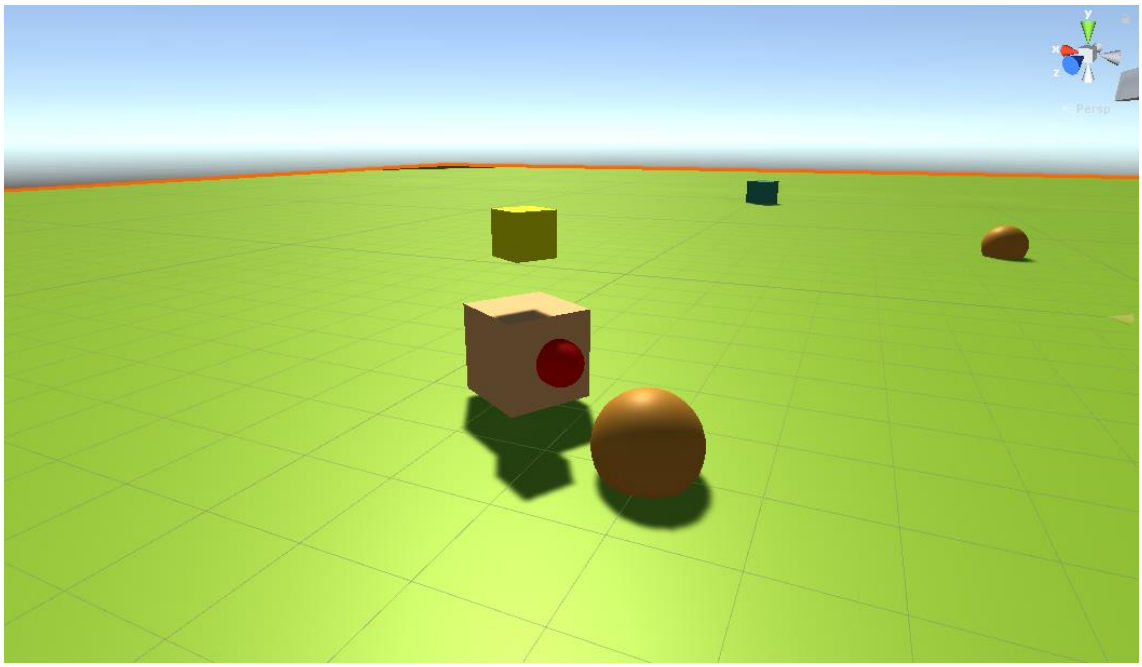
```
private GameObject SearchNewEnergyTarget()
{
      float shortestDistance = Mathf.Infinity;

      GameObject[] food = GameObject.FindGameObjectsWithTag("Food");

      if(food.Length > 0)
      {
            GameObject closestEnergy = food[Random.Range(0, food.Length)];

            for (int i = 0; i < food.Length; i++)
            {
                  float distance = Vector3.Distance(villager.transform.position,
                        food[i].transform.position);
                  if(distance < shortestDistance)
                  {
                        shortestDistance = distance;
                        closestEnergy = food[i];
                  }
            }
            return closestEnergy;

      }
      else
      {
            villager.health -= 5;
            if (villager.health <= 0)
            {
                  villager.health = 0;
            }
            return null;
      }
}
```

Esimerkkikoodi 6. Lähimmän ravinnon löytäminen.

Ensimmäisenä määritetään muuttuja shortestDistance. Muuttujalle alustetaan aluksi positiivinen ääretön luku Unityn matematiikkafunktiolla, ja oikea lyhin etäisyys lasketaan myöhemmin for-silmukassa. Seuraavaksi luodaan lista kaikista ”Food”-merkityistä peliobjekteista, ja mikäli näitä objekteja on enemmän kuin nolla, iteroidaan tämä lista. Aluksi

annetaan closestEnergy-muuttujan arvoksi satunnainen ravinnonkohde, sillä muuttuja on alustettava jollakin arvolla. Myös tämän muuttujan oikea objekti saadaan seuraavassa for-silmukassa.

Tämä on tyypillinen for-silmukka, jossa indeksi i on nolla, ja sitä kasvatetaan yhdellä ravinnonkohteista muodostetun listan pituuden verran. Jokaisen iteroinnin kohdalla tarkistetaan, onko etäisyys pelihahmon ja ravinnonkohteen välillä lyhyempi kuin alkuun määritelty shortestDistance-muuttuja, ja aina kun näin on, muutetaan shortestDistance-muuttujan arvoksi tämä lyhyempi etäisyys ja samalla lähimmäksi kohteeksi kyseinen peliobjekti. Listan iteroinnin jälkeen palautetaan pelihahmoa lähinnä oleva ravinnonkohde-peliobjekti. Kuvassa 12 näkyy pelitilanne, jossa tekoälyn ohjaama hahmo on menossa ravinnonlähteen luo.



Kuva 12. Pelihahmo menossa ravintokohteelle ravitsemustilassa.

Mikäli pelimaailmassa ei ole yhtään ravinnonkohteita, vähennetään pelihahmon ravitsemusta eikä palauteta kohdetta. Mikäli ravitsemus menisi vähennyksen takia alle nollan, palautetaan arvo nollaksi.

Lepotila

Kuten aikaisemmissakin tiloissa, lepotila perii IEnemyState-rajapinnan ja siellä alustetut metodit: OnTriggerEnter, siirtymät tilojen välillä ja UpdateState. Myös tässä luokassa alustetaan StatePatternEnemy-pelihahmo. Lepotilan muuttujia ovat resting, haveTarget ja restTimer, jotka alustetaan luokan alussa. Tässä luokassa OnTriggerEnter ei suorita mitään toimintoja. UpdateState kutsuu Rest-metodin. Lepotilan Rest-metodin toiminta näytetään esimerkkikoodissa 7.

```
villager.navMeshAgent.speed = 1;
villager.indicator.material.color = Color.cyan;

if(Vector3.Distance(villager.shelter.transform.position,
      villager.transform.position) > 30 && haveTarget == false)
{
      villager.navMeshAgent.isStopped = true;
      resting = true;
}
      else
{
      if(villager.inShelter == false)
      {
            villager.navMeshAgent.destination =
            villager.shelter.transform.position;

            villager.navMeshAgent.isStopped = false;
            haveTarget = true;
      }
}

if (villager.navMeshAgent.remainingDistance <=
      villager.navMeshAgent.stoppingDistance &&
      !villager.navMeshAgent.pathPending)
{
      villager.navMeshAgent.isStopped = true;
      haveTarget = false;
      resting = true;
}

if(resting == true && villager.inShelter == true)
{
      restTimer += Time.deltaTime * 5;
      villager.energy += restTimer;
}
else if (resting == true && villager.inShelter == false)
{
      restTimer += Time.deltaTime * 3;
      villager.energy += restTimer;
}
if (villager.energy >= villager.maxRestTime)
{
      villager.energy = villager.maxEnergy;
      ToNormalState();
}
```

Esimerkkikoodi 7. Rest-metodin toiminta.

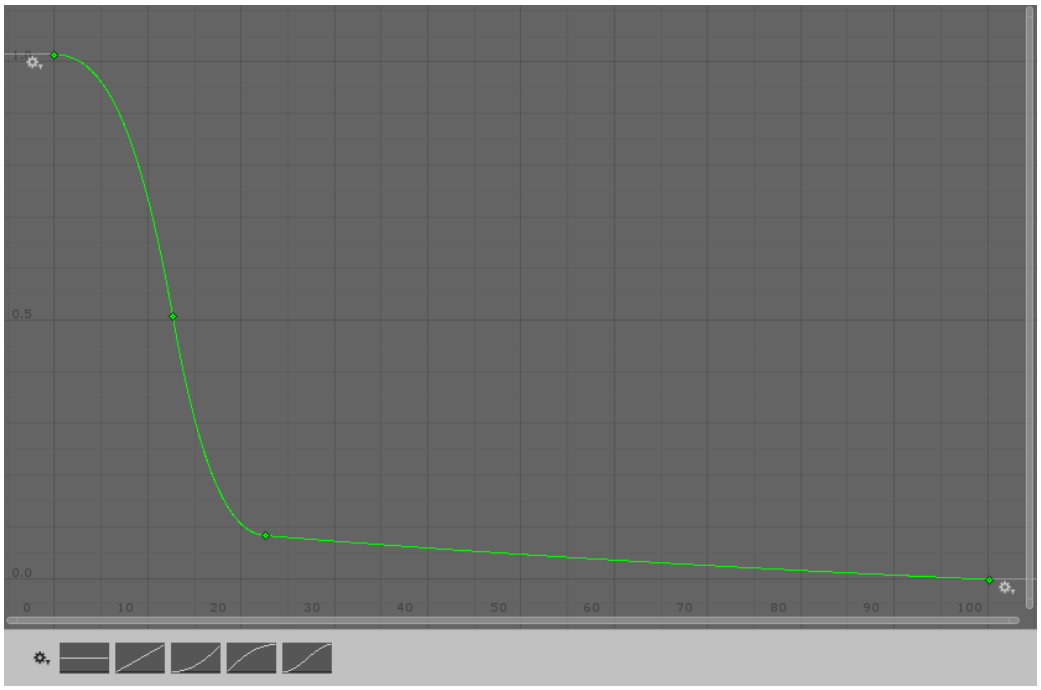
Metodin alussa määritetään pelihahmon nopeutta normaalia hitaammaksi ja vaihdetaan indikaattorin materiaalin väriä. Seuraavassa if-lausekkeessa tarkistetaan ensin lepopaikan ja hahmon välinen etäisyys. Mikäli se ylittää asetetun arvon eikä hahmolla ole lepopaikan kohdetta valittuna, hahmo nukahtaa sille paikalle, missä sillä hetkellä on. Muussa tapauksessa tarkistetaan, ettei hahmo ole jo lepopaikassa, ja määritetään hahmon päämääräksi lepopaikan positio pelimaailmassa. Samalla muutetaan hahmon pysähdyksissä oleminen epätodeksi ja haveTarget-muuttuja todeksi.

Seuraava if-lauseke tarkistaa, että hahmon jäljellä oleva etäisyys on pienempi tai yhtä suuri kuin kohteen pysähtymisetäisyys ja ettei hahmolla ole uutta liikeprosessia. Näiden pitäessä paikkaansa hahmo pysäytetään, haveTarget muutetaan epätodeksi ja resting-muuttuja todeksi. Kun resting-muuttuja on totta samalla kun hahmo on lepopaikassa, restTimer-muuttujaa kasvatetaan nopeammin ja tämä aika lisätään hahmon energiaan. Mikäli hahmo lepää, mutta ei ole lepopaikassa, restTimer-muuttujaan lisätty aika ei kasva yhtä nopeasti. Mikäli hahmon energia nousee määriteltyyn täyteen energiaan, palautetaan hahmo normaalitilaan. Siirryttäessä lepotilasta pakotilaan lisätään restTimer-muuttujan arvo hahmon energiaan, ja sitten nollataan ajastin ennen tilan vaihtumista.

Pakotila

Pakotilaluokka on ravitsemus- ja lepotiloja yksinkertaisempi, normaalitilaa vastaava tilaluokka. Tässäkin luokassa peritään sama rajapinta kuin muissa, ja se sisältää samat tuosta rajapinnasta perityt metodit.

Pakotilan UpdateState-metodissa suoritetaan RunAway-metodi, joka saa tekoälyn ohjaaman kyläläisen juoksemaan pakoon pelaajan ohjaamaa zombie-hahmoa. Tässäkään luokassa OnTriggerEnter-metodi ei tee mitään, eikä tiloihin siirtymisissä ole mitään poikkeavaa. Tällä luokalla ei myöskään ole mitään omia muuttujia. Kuvassa 13 on määritelty pakoon pääsyn tarvekäyrä.



Kuva 13. Käyrä pelihahmon pakoon pääsyn tarpeesta.

Esimerkkikoodi 8 näyttää pelihahmon pakoon juoksun toimintavan ohjelmakoodissa.

```
private void RunAway()
{
      villager.indicator.material.color = Color.red;
      villager.energy -= Time.deltaTime * 5;
      villager.health -= Time.deltaTime * 3;

      Vector3 zombieToVillager = villager.transform.position –
            villager.zombie.transform.position;

      villager.navMeshAgent.destination = villager.transform.position +
            Vector3.Normalize(zombieToVillager) * 20f;

      float speed = 20 - Vector3.Distance(villager.zombie.transform.position,
            villager.transform.position);

      if(speed < 3) { speed = 3; }

      villager.navMeshAgent.speed = speed;
      villager.navMeshAgent.isStopped = false;
}
```

Esimerkkikoodi 8. Pakotilan RunAway-metodi.

Kuten muissakin tilaluokissa, muutetaan indikaattorin materiaaliväri helpottamaan tilojen seurantaa. Seuraavaksi vähennetään korotetusti hahmon energiaa ja ravitsemusta

hahmon paetessa. Määriä voi muuttaa tilanteeseen sopivaksi; ajatus on, että pakeneminen kuluttaa normaalia enemmän hahmon jaksamista.

Tämän jälkeen lasketaan vektori pelihahmon ja pelaajan ohjaaman zombie-hahmon positioiden välillä. Tätä etäisyyttä käytetään laskemaan päämäärä pakenevalle pelihahmolle. Päämäärä on uusi Vector3-muuttuja, joka lasketaan lisäämällä normalisoitu vektori zombien ja pelihahmon etäisyydestä kertomalla se luvulla 20 ja lopuksi lisätään tämä hahmon alkuperäiseen sijaintiin. Hahmon nopeutta pakenemisen aikana säädetään lukuun 20, josta vähennetään vektorietäisyys zombien ja pelihahmon positioiden välillä. Mikäli nopeus olisi alle kolme, muutetaan nopeus aina kolmeksi. Aivan viimeisenä määritetään vielä, että pelihahmo ei ole pysähtynyt. Muuta pakotilassa ei tehdäkään.

Tilojen hallinta

Tilojen hallinta suoritetaan pääasiassa StateEnemyPattern-luokassa. Tämä ohjelmakoodi on yhdistetty Villager-peliobjektiin eli pelihahmoon, jonka toimintaa tekoäly ohjaa. Tässä luokassa alustetaan kaikki muuttujat, joita tarvitaan tilaluokkien toiminnan toteuttamisessa. Kun tilaluokkaan luodaan instanssi StateEnemyPattern-luokasta, voidaan viitata kaikkiin StateEnemyPattern-luokan julkisiin muuttujiin tilaluokissa. Myös IEnemyState rajapinta kuuluu tilojen hallintaan, sillä kaikki tilaluokat perivät tämän rajapinnan ja siellä alustetut metodit. Vaikka rajapinta ei sisällä paljon koodia tai toimintoja, on sen olemassaolo välttämätön ohjelman toiminnalle. Rajapinnan määrittely nähdään esimerkkikoodissa 9.

```
public interface IEnemyState
{

    void UpdateState();

    void OnTriggerEnter(Collider other);

    void ToNormalState();
    void ToHealthState();
    void ToRestState();
    void ToRunAwayState();

}
```

Esimerkkikoodi 9. Vihollishahmon tilarajapinta.

StateEnemyPattern-luokka on kenties tärkein luokka tekoälyn toiminnan kannalta. Tässä luokassa lasketaan kunkin toiminnon tarve ja määritetään pelihahmon muuttujat. Kun peli käynnistetään, ensimmäisenä luokka suorittaa Awake-metodin, jossa haetaan pelihahmon Navigation Mesh Agent, sekä ilmennetään kaikki tilaluokat. Pelin käynnistyessä ajetaan myös Start-metodi, jossa tässä projektissa on vain tyylitelty testaukseen tarkoitetut tekstit ja määritelty animaatiokäyrät. Suoritettavan tarpeen laskeminen suoritetaan omassa metodissaan esimerkkikoodissa 10.

```
private void CalculateUtility()
{
      float[] urges = {AI_Curves.normal.Evaluate(CalculateNormalUrge()),
                       AI_Curves.health.Evaluate(CalculateHealthUrge()),
                       AI_Curves.rest.Evaluate(CalculateRestUrge()),
                        AI_Curves.runAway.Evaluate(CalculateRunAwayUrge())};

      int maxValue = 0;
      float max = urges[0];

      for(int i = 0; i < urges.Length; i++)
      {
            if(urges[i] > max)
            {
                  max = urges[i];
                  maxValue = i;
            }
      }

      if(maxValue == 0 && currentState != normalState)
      { currentState.ToNormalState(); }
      if(maxValue == 1 && currentState != healthState)
      { currentState.ToHealthState(); }
      if(maxValue == 2 && currentState != restState)
      { currentState.ToRestState(); }
      if(maxValue == 3 && currentState != runAwayState)
      { currentState.ToRunAwayState(); }
}
```

Esimerkkikoodi 10. Suurimman tarpeen laskenta CalculateUtility-funktiossa.

CalculateUtility-metodissa luodaan lista pelihahmon tarpeista. Listaan laitetaan evaluoidut käyrät kaikista tarpeista, ja käyrä, josta saadaan korkein tarve, määritetään senhetkiseksi toiminnaksi.

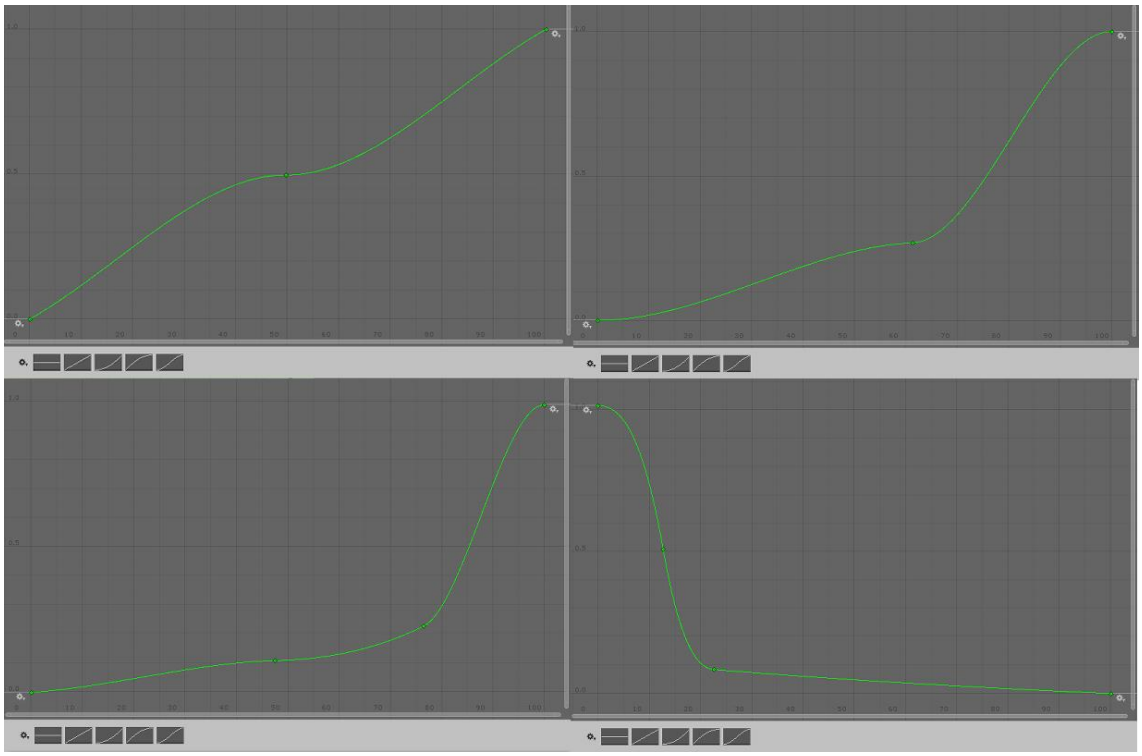
Luokan Update-funktiossa kutsutaan tätä laskentametodia ja määritetään senhetkinen tila pelihahmolle. Update-funktiossa myös lisätään ajan kuluessa timeCounter- ja lastmeal-muuttujien arvoja ja vähennetään hahmon energiaa ja ravitsemusta. Tässä samassa funktiossa myös lasketaan pelimaailman päivän kulumista: pelimaailmassa päivällä on neljä vaihetta. Näitä vaiheita voisi lisätä haluamakseen.

5 Tekoälyn toiminnan arviointi

5.1 Toteutuneet ominaisuudet

Hahmon tekoälyyn saatiin toteutettua neljä eri tilaa, joihin pelihahmo siirtyy tiettyjen tarvelaskelmien perusteella. Pelihahmo liikkuu satunnaisesti pelimaailmassa normaalitilassa, hakeutuu lähimmän ravinnon luokse, kun ravitsemustarve kasvaa, nukahtaa paikalleen lepopaikan ollessa liian kaukana, ja tarpeeksi lähellä ollessaan hakeutuu lepopaikkaan. Viimeisenä pelihahmo pakenee pelaajaa pelaajan tullessa lähelle pelihahmoa.

Jokainen näistä ominaisuuksista toimii, kunhan tarve on laskettu ja tarpeiden käyrät asetettu mielekkäästi. Kuvassa 14 näkyy tekoälyllä toimivalle pelihahmolle Unitylla määritetyt tarvekäyrät. Normaalitilan käyrä, joka mittaa pelihahmon energiaa, vähenee aluksi melko tasaisesti, kunnes puolessavälissä käyrää energian kuluminen lähes pysähtyy ja lopulta jatkaa taas tasaisesti vähentymistä. Ravitsemuskäyrä laskee aluksi melko jyrkästi, kunnes tasoittuu ja lopuksi laskee taas kiihtyvämmin. Lepokäyrä on lähes samanlainen ravitsemuskäyrän kanssa, mutta se laskee jyrkemmässä kulmassa ravitsemuskäyrää alemmas, ennen kuin jatkaa tasaista vähentymistä. Pakoon juoksun käyrä eroaa eniten muista, sillä sen käyrän minimi- ja maksimiarvot ovat eri puolella käyrän x-akselia. Pakoon juoksun käyrällä määritetään, että pelaajan ohjaama zombie-hahmo voi tulla melko lähelle tekoälyn ohjaamaa pelihahmoa, ennen kuin hahmon tarve juosta pakoon nousee kovinkaan ylös.



Kuva 14. Tekoälyn ohjaaman hahmon tarvekäyrät Unityssa.

Teoriassa siis tekoälyn perustoiminnot toimivat toivotulla tavalla. Yksi suurimpia ongelmia oli suunnittelun huolimattomuus. Alkuperäisessä suunnitelmassa tekoäly olisi ohjannut myös zombie-laumaa, mutta pelaamisen kannalta olisi ollut yksitoikkoista, jos pelaaja ei voi ollenkaan ohjata pelihahmoja. Ratkaisua, jossa olisi voinut yhdistää tekoälyn toimintaa ja hahmojen ohjaamista itse, ei ehditty keksimään ajoissa.

Normaalitilan toiminnan ei ole tarkoituskaan olla kovin monimutkainen, ja tämänhetkinen ratkaisu on oikein toimiva. Mahdollinen lisäys voisi esimerkiksi olla vahingon saaminen, kun pelihahmoon osuu. Myös ravitsemustila toimii suurimmaksi osaksi tarpeiden mukaisesti. Yhtenä selvänä ongelmana on, että pelihahmot eivät tunnista, onko heidän ravintokohteensa jo valittu. Toimintoa testatessa kävi ilmi, että pelihahmot voivat helposti jäädä jumiin samalla ravinnonlähteelle mennessään sinne samaan aikaan. Tätä ei valitettavasti ehditty korjata, sillä se oli suhteellisen pieni ja vain silloin tällöin esiintyvä vika.

Eniten parannettavaa on lepotilan toiminnassa. Levontarpeen laskennassa oli alusta asti paljon ongelmia: ohjelmassa määritelty päivänaika nosti aluksi unentarpeen aivan liian korkealle heti, kun päivänaika oli arvoltaan yli yksi. Pelihahmon oli lähes mahdoton

päästä pois lepotilasta, sillä levontarve ylitti muut tarpeet jatkuvasti. Ainoastaan ravinnontarve nousi välillä yli levon, jolloin hahmo heräsi syömään. Syömisen jälkeen hahmo nukahti heti uudelleen, mikäli päivänaika oli yli yhden.

Toimintaa yritettiin muuttaa useita kertoja laskukaavaa ja tarvekäyrää muuttamalla, mutta muutokset menivät yleensä vain huonompaan suuntaan. Lopulta lepotila päätyi pisteeseen, jossa se ei koskaan nouse korkeimmaksi tarpeeksi. Tarve palautui alhaiseksi aina, kun se oli lähellä nousta yli jostain toisesta tarpeesta. Tätä ei valitettavasti ehditty korjaamaan kuntoon.

5.2 Tekoälyn toiminnan arviointi ja vertailu

Esimerkin vuoksi verrataan projektin tekoälytoteutusta The Sims -pelisarjan hahmojen tekoälyn toimintaan. The Sims -hahmojen toiminta oli kenties lähinnä sitä, mitä pyrittiin projektissa simuloimaan. Tässä projektissa hahmon tarpeita on selvästikin paljon vähemmän, kuin The Sims -peleissä, mutta toteutuneissakaan tarpeissa ei ole päästy yhtä hienosäädettyyn painottamiseen tarpeiden kanssa. Sen sijaan, että tarpeista valittaisiin aina se, jonka arvo on korkein, The Sims -peleissä on selvästi vielä piilotettua painotusta eri tarpeissa. Esimerkiksi tarve käydä vessassa voi olla tärkeämpää kuin pesulla käynti, vaikka tarpeet näkyisivät olevan yhtä korkealla. Tässä projektissa olisi voitu ottaa käyttöön esimerkiksi satunnainen valinta korkeiden tarpeiden valinta, jottei päätöksenteko tuntuisi niin kankealta.

Tekoälyn toteutuksessa olisi ollut hyvä saada kehitettyä järjestelmä, joka olisi pystynyt jollain tavalla ottamaan huomioon tekoälyllä toimivien pelihahmojen määrän ja muokkaamaan käyttäytymismalleja sen mukaan. Nykyinen järjestelmä toimii yksittäisessä hahmossa eikä ota huomioon muita tekoälyllä toimivia hahmoja. Erityisesti tämä ilmenee projektissa ravinnonlähteen valinnassa, missä pelihahmot saattavat hakeutua samalle kohteelle, mikäli se on kummallekin hahmolle lähin kohde.

Alkuperäisessä suunnitelmassa, jossa tekoäly olisi ohjannut zombie-hahmoja, ajatus oli luoda järjestelmä, jossa zombie-hahmojen määrä vaikuttaa merkittävästi pelin kulkuun ja muiden hahmojen käyttäytymiseen. Mitä useampia zombie-hahmja olisi, sitä todennäköisemmin ne hyökkäisivät kyläläisten kimppuun. Mikäli peliprojektissa olisi ollut myös

aggressiivisia vihollisia, olisi zombie-hahmoja pitänyt olla vielä enemmän. Myös vahvemmat viholliset olisivat saattaneet juosta karkuun, jos vastassa olisi ollut todella monta zombie-hahmoa.

Toteutuneessa järjestelmässä tämän idean olisi voinut soveltaa eri tavoin. Tekoälyn ohjaamat hahmot olisivat esimerkiksi voineet hakeutua toistensa luokse ennen nukahtamista tai ylipäätään pyrkiä pysymään yhdessä, mikäli hahmojen energia ja ravitsemus laskeutuvat alle tietyn pisteen.

Toteutunut tekoäly vastaa tämänhetkisessä tilassaan enemmän Pac-Manin aaveita: hahmolla on eri tiloja, joihin siirrytään pelin aikana ja toteutetaan eri toimintoa perustuen tilaan, jossa hahmo on. Projektin toteutus kyllä mahdollistaa luonnollisemman siirtymän tilojen välillä verrattuna Pac-Man-pelin hahmoihin.

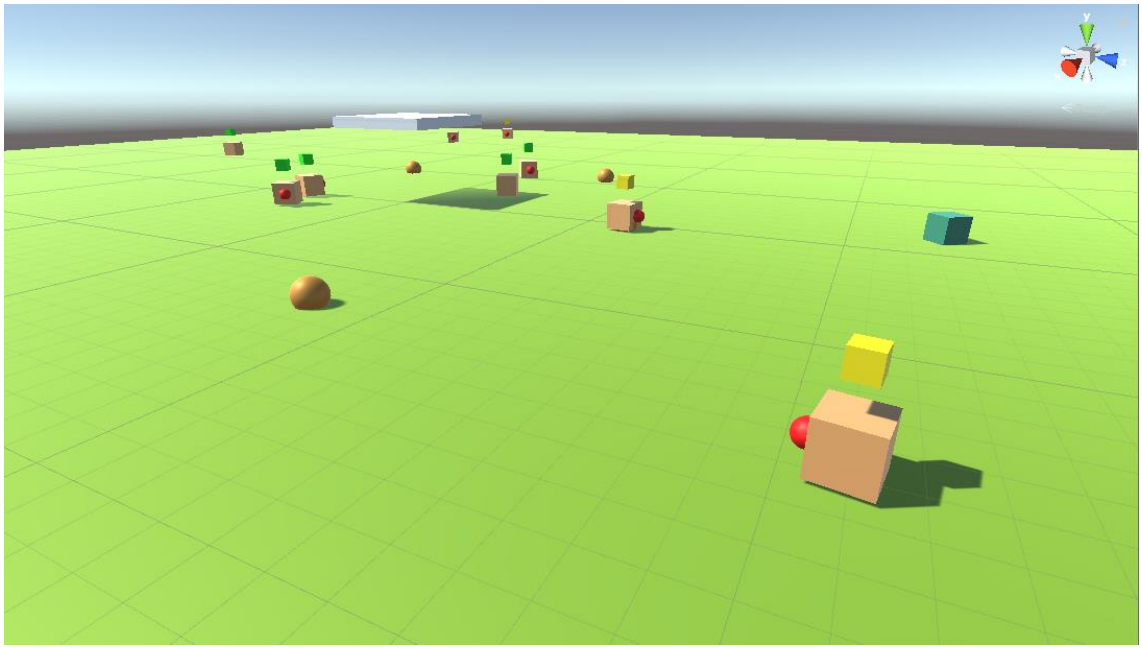
5.3 Tekoälyn kehittäminen

Tekoälyn toimintaan jäi paljon kehitettävää; erityisesti lepotilan toiminta tulisi saada toimimaan mielekkäämmällä tavalla. Lisäksi toinen jo mainittu ominaisuus, satunnaisuuden lisääminen korkeimpiin tarpeisiin, olisi varmasti toteuttamisen arvoinen jatkotoimenpide. Alkuperäinen toive oli myös saada peliin aggressiivisesti toimiva vihollishahmo, jonka läsnäolo vaikuttaisi myös passiivisten pelihahmojen toimintaan. Tätä toteutusta voisi lähteä työstämään, kun aikaisemman tekoälyn toiminta on saatu toimimaan kunnolla ainakin pääpiirteittäin. Pelihahmolle tulisi silloin lisätä myös uusia tarpeita, ja niiden toiminta yhdessä jo tehtyihin tulisi myös silloin saada toimimaan.

Aggressiivinen hahmo toisi myös uhan pelaajan ohjaamalle hahmolle, ja tällöin voitaisiin lähteä toteuttamaan myös alkuperäisessä suunnitelmassa toivottua zombie-hahmon tekoälyä. Insinöörityöprojektissa tehtyä tekoälytoteutusta voisi käyttää lähes suoraan zombie-hahmolle. Tällöin pitäisi vain lisätä pelihahmojen jahtaamiselle ja hyökkäämiselle toiminnot. Zombie-hahmoja saataisiin tällöin useampia peliin, ja hahmojoukkojen välinen toiminta pääsisi paremmin esiin. Zombie-hahmojen määrä olisi tällöin voinut määrittää kyläläishahmojen käyttäytymistä eri tilanteissa. Esimerkkitilanteena kyläläishahmot eivät menisi yhtä herkästi hakemaan ravintoa pelimaailmassa, jos havaitsisivat lähettyvillä

olevan paljon zombie-hahmoja. Vastaavasti kyläläishahmot voisivat toimia uskaliaammin, jos zombie-hahmoja ei olisikaan kovin paljon.

Tämänhetkisessä tilassaan tekoälyn toiminnassa ei juurikaan tule esiin hahmojoukon vaikutus yksittäisten hahmojen toimintaan, joka oli alun perin yksi tärkeimmistä toiveista tekoälyn toiminnassa. Kuvassa 15 näkyy useampia tekoälyllä ohjattuja pelihahmoja, joista osalla on ravitsemustarve noussut, kun osalla se on vielä normaali. Tämä johtuu siitä, että pelaaja jahtasi näitä yksilöitä ja siitä syystä hahmojen ravinnontarve on noussut muita korkeammaksi. Vaikka siis tekoälyn ohjaamien hahmojen tilat voivat poiketa toisistaan pelin aikana, ei tekoäly huomioi millään tavalla muiden pelihahmojen tarpeita ja tiloja.



Kuva 15. Useampia tekoälyllä ohjattuja hahmoja pelimaailmassa.

Kehnon suunnittelun ja oikean toteutustavan löytämisen vaikeuden takia aikaa itse tekoälyn hiomiselle jäi todella vähän. Tästä johtui myös, ettei kaikkia ominaisuuksia saatu toimivaan täysin halutulla tavalla ja tekoälyn toimintaan jäi selkeitä puutteita. Nyt, kun toteutustapa on saatu valittua ja siihen on paremmin tutustuttu, olisi tekoälyn hiominen tulevaisuudessa todennäköisesti selkeämpää ja tehokkaampaa.

6 Yhteenveto

Insinöörityössä perehdyttiin tekoälyyn konseptina videopeleissä ja niiden ulkopuolella ja tehtiin katsaus muutamiin käytetyimpiin tekoälymenetelmiin. Työssä toteutettiin peliprojektiin tarvepohjainen tekoälyratkaisu, jossa pelaaja pelaa Zombie-hahmolla ihmishahmojen keskellä. Ihmishahmoilla on tarpeidensa mukaan eri tiloja, jotka vaikuttavat hahmon toimintatapaan pelissä. Projekti ja tekoäly toteutettiin Unity-pelimoottorilla hyödyntäen tarvetekoälyjärjestelmää.

Tutkimusvaiheessa perehdyttiin tekoälyyn laajempana osa-alueena ja selvitettiin tekoälyn alkuperää pelinkehityksenkin ulkopuolelta. Tämän lisäksi tutustuttiin tekoälyn toimintaperiaatteisiin, kuten koneoppimiseen ja sen alalajeihin, muun muassa neuroverkkoihin. Näistä siirryttiin esimerkkeihin eri tekoälytoteutuksista videopeleissä ja digitalisoiduissa lautapeleissä. Tämä antoi hyvää taustatietoa tekoälystä teknologiana ja siitä, miten sen hyödyntäminen todella näkyy videopeleissä. Vielä toistaiseksi näyttäisi olevan niin, että tekoälyn toimintaperiaatteita voidaan hyödyntää videopeleissä, mutta puhtaasta tekoälystä ei aina voida puhua, kun on kyse videopeleistä. Esimerkeistä löydettiin sopivanoloinen referenssi tämän insinöörityön tekoälytoteutukseen.

Suunnitteluvaiheessa ilmeni jo muutamia ongelmia alkuperäisen idean kanssa, ja tämä johti jokseenkin isoihin muutoksiin tekoälyn toiminnan suunnittelussa. Ohjattavan hahmon sijaan tekoälyä lähdettiin toteuttamaan passiiviselle vihollishahmolle. Myös alkuperäinen toteutustapa vaihtui: tilakoneen sijaan projektissa käytettiinkin tarvejärjestelmää tekoälyn luomiseen. Peliprojektin perusidea ei tästä kuitenkaan kärsinyt liikaa, mutta suunnitelmien muutokset johtivat hieman keskeneräisempään suunnitelmaan toivotuista ominaisuuksista.

Vaikka tekoälyn perustoiminnot toimivat pääpiirteittäin, työhön jäi vielä paljon parannettavaa. Erityisesti muutamien tarpeiden evaluoinnissa ja laskennassa on vielä selkeästi ongelmia, jotka vaikuttavat tekoälyn toimintaan suuresti. Mikäli tämä ongelmat korjattaisiin, toimisi tekoäly hyvänä pohjana, jonka päälle toimintaa voisi laajentaa suhteellisen pienellä vaivalla.

Insinöörityötä tehdessä tarvejärjestelmän toimintaperiaate tuli tutuksi, sillä siihen ei ennen työn aloittamista ollut perehdytty ollenkaan. Tämä lisäsi suuresti ymmärrystä eri pelitekoälyjen toimintamalleista ja mahdollisista toteutuksista sekä vahvisti osaamista Unityn työkalujen käytössä. Tarvejärjestelmä on helposti videopeleihin sovellettava ja mukautettava järjestelmä, jota voi eri tavoin hyödyntää myös tulevaisuudessa.

Lähteet

1 Millington, Ian. 2006. Artificial Intelligence for Games. San Francisco: Elsevier.

2 Yannakakis, Gergios N. 2012. Game AI Revisited. Proceedings of the 9th ACM Computing Frontiers Conference, s. 285–292.

3 Russel, Stuart & Norvig, Peter. 2009. Artificial Intelligence: A Modern Approach. 3rd ed. New Jersey: Pearson.

4 Shead, Mark. State Machines – Basics of Computer Science. Verkkoaineisto. Mark Shead. <https://blog.markshead.com/869/state-machines-computer-science/>. Luettu 5.6.2019.

5 Game Maker Basics: State Machines. 2017. Verkkoaineisto. Amazon Appstore Blogs. <https://developer.amazon.com/blogs/appstore/post/c92030bb-6ab8-421f-b0da-a7231a59561d/gamemaker-basics-state-machines>. Päivitetty 27.10.2017. Luettu 5.6.2019

6 Nath, Kousik. 2019. State Machine Design pattern —Part 1: When, Why & How. Verkkoaineisto. Medium. <https://medium.com/datadriveninvestor/state-machine-design-pattern-why-how-example-through-spring-state-machine-part-1-f13872d68c2d>. Päivitetty 15.1.2019. Luettu. 5.6.2019.

7 A* Search Algorithm. Verkkoaineisto. GeeksforGeeks. <https://www.geeksforgeeks.org/a-search-algorithm/>. Luettu 27.6.2019.

8 Getting started with Machine Learning. Verkkoaineisto. GeeksforGeeks. <https://www.geeksforgeeks.org/getting-started-machine-learning/>. Luettu 10.6.2019.

9 Luo, Joshua. J. 2019. An Exploration of Neural Networks Playing Video Games. Medium. <https://towardsdatascience.com/an-exploration-of-neural-networks-playing-video-games-3910dcee8e4a>. Päivitetty 22.5.2019. Luettu 23.8.2019.

10 Buckland, Mat. 2002. AI Techniques for Game Programming. Ohio: Premier Press.

11 Tutak, Mandela & Brodny, Jaroslaw. 2019. Predicting Methane Concentration in Longwall Regions Using Artificial Neural Networks. International Journal of Environmental Research and Public Health 16, s. 9.

12 Decision Tree. Verkkoaineisto. GeeksforGeeks. <https://www.geeksforgeeks.org/decision-tree/>. Luettu 24.8.2019.

13 Hartikka, Lauri. 2017. A step-by-step guide to building a simple chess AI. Verkkoaineisto. FreeCodeCamp. <https://www.freecodecamp.org/news/simple-chess-ai-step-by-step-1d55a9266977/>. Päivitetty 30.3.2017. Luettu 11.9.2019.

14 Mateas, Michael. 2003. Expressive AI: Games and Artificial Intelligence. DiGRA Conference.

15 Bourse, Yoann. 2012. Artificial Intelligence in The Sims series. Verkkoaineisto. Yoann Bourse. <http://www.yoannbourse.com/ressources/docs/ens/sims-slides.pdf>. Luettu 21.10.2019.

16 Graham, David “Rez”. 2013. Game AI Pro: Collected Wisdom of Game AI Professionals. Florida: CRC Press.

17 Dill, Kevin. 2012. Design Patterns for the Configuration of Utility-Based AI. Interservice/Industry Training, Simulation, and Education Conference (I/ITSEC) 2012 Paper No. 12146.

18 The Sims 2 Motives. Verkkoaineisto. Sims Wiki. <http://simswiki.info/wiki.php?title=Sims_2_My_motive>. Luettu 21.10.2019.

19 Axon, Samuel. 2016. Unity at 10: For better—or worse—game development has never been easier. Verkkoaineisto. Ars Technica. <https://arstechnica.com/gaming/2016/09/unity-at-10-for-better-or-worse-game-development-has-never-been-easier/#>. Päivitetty 27.9.2016. Luettu 21.10.2019.

20 Brodkin, Jon. 2013. How Unity3D Became a Game-Development Beast. Verkkoaineisto. Dice. <https://insights.dice.com/2013/06/03/how-unity3d-become-a-game-development-beast/>. Päivitetty 3.6.2013. Luettu 22.10.2019.

21 Introduction. 2017. Verkkoaineisto. Microsoft. <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/>. Päivitetty 7.1.2017. Luettu 23.10.2019.

22 Sivonen, Veli-Matti. 2004. C#-kieli. Verkkoaineisto. Helsingin yliopisto. <https://www.cs.helsinki.fi/u/pohjalai/k04/ohpe/seminar/Sivonen-CSharp.pdf>. Päivitetty 4.4.2004. Luettu 23.10.2019.

23 Coding in C# in Unity for beginners. Verkkoaineisto. Unity. <https://unity3d.com/learning-c-sharp-in-unity-for-beginners>. Luettu 23.10.2019.