Joonas Hyttinen

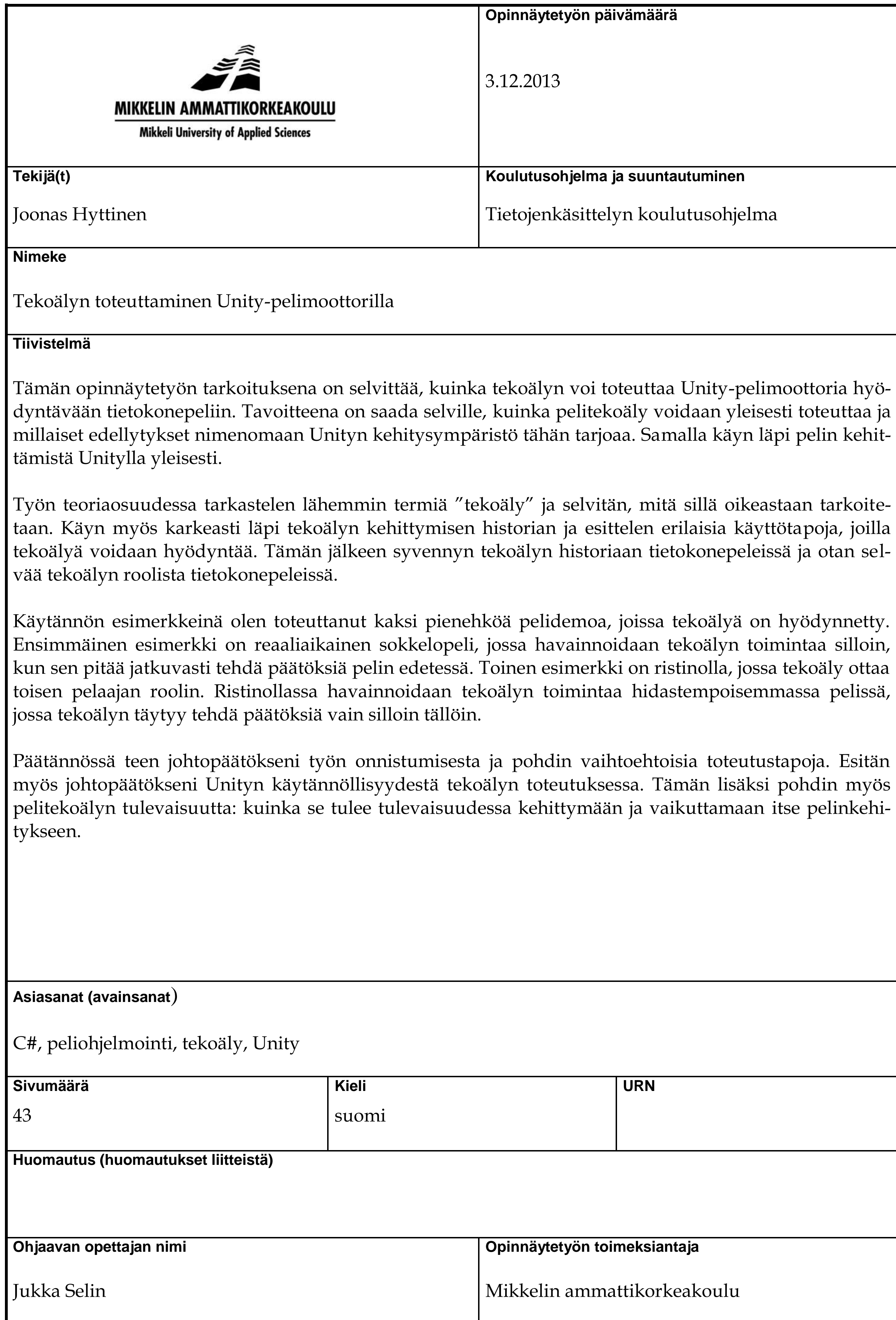
TEKOÄLYN TOTEUTTAMINEN UNITY-PELIMOOTTORILLA

Opinnäytetyö
Tietojenkäsittely

Joulukuu 2013

MIKKELIN AMMATTIKORKEAKOULU
Mikkeli University of Applied Sciences

KUVAILULEHTI



MIKKELIN AMMATTIKORKEAKOULU
Mikkeli University of Applied Sciences

Opinnäytetyön päivämäärä

3.12.2013

Tekijä(t)

Joonas Hyttinen

Koulutusohjelma ja suuntautuminen

Tietojenkäsittelyn koulutusohjelma

Nimeke

Tekoälyn toteuttaminen Unity-pelimoottorilla

Tiivistelmä

Tämän opinnäytetyön tarkoituksena on selvittää, kuinka tekoälyn voi toteuttaa Unity-pelimoottoria hyödyntävään tietokonepeliin. Tavoitteena on saada selville, kuinka pelitekoäly voidaan yleisesti toteuttaa ja millaiset edellytykset nimenomaan Unityn kehitysympäristö tähän tarjoaa. Samalla käyn läpi pelin kehittämistä Unitylla yleisesti.

Työn teoriaosuudessa tarkastelen lähemmin termiä ”tekoäly” ja selvitän, mitä sillä oikeastaan tarkoitetaan. Käyn myös karkeasti läpi tekoälyn kehittymisen historian ja esittelen erilaisia käyttötapoja, joilla tekoälyä voidaan hyödyntää. Tämän jälkeen syvennyn tekoälyn historiaan tietokonepeleissä ja otan selvää tekoälyn roolista tietokonepeleissä.

Käytännön esimerkkeinä olen toteuttanut kaksi pienehköä pelidemoa, joissa tekoälyä on hyödynnetty. Ensimmäinen esimerkki on reaaliaikainen sokkelopeli, jossa havainnoidaan tekoälyn toimintaa silloin, kun sen pitää jatkuvasti tehdä päätöksiä pelin edetessä. Toinen esimerkki on ristinolla, jossa tekoäly ottaa toisen pelaajan roolin. Ristinollassa havainnoidaan tekoälyn toimintaa hidastempoisemmassa pelissä, jossa tekoälyn täytyy tehdä päätöksiä vain silloin tällöin.

Päätännössä teen johtopäätökseni työn onnistumisesta ja pohdin vaihtoehtoisia toteutustapoja. Esitän myös johtopäätökseni Unityn käytännöllisyydestä tekoälyn toteutuksessa. Tämän lisäksi pohdin myös pelitekoälyn tulevaisuutta: kuinka se tulee tulevaisuudessa kehittymään ja vaikuttamaan itse pelinkehitykseen.

Asiasanat (avainsanat)

C#, peliohjelmointi, tekoäly, Unity

Sivumäärä	Kieli	URN
43	suomi	

Huomautus (huomautukset liitteistä)

Ohjaavan opettajan nimi

Jukka Selin

Opinnäytetyön toimeksiantaja

Mikkelin ammattikorkeakoulu

DESCRIPTION

MIKKELIN AMMATTIKORKEAKOULU Mikkeli University of Applied Sciences	**Date of the bachelor's thesis** 3 December 2013
Author(s) Joonas Hyttinen	**Degree programme and option** Business Information Technology

Name of the bachelor's thesis

Development of artificial intelligence with the Unity game engine

Abstract

The purpose of this bachelor's thesis was to find out how to implement artificial intelligence in a computer game by using the Unity game engine. The goal was to find out how computer game AI could be created in general, and how Unity could be utilized to make the task easier. General game development with Unity was also studied.

The theoretical part of this thesis studied the term artificial intelligence itself. I clarified the meaning of artificial intelligence, summarized its brief history and discovered what different purposes it could be used for. I also studied the uses of AI in computer games and dealt with the history of computer game AI.

As the practical part of my thesis I made two small game examples utilizing AI. The first one was a real-time maze game that demonstrated AI functionality in a fast-paced action game. The second game was a simple game of tic-tac-toe. This example demonstrated AI functionality in a slow-paced turn-based strategy game.

The summary presented my thoughts on the results of the study and discussed how a different approach could have affected the outcome. I also presented my conclusion on how practical Unity was when developing artificial intelligence. In the end, I discussed the present state of game AI and how it might change in the future.

Subject headings, (keywords)

artificial intelligence, C#, game programming, Unity

Pages	Language	URN
43	Finnish	

Remarks, notes on appendices

Tutor	Bachelor's thesis assigned by
Jukka Selin	Mikkeli University of Applied Sciences

SISÄLTÖ

1 JOHDANTO ... 1

2 TEKOÄLY ... 2

2.1 Tekoäly käsitteenä ... 2

2.1.1 Vahva tekoäly ... 3

2.1.2 Heikko tekoäly ... 4

2.2 Historia ... 5

2.2.1 Tekoälyn synty ... 5

2.2.2 Nousu- ja laskukaudet ... 6

2.2.3 90-luvulta nykyaikaan ... 8

2.3 Sovellustapoja ... 9

3 TEKOÄLY TIETOKONEPELEISSÄ ... 12

3.1 Pelitekoälyn historia ... 12

3.2 Käyttötarkoitukset peleissä ... 17

3.2.1 Pelien pelaaminen ... 17

3.2.2 Äärellinen tilakone ... 20

3.2.3 Sumea logiikka ... 21

3.2.4 Polunetsintä ... 23

3.2.5 Keinotekoiset neuroverkot ... 25

4 ESIMERKKIEN TOTEUTTAMINEN ... 27

4.1 Sokkeloesimerkki ... 28

4.1.1 Reitinetsinnän toteuttaminen ... 29

4.1.2 Tilakoneen toteuttaminen ... 33

4.2 Ristinolla ... 37

5 PÄÄTÄNTÖ ... 41

LÄHTEET ... 44

1 JOHDANTO

Tekoäly on keskeinen ja hyvin arvostettu tietokonepelien osa-alue. Siinä missä varhaiset ohjelmointitekniikat ja rajallinen prosessointiteho mahdollistivat lähinnä kömpelön ja ennalta arvattavan tekoälyn toteuttamisen, nykyiset tekoälyalgoritmit ja tehokkaat tietokoneet mahdollistavat paljon arvaamattomamman ja älykkäämmän oloisen tekoälyn luomisen. Toisin kuin ehkä voisi kuvitella, kekseliään tekoälyn kehittäminen ei välttämättä vaadi valtavaa työpanosta eikä mittavia investointeja. Tästä syystä monet pienet pelikehittäjät yrittävätkin voittaa pelaajat puolelleen mahdollisimman nerokkaiden ja omalaatuisten tekoälyratkaisujen avulla, joiden kehittämiseen isoimmilla pelistudioilla ei useimmiten ole kiinnostusta.

Tämän opinnäytetyön tarkoituksena on selvittää, millä eri keinoin tekoälyn voi toteuttaa Unity-kehitysympäristössä tehtyyn peliin, miten se onnistuu ja kuinka hyvät edellytykset tekoälyn toteutukselle Unity tarjoaa. Työn aihe on saanut alkunsa omien kiinnostusteni pohjalta. Ajatus pelin kehittämisestä Unitylla on kiehtonut minua jo pidemmän aikaa, ja opinnäytetyö antoi yrittämiselle hyvät edellytykset. Opinnäytetyön toimeksiantajana on Mikkelin ammattikorkeakoulu.

Opinnäytetyön toisessa luvussa perehdyn itse tekoälyyn ja käyn karkeasti läpi tekoälyn kehittymisen historian. Tarkoituksena on saada selville, mitä käsitteellä ”tekoäly” oikeastaan tarkoitetaankaan ja milloin jotakin ohjelman toiminnallisuutta voidaan tällaiseksi luonnehtia. Käyn myös läpi joitakin esimerkkitapauksia, joissa tekoälyä voidaan hyödyntää tai on hyödynnetty.

Kolmannessa luvussa pureudun enemmän tekoälyn rooliin tietokonepeleissä. Perehdyn siihen, miten tekoäly on tietokonepeleissä kehittynyt ja miten pelit puolestaan tekoälyn kehityksen myötä. Esittelen myös joitakin tavanomaisimpia tekniikoita, joita peleissä usein käytetään tekoälyn saralla.

Neljännessä luvussa kerron puolestaan itse käytännön toteutuksesta. Projektini päätavoitteena on tuottaa pieniä pelidemoja, joissa toimii jonkinlainen tekoäly. Lopputuloksen saavuttamiseen on tarkoituksena käyttää ainoastaan vapaasti saatavilla olevia, il-

maisia ohjelmistoja ja liitännäisiä. Esittelen itse esimerkit ja kerron sen jälkeen tarkemmin niiden suunnittelusta ja toteutuksesta.

Päätännössä esitän johtopäätökseni projektin etenemisestä ja valmiista lopputuloksesta. Arvioin lopputuloksen ja pohdin, mitä projektissa olisi voinut tehdä eri tavalla ja mitä etua Unitysta on tekoälyä toteutettaessa. Pohdin myös työssä käytettyjen tekniikoiden ja toteutustapojen soveltuvuutta suurempiin tai monimutkaisempiin projektikokonaisuuksiin.

2 TEKOÄLY

Tässä luvussa selvitän, mitä termillä ”tekoäly” oikeastaan tarkoitetaan. Tämän lisäksi valotan tekoälyn historiaa ja esittelen erilaisia tekoälytyyppejä. Käyn läpi myös joitakin yleisimpiä tekoälyn sovellustapoja ja esimerkkitapauksia, joissa tekoälyä on hyödynnetty. Tarkoituksena on keskittyä enemmänkin siihen, mitä konkreettista tekoälyllä voi tehdä, kuin siihen, miten tekoäly teknisesti ottaen toimii.

2.1 Tekoäly käsitteenä

Tekoäly, eli *artificial intelligence,* on tietokonetieteen haara, jonka tavoitteena on saada tietokoneet ajattelemaan, päättelemään ja käyttäytymään itsenäisesti ihmisen tavoin (Brookshear 2003, 437). Tekoälyllä voidaan myös tarkoittaa tietokoneohjelmaa, joka täyttää edellä mainitut kriteerit, eli vaikuttaa toimivan älykkäästi.

Yhtenä tekoälyn perusongelmista on pidetty sitä, että käsite ”älykkyys” on hyvin epämääräinen, eikä sen tarkasta määritelmästä tekoälyn kontekstissa ole päästy yksimielisyyteen. Tekoäly-termin keksineen John McCarthyn mukaan (2007) ongelmana on, että emme vielä pysty määrittelemään, mitkä kaikki laskennalliset menetelmät voimme luokitella älykkäiksi, ja mitä me emme voi. Yksimielisyyteen ei ole päästy myöskään siitä, voidaanko tekoälyä pitää älykkäänä silloin, kun se ajattelee ja toimii niin kuin ihminenkin toimisi vai silloin, kun se pyrkii toimimaan mahdollisimman rationaalisesti, moraalisista ja inhimillisistä tekijöistä piittaamatta (Russell & Norvig 2005, 1–2).

2.1.1 Vahva tekoäly

Vahvalla tekoälyllä tarkoitetaan tekoälyä, joka on oikeasti älykäs. Sen odotetaan siis käytännössä pystyvän samoihin asioihin kuin esimerkiksi ihmisen aivot. Vahvan tekoälyn toteuttamista voitaneen pitää myös tekoälyn tutkimuksen äärimmäisenä tavoitteena. Termiä vahva tekoäly käytti ensimmäisen kerran filosofi John Searle vuonna 1980. (Copeland 2000a.) Vahvalla tekoälyllä voidaan tarkoittaa myös filosofista hypoteesia, jonka mukaan tietokoneet kykenevät oikeasti ajattelemaan, eivätkä ainoastaan simuloimaan ajatteluprosessia (Pfeifer ym. 2006, 17–18).

Thrunin mukaan (2005) yksi tekoälykehityksen suurimmista unelmista on aina ollut rakentaa keinotekoinen, ihmisen arkiaskareiden suorittamiseen kykenevä, älykäs robotti. Hänen mukaansa on kuitenkin ensin kehitettävä paljon tehokkaampia tapoja esittää ja käsitellä tietoa, ennen kuin tämä on edes teknisesti mahdollista.

Tarkasta vahvan tekoälyn määritelmästä ei ole toistaiseksi päästy yhteisymmärrykseen, mutta pääsääntöisesti siltä vaaditaan seuraavia ominaisuuksia:

- kyky havainnoida ympäristöä ja oppia
- omaa jonkin tasoisen tietoisuuden
- ajattelee, perustelee käyttäytymistään
- kyky kommunikoida tai olla vuorovaikutuksessa (Russell & Norvig 2003, 2–3).

Kaikista tähänastisista pyrkimyksistä huolimatta ihminen ei ole toistaiseksi onnistunut luomaan vahvaa tekoälyä. Tutkijat eivät myöskään ole pystyneet aukottomasti todistamaan keinotekoisen älykkyyden olevan edes teoreettisesti mahdollista. Pelkkä kehityksen puute ei toisaalta myöskään todista sen mahdottomuutta. (Copeland 2000b.)

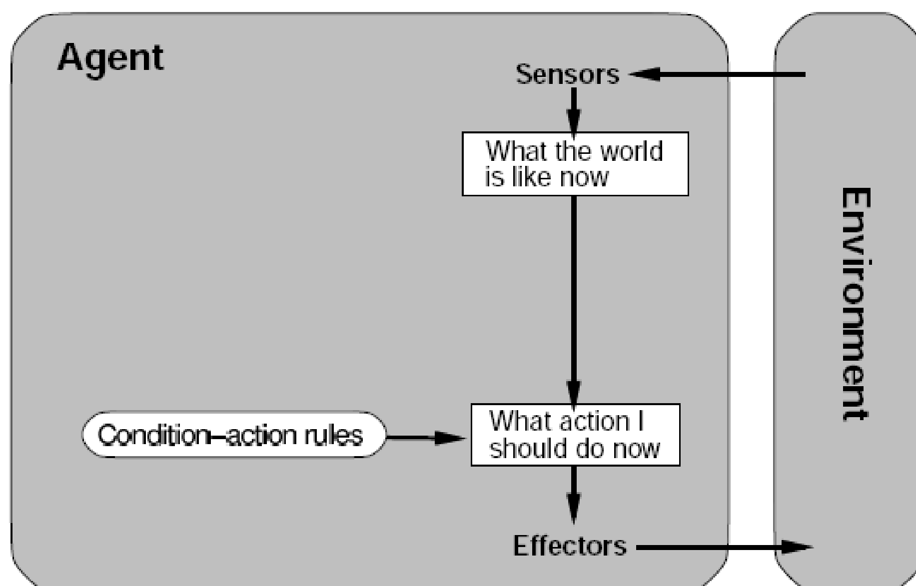
Älykkäät koneet ja niistä unelmoiminen on jättänyt jälkensä myös tieteiskirjallisuuteen ja – elokuviin. Hyvä esimerkki tästä on vuonna 1968 julkaistu elokuva *2001:A Space Odyssey* ja kyseisessä elokuvassa esiintyvä *HAL*- tietokone, joka joidenkin katsojien mielestä käyttäytyi ihmismäisemmin kuin elokuvan varsinaiset ihmishahmot

(Perkowitz 2004, 38). Tämän kaltaiset tieteiskuvitelmat ovat nostaneet esille joitakin tekoälyyn ja sen luomiseen liittyviä moraalikysymyksiä (Gunkel 2012, 1–2).

2.1.2 Heikko tekoäly

Heikolla tekoälyllä tarkoitetaan tekoälyä, joka osaa ratkaista sille annettuja ongelmia ennalta määritettyjen toimintatapojen ja sääntöjen avulla. Toisin kuin vahva tekoäly, heikko tekoäly ei ole oikeasti älykäs, eikä se osaa ratkaista muita kuin ainoastaan sellaisia ongelmia, mitä se on ohjelmoitu ratkaisemaan. Heikon tekoälyn toteuttamiseen riittää tyypillisesti, että se vain vaikuttaa toimivan älykkäästi. (Johnston 2008, 435.)

Tekoälyn toiminnan havainnollistamiseksi voidaan käyttää niin sanottuja älykkäitä agentteja. Agentti voi olla mikä tahansa ”älykäs” entiteetti, joka havainnoi ympäristöä ja on vuorovaikutuksessa ympäristön kanssa: esimerkiksi robotti tai tietokoneohjelma. (Russell & Norvig 2007, 32.) Kuvassa 1 on nähtävissä esimerkki yksinkertaisesta sääntöpohjaisesta agentista (simple reflex agent), joka havainnoi ympäristöä sensoriensa avulla, valitsee seuraavan toimintonsa ympäristöstä saadun tiedon ja ohjelman antamien ehtojen avulla ja lopulta suorittaa sen (mts. 46–47).



KUVA 1. Yksinkertainen agentti (Imperial College London 2006)

Heikkokin tekoäly voi parhaimmillaan olla hyvin monimutkainen käyttötarkoituksesta riippuen. Esimerkiksi shakin peluu vaatii tehokkaita hakualgoritmeja parhaiden siirto-

jen löytämiseksi. Oleellinen ero heikon ja vahvan tekoälyn välillä on kuitenkin se, ettei heikolta tekoälyltä edellytetä itsetietoisuutta eikä kognitiivisia taitoja. Joidenkin mielestä heikkoa tekoälyä ei voi edes tekoälyksi kutsua. (Kumar 2008, 14.)

2.2 Historia

Varsinainen ajatus itsenäiseen ajatteluun kykenevistä koneista on itsessään hyvin vanha. Jo antiikin Kreikan myyteissä on mainintoja inhimillisistä konemiehistä (Russell & Norvig 2005, 939) ja jo antiikin Egyptissä, Kreikassa ja Kiinassa rakennettiin alkeellisia, yksinkertaisia toimintoja toistavia mekaanisia robotteja (History of Automata 2009). Vaikka nämä alkeelliset laitteet aiheuttivat aikanaan niin hämmästystä kuin ihailuakin, niiden toiminnalla ei ollut mitään tekemistä älyllisyyden kanssa.

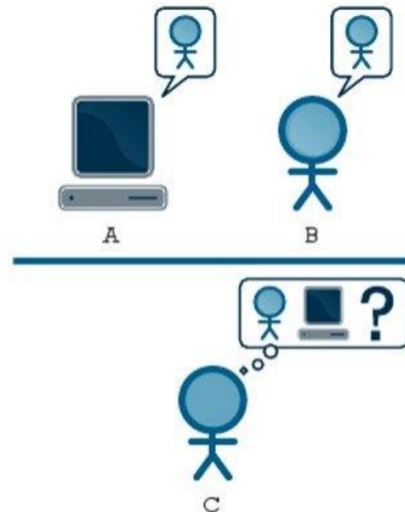
Teknologiset edellytykset tekoälyn toteuttamiselle tulivat 1940-luvulla, elektronisen tietokoneen kehittämisen myötä. Ensimmäiset näistä oli kuitenkin suunniteltu lähinnä purkamaan Toisen maailmansodan aikaisia salakirjoituksia, ja vei vielä aikansa, ennen kuin havaittiin, että tietokoneita voisi hyödyntää paljon monimutkaisempiinkin tarkoituksiin. (Jones 2003, 3.)

2.2.1 Tekoälyn synty

Yksi varhaisimmista tekoälyn syntyyn vaikuttaneista tutkijoista oli yhdysvaltalainen matemaatikko Norbert Wiener. Vuonna 1948 julkaistussa kirjassaan *Cybernetics* hän vertasi ihmisiä koneisiin ja esitti inhimillisen käyttäytymisen olevan koneellisesti imitoitavissa. Wiener teorioi ihmiskäytöksen perustuvan pitkälti yksinkertaiseen palautemekanismiin, jossa ihminen toimii eri tavoin ympäristöstä saadun palautteen mukaan, kunnes haluttu lopputulos saavutetaan. (History of Computers and Computing 2013.)

Uraauurtavaa tutkimusta aiheen parissa teki englantilainen matemaatikko ja logiikan tutkija Alan Turing. Vuonna 1950 julkaistussa artikkelissaan *Computing Machinery and Intelligence* Turing esitteli tekoälytestin, joka on sittemmin saanut nimen *Turingin testi.* Testin tyypillisimmässä asetelmassa (kuva 2) testiin kuuluu kolme osapuolta: tietokone A, henkilö B ja henkilö C. Henkilö C on kuulustelija, joka kysyy osapuolilta A ja B sarjan erilaisia kysymyksiä. Tietokone A ja henkilö B puolestaan vastaavat

kuulustelijan esittämiin kysymyksiin kirjoitettujen viestien välityksellä. Testin lopussa kuulustelijan on saamiensa vastausten perusteella pääteltävä, kumpi vastaavista osapuolista on tietokone. Testin tarkoituksena on saada kuulustelija päättelemään väärin. (Pfeifer & Scheier 1999, 16–17.)



KUVA 2. Turingin testi (Hayes 2012)

1950-luvulla tehtiin myös ensimmäiset tekoälyä soveltavat ohjelmat, kuten *Logic Theorist* ja *General Problem Solver*. Maininnan arvoinen on myös Arthur Samuelin tekemä tammipeli, jonka tekoäly oppi pelaamaan peliä niin hyvin, että se ennen pitkää voitti tekijänsä. Tekoälyn ohjelmointia varten kehitettiin myös kaksi ohjelmointikieltä: *IPL* ja *LISP*. (Jones 2003, 4.)

Varsinainen tekoälyä kuvaava termi *artificial intelligence* syntyi kesällä 1956 Dartmouth Collegessa. Tuolloin pidettiin John McCarthyn aloitteesta kaksi kuukautta kestänyt tutkimusseminaari, jonka seurauksena tekoälystä tehtiin oma tutkimusalansa. Nimen alalle keksi McCarthy, ja sen käyttöönotosta päätettiin yksimielisesti. (Russell & Norvig 2005, 17.)

2.2.2 Nousu- ja laskukaudet

1960-luvulla tekoälyn tutkimus ja kehitys kukoisti. Teknologia kehittyi nopeasti ja yhä useampi tutkija alkoi kiinnostua alasta. Tietoisuus alasta ja sen mahdollisuuksista alkoi levitä, mutta toisaalta ala alkoi kerätä myös kritiikkiä. (Jones 2003, 5.) Itse teko-

älyn tutkimuksen saralla keskityttiin pääosin hakualgoritmien ja heuristiikan tutkimiseen (Shi 2011, 3).

Innostuksesta huolimatta varsinainen innovaatio jäi tekoälyn saralla melko vaatimattomaksi. Todellisuudessa tutkijat olivat useimmiten ylioptimistisia ja heidän odotuksensa epärealistisia. Esimerkiksi Herbert Simon ennusti vuonna 1957, että tietokone yltäisi 10 vuoden sisällä shakkimestarin tasolle ja pystyisi osoittamaan todeksi jonkin merkittävän matemaattisen teoreeman – näin ei kuitenkaan käynyt. (Russell & Norvig 2005, 21.)

Teknologiset rajoitteet, kykenemättömyys vastata mahdottoman korkeisiin odotuksiin ja lisääntyvä kritiikki aiheuttivat lopulta 1970-luvulla tekoälyn tutkimuksen romahtamisen. Yleisesti ajateltiin, että tekoälyllä ei pystyttäisi toteuttamaan mitään käytännöllistä. Tämän seurauksena tekoälyprojekteja lopetettiin tai niiden rahoitusta leikattiin. (Jones 2003, 5; Russell & Norvig 2005, 21–22.)

Vaikeuksista huolimatta tekoälyn tutkimus ei onneksi jäänyt polkemaan paikalleen. 1970-luvun tekoälytutkimuksessa kokeiltiin esimerkiksi sumean logiikan hyödyntämistä ensimmäistä kertaa käytännössä, sekä kehitettiin Prolog-ohjelmointikieli (Jones 2003, 6). Tutkimuksen pääpaino oli tuolloin kuitenkin interaktiivisten, kommunikointiin kykenevien ohjelmien kehittämisessä ja luonnollisten kielten käsittelyn tutkinnassa (Shi 2011, 3).

Tekoälyn kehitys lähti 80-luvulla jälleen nousuun. Alan elpymistä edesauttoi suuresti se, että tekoälypohjaisilla ratkaisuilla alkoi olla runsaasti kaupallista kysyntää. Merkittäviä tulonlähteitä olivat LISP-tietokoneet sekä erilaiset asiantuntijajärjestelmät. Myös niin kutsuttujen neuroverkkojen tutkimusta jatkettiin. Tekoälypohjaisten laitteistojen ja ohjelmistojen liikevaihto oli vuoteen 1986 mennessä yli 400 miljoonaa dollaria. (Jones 2003, 6.)

Suuresta kasvusta huolimatta, ja osittain siitä johtuen, tekoälyn saralla koettiin 80-luvulla myös uusi taantumakausi. Tämä johtui pitkälti sekä LISP-tietokoneiden että asiantuntijajärjestelmien markkinoiden romahtamisesta. Applen ja IBM:n kehittämät työpöytätietokoneet syrjäyttivät LISP-tietokoneet käytännössä kokonaan vuonna

1987, kun taas asiantuntijajärjestelmät osoittautuivat epäkäytännöllisiksi, vaikeiksi käyttää tehokkaasti ja liian kalliiksi ylläpitää. Vuonna 1984 tekoälyn taantumakausista alettiin käyttää yleisnimitystä *AI Winter*, vapaasti käännettynä tekoälyn talvi. (Machines like us 2007.)

2.2.3 90-luvulta nykyaikaan

1990-luvulla, useiden pettymysten ja harha-askelten jälkeen, havahduttiin tekoälyn saralla siihen, että ihmismäisesti toimivan ”vahvan” tekoälyn ensisijainen tavoittelu ei olisi käytännöllisesti kannattavaa, eikä nykyteknologialla edes kovinkaan toteuttamiskelpoista. Tällöin oivallettiin, että tekoälyn toimintaperiaatteella ei ole juurikaan merkitystä, kunhan se vain osaa suorittaa sille annetut tehtävät mahdollisimman tehokkaasti. Tämän seurauksena erilaisia ”heikon” tekoälyn sovellutuksia alettiin nähdä hyödynnettävän enenevissä määrin. (Jones 2003, 6.)

Kenties yksi legendaarisimmista 90-luvun saavutuksista tekoälyn saralla oli, kun IBM:n kehittämä *Deep Blue*-tietokone (kuva 2) onnistui vuonna 1997 voittamaan shakkiottelun lajin silloista maailmanmestaria, Garri Kasparovia, vastaan. Deep Blue ja Kasparov olivat ottaneet kertaalleen yhteen jo vuotta aikaisemmin, mutta tuolloin Kasparovin onnistui voittaa Deep Blue. Tapaus sai aikanaan paljon julkisuutta ja sitä pidetään nykyisin yhtenä tekoälyn kehityksen tärkeimmistä virstanpylväistä. (IBM 2003.)

KUVA 3. IBM Deep Blue (Love 2011)

Heinäkuussa 2007 Albertan yliopiston tutkijat onnistuivat puolestaan ratkaisemaan tammipelin *Chinook*-nimisen ohjelman avulla. Tämä tarkoitti käytännössä sitä, että ohjelma tunsi kaikki pelin mahdolliset pelitilanteet ja pystyi ennustamaan pelin lopputuloksen mistä tahansa pelitilanteesta lähtien. Tämän perusteella pystyttiin osoittamaan, että tammipeli voi päättyä vain tasapeliin, mikäli molemmat pelaavat osapuolet pelaavat peliä täydellisesti. (Sreedhar 2007.)

Viime vuosikymmenten aikana ovat kehittyneet muun muassa seuraavat tekoälyn osa-alueet: puheenymmärtäminen, tiedonlouhinta, diagnostiikka, robotiikka, tietokonenäkö, tietämyksen esittäminen, älykkäät agentit, suunnittelu ja aikataulutus, pelien pelaaminen, itseohjautuvuus, logistiikka, kielenymmärtäminen sekä ongelmanratkaisu. (Russell & Norvig 2003, 26–28.)

2.3 Sovellustapoja

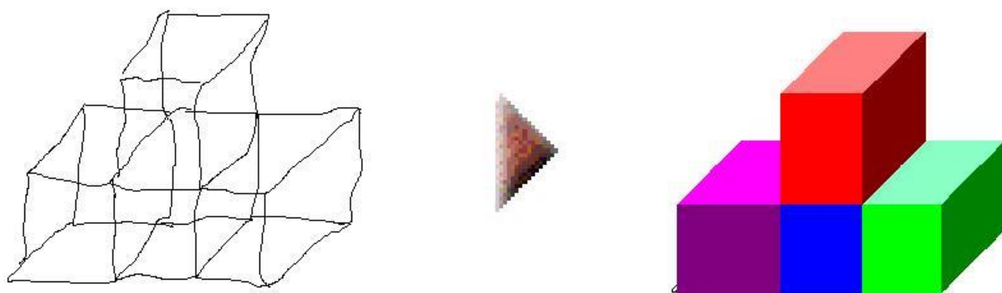
Kaikista tekoälyn mahdollisista soveltamistavoista on lähes mahdotonta tehdä kattavaa selvitystä, koska erilaisten sovellusten kirjo on valtava. Tekoälyä voidaan hyödyntää lähes kaikissa ongelmanratkaisutilanteissa. Tämän lisäksi erilaisten käyttötarkoitusten välillä voi ilmetä päällekkäisyyksiä, sillä joidenkin ongelmien ratkaisemiseen voidaan tarvita useita tekoälyn tekniikoita. Tässä luvussa on kuitenkin listattuna joitakin tyypillisiä ongelmatapauksia, joita tekoälyn avulla pyritään ratkaisemaan.

Pelien pelaaminen on yksi vanhimmista ja pisimmälle kehittyneistä tekoälyn sovelluksista. Monet pelejä pelaavat tekoälyt ovat nykyisin jo niin tehokkaita, että ne vastaavat tasoltaan ammattilaispelaajia. (Russell & Norvig 2003, 162.) Vaikka jotkin tutkijat pitävätkin pelien pelaamista epäolennaisena tekoälyn kehityksen kannalta, se kiinnostaa useita tutkijoita sen haasteellisuuden vuoksi (mts. 180–181). Luultavasti tunnetuin esimerkki pelejä pelaavasta tietokoneesta on IBM:n kehittämä Deep Blue-shakkitietokone. Pelien pelaamisen lisäksi tekoäly on myös tärkeä osa moderneja tietokonepelejä, mutta tästä aiheesta kerron kattavammin myöhemmin.

Asiantuntijajärjestelmät ovat tietokonejärjestelmiä, joiden tehtävänä on pystyä vastaamaan samanlaisiin kysymyksiin ja ratkaisemaan samanlaisia ongelmia, kuin mitä oikea asiantuntijakin pystyisi ratkaisemaan. Järjestelmän toiminta perustuu pitkälti

syy-seuraus-päättelyyn, jossa järjestelmä utelee käyttäjältä lisätietoja, kunnes sillä on käytettävissään tarpeeksi tietoa ratkaistakseen käyttäjän ongelma. Asiantuntijajärjestelmän toteuttamiseen vaaditaan paljon apua oikealta asiantuntijalta, jonka tehtävänä on syöttää järjestelmään kaikki ongelmanratkaisuun vaadittava informaatio. Toimivan asiantuntijajärjestelmän toteuttaminen voi kuitenkin olla toisinaan ongelmallista. Ensinnäkään asiantuntijat eivät välttämättä pysty yksiselitteisesti perustelemaan päätöksentekoaan, mikä puolestaan on tietokoneen näkökulmasta välttämätöntä. Toisekseen saattaa olla vaikeaa muuttaa asiantuntijan antama tieto sellaiseen muotoon, että siitä olisi hyötyä asiantuntijajärjestelmässä. (Brookshear 2003, 479.)

Tietokonenäkö on tieteen ala ja teknologia, jolla pyritään analysoimaan visuaalista dataa ja muuttamaan se tietokoneelle helpommin käsiteltävään muotoon. Tarkoituksena on löytää esimerkiksi kameroiden välityksellä saadusta kuvasta erilaisia rakenteita ja kuvioita, jotka tietokoneohjelma pystyy tulkitsemaan tuntemikseen objekteiksi (kuva 4). Tietokonenäköä hyödynnetään esimerkiksi robotiikassa ja käsialan tunnistuksessa. (Vision Recognition 2003.)



KUVA 4. Objektien tunnistus (Vision Recognition 2003)

Puheentunnistus on tietokonenäön tavoin koneelliseen havainnointiin liittyvä tekoälyteknologia. Puheentunnistuksesta erityisen vaikeaa tekee se, että äänestä ei ole yhtä helppoa löytää yhteneviä kuvioita kuin visuaalisesta datasta. Tunnistusta vaikeuttavat taustamelu ja erilaiset lausuntatavat. (Russell & Norvig 2003, 568.)

Robotiikan saralla hyödynnetään tekoälyä laajamittaisesti. Roboteilla tarkoitetaan älykkäitä agentteja, jotka kykenevät olevaan fyysisessä vuorovaikutuksessa. Riippuen siitä, minkälaista robottia tavoitellaan, vaaditaan sen toteuttamiseen jonkin asteista

havainnointia, päätöksentekoa sekä toiminta- ja liikuntakykyä. Suurin osa nykyisistä roboteista on suunniteltu tekemään töitä ja korvaamaan ihmistyövoima äärimmäisen raskaiden tai vaikeiden työtehtävien osalta. Tutkimusta tehdään kuitenkin myös erilaisten lelurobottien sekä ihmismäisten humanoidirobottien parissa. (Niemueller & Widyadharma 2003, 2–12.) Kuvassa 5 on Honda Asimo, Hondan kehittämä humanoidirobotti.

KUVA 5. Honda Asimo- humanoidirobotti (Honda 2011)

Tekoälyä on myös käytetty onnistuneesti lääketieteellisen diagnoosin antamiseen. Tämän tyyppisen ohjelman toiminta on perustunut pitkälti keinotekoisiin neuroverkkoihin. Näillä neuroverkoilla on kyetty toteuttamaan oppiva tekoäly, joka osaa päätellä, mikä on terveelle ihmiselle normaalia, ja mikä ei. (Bond 2010.)

Muita tekoälyn yhteydessä mainittuja tekniikoita ovat muun muassa tiedonlouhinta ja automatisoitu suunnittelu ja aikataulutus. Tiedonlouhinnalla tarkoitetaan tärkeän tiedon suodattamista valtavista tietomassoista, kun taas automatisoitu suunnittelu ja aikataulutus koostuvat erilaisista menetelmistä, joiden avulla järjestelmät pystyvät automaattisesti valmistelemaan ja organisoimaan erilaisia työprosesseja älykkäiden agenttien, kuten robottien tai muiden autonomisten laitteiden, suoritettavaksi. (Palace 1996.)

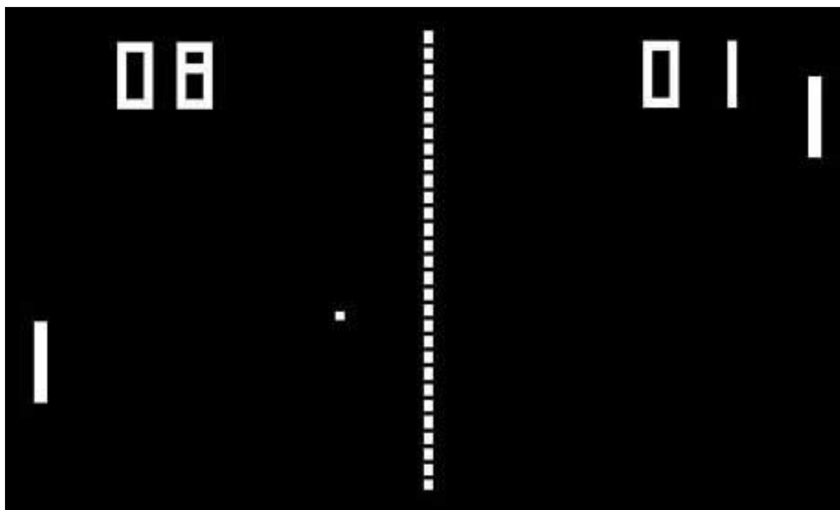
3 TEKOÄLY TIETOKONEPELEISSÄ

Tässä luvussa esitellään tekoälyn sovellutuksia tietokonepeleissä. Ensimmäisenä käyn läpi hiukan pelien historiaa ja selvitän, kuinka tekoälyn kehittyminen on vaikuttanut aikojen saatossa pelien kehittymiseen ja päinvastoin. Tämän jälkeen kerron tarkemmin siitä, mitä pelitekoäly oikeastaan pitää sisällään ja mitä erilaisia tekoälyyn liittyviä tekniikoita peleissä oikein käytetään.

3.1 Pelitekoälyn historia

Pelitekoäly on käsitteenä lähes yhtä vanha kuin tekoäly yleensä. Ensimmäiset pelejä pelaavat tekoälyohjelmat tehtiin jo 50-luvulla ja ne kuuluivat maailman ensimmäisiin valmiisiin tekoälyohjelmiin. Itse asiassa jotkut pitävät Cristopher Stracheyn tekemää tammipelin tekoälyä maailman ensimmäisenä toimivana tekoälyohjelmana. Kyseinen ohjelma osasi pelata sujuvalla nopeudella erän tammea alusta loppuun vuonna 1952. Dietrich Prinz ohjelmoi puolestaan samoihin aikoihin maailman ensimmäisen shakkia pelaavan ohjelman. (Copeland 2000c.)

Pelitekoälyn kehitys alkoi saada kunnolla tuulta alleen ensimmäisten videopelien myötä 1970-luvun alussa. Näiden joukossa oli muun muassa vuonna 1972 valmistunut *Pong*. Pong on yksinkertainen 2-ulotteinen kahden pelaajan tennispeli, joka koostuu kahdesta ohjattavasta mailasta ja jatkuvasti liikkuvasta pallosta (kuva 6).



KUVA 6. Pong (Parker & King 2008)

Pongin tarkoituksena on lyödä palloa mailoilla ja saada pallo kulkemaan ulos vastustajan puoleisesta peliruudun reunasta. Tekoälyn roolina pelissä on ottaa toisen pelaajan rooli silloin, kun peliä pelaa vain yksi pelaaja. Tekoälyn vaikeustasoa säätämällä pelaaja pystyy määrittämään, kuinka todennäköisesti tekoälypelaaja onnistuu torjumaan pallon. (Wexler 2002.)

Tekoälystä tuli keskeinen osa tietokonepelejä 1970-luvun lopulla, kun peleistä alettiin tehdä enenevissä määrin yksinpelattavia. Sen sijaan, että pelaajat haastaisivat pelissä toisensa, he joutuisivatkin haastamaan itse pelin. Tällaisen pelityypin popularisoi vuonna 1978 valmistunut menestyksekäs kolikkopeli *Space Invaders*. Pelin tavoitteena on ohjata yksinäistä avaruusalusta ja torjua ampumalla näytön yläreunasta lähestyvä muukalaisten invaasio (kuva 7). Pelistä tuli Japanissa niin suosittu, että se aiheutti aikanaan pulan 100 jenin kolikoista. (Old-computers 2013.)

KUVA 7. Space Invaders (Retro Gamer 2013)

Vaikka Space Invadersin avaruusoliot osasivatkin liikkua ja uhata pelaajan avaruusalusta, niiden käyttäytymistä ei voinut luonnehtia kovinkaan älykkääksi. Ne eivät millään tavoin reagoineet ympäristöön eivätkä pelaajan toimintaan, vaan seurasivat ainoastaan niille ennalta määrättyjä liikekuvioita. Vuonna 1979 julkaistussa ammuntapelissä *Galaga* tekoäly oli jo aavistuksen hienostuneempi, sillä yksittäiset viholliset kykenivät toimimaan jo jossain määrin itsenäisesti. (Stahl 2013.)

Namcon vuonna 1980 julkaisemassa pelissä *Pac-Man* (kuva 8) viholliset on saatu vaikuttamaan entistäkin nerokkaammilta, vaikka pelin tekoäly ei teknisesti kovin pitkälle kehittynyt olekaan. Pelissä esiintyvät haamuviholliset osaavat navigoida pelin sokkeloissa sujuvasti ja osaavat tilanteen mukaan joko jahdata pelaajaa tai paeta. Jokaisella vihollisella on myös oma yksilöllinen luonteensa, joka vaikuttaa niiden käyttäytymiseen. (Birch 2010.)

KUVA 8. Pac-Man (Retro.mmgn.com 2012)

Vielä 80-luvulla pelien tekoäly oli pitkälti skriptattua eli ennalta määrättyä ja tekoäly lähinnä toisti jatkuvasti samoja toimintoja uudestaan ja uudestaan. Syy oli enimmäkseen luultavasti sen aikaisten laitteiden rajoitteellisessa suorituskyvyssä ja tehokkaiden toteutusmenetelmien puutteessa. Tilanne kuitenkin muuttui 90-luvulla tekoälyn ja teknologian kehittyessä.

Vuonna 1992 julkaistiin uraauurtava reaaliaikainen strategiapeli *Dune II* (kuva 9), joka vaati tekoälyltä paljon enemmän, kuin mihin oli tähän asti totuttu. Tekoälypelaajan tuli kyetä rakentamaan oma tukikohta sekä oma armeija, löydettävä pelaajan tukikohta ja pystyttävä taistelemaan pelaajan armeijan kanssa. Jälkeenpäin ajatellen Dune II:n tekoäly oli itse asiassa hyvin heikko: tekoälypelaaja ei armeijallaan muuta tehnyt kuin rynnistänyt suoraan pelaajan tukikohtaa kohti, sen enempää taktiikoista piittaamatta. Heikkouksiensa ylittämiseksi tietokonepelaaja myös huijasi pelissä: esimerkki-

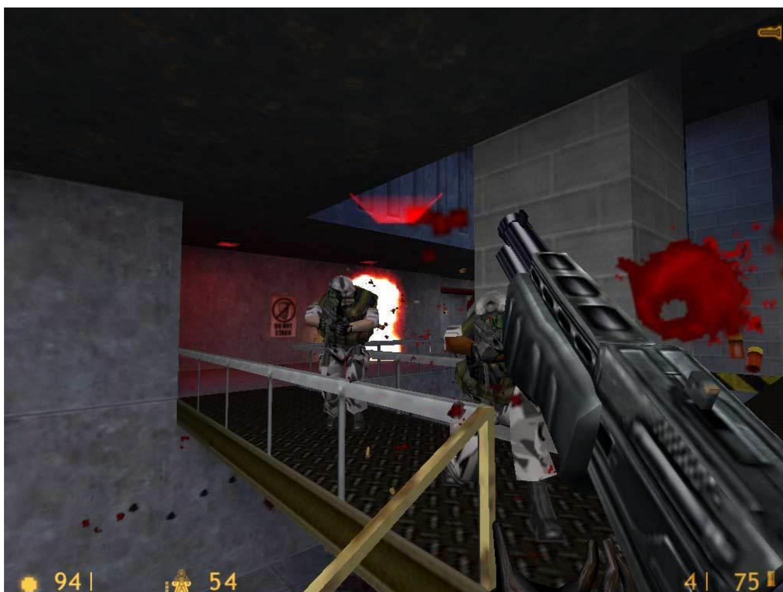
nä tietokonepelaajalla oli loputtomasti resursseja armeijan ja tukikohdan rakentamista varten, toisin kuin pelaajalla. Peli oli kaikesta huolimatta aikansa saavutus ja loi perustan tulevia strategiapelejä varten. (Schwab 2008, 116–117.) Muita aikaisia RTS (real-time strategy) lajin varhaisimpia edustajia ja uranuurtajia olivat muun muassa *Herzog Zwei* (1989), *Warcraft* (1994) ja Dune II:n tekijöiden seuraava strategiapeli, *Command & Conquer* (1995).

KUVA 9. Dune II (Massaad 2011)

Ei kestänyt kuitenkaan kovinkaan kauaa, ennen kuin tekoälystä pystyttiin tekemään paljon hienostuneempaa ja käytökseltään arvaamattomampaa. Vuonna 1996 julkaistu *Creatures* edisti pelitekoälyä ennennäkemättömillä tavoilla ja avasi uusia mahdollisuuksia pelisuunnittelun saralla. Creaturesissa pelaaja on vuorovaikutuksessa kuvitteellisten Norn-otusten kanssa ja pyrkii opettamaan Norneja selviytymään itsenäisesti, ilman pelaajan väliintuloa. (Champandard 2007.) Samankaltaista konseptia suuremmassa mittakaavassa noudatti vuonna 2001 julkaistu *Black & White* (kuva 10), jossa pelaaja pyrkii erilaisten luonnonmullistusten avulla vaikuttamaan kokonaisten ihmiskansojen elämään (Wexler 2002).

KUVA 10. Black & White (Thursten 2013)

Uusien, innovatiivisten tekoälypainotteisten pelikonseptien lisäksi 90-luvulla tehtiin edistystä myös perinteisempien pelityyppien tekoälyn saralla. Vuonna 1998 julkaistu *Half-Life* (kuva 11) nosti rimaa räiskintäpelien osalta niin tarinankerronnan, kenttä-suunnittelun kuin tekoälynkin osalta. Ensimmäistä kertaa räiskintäpeleissä viholliset osasivat oikeasti tukea toistensa hyökkäystä, pyrkiä hyökkäämään pelaajaa kohti takaa tai sivusta, ottaa suojaa tulitaistelun aikana ja houkutella pelaajan pois suojan takaa esimerkiksi käsikranaattien avulla. Pelin viholliset pystyivät myös kommunikoimaan keskenään ja tiedottamaan toisiaan aikeistaan. Tätä tietoa pelaajakin pystyi käyttä-mään omaksi edukseen. (Dominguez 2012.)



KUVA 11. Half-Life (Steam 2013)

Pelien tekoäly on jatkanut tasaisesti kehitystään ja pelikehittäjät ovat pyrkineet keksimään lisää uudenlaisia tapoja soveltaa tekoälyä pelinkehityksessä. Esimerkiksi vuonna 2008 julkaistussa räiskintäpelissä *Left 4 Dead* oli omalaatuinen tekoälyominaisuus, jota markkinoitiin nimellä *AI Director.* Ominaisuuden tarkoituksena oli taata jokaiselle pelikerralle omanlaisensa pelikokemus muuttamalla vihollisten vahvuutta ja lukumäärää sekä saatavilla olevien auttavien esineiden määrää sen perusteella, kuinka hyvin pelaajat suoriutuvat pelissä (Booth 2009). Vuonna 2012 julkaistussa pelissä *Assassin's Creed III* tekoälyä käytettiin puolestaan niinkin triviaalilta vaikuttavaan asiaan, kuin simuloimaan päähahmon kävelyanimaatio mahdollisimman aidontuntuisesti (Griliopoulos 2013).

3.2 Käyttötarkoitukset peleissä

Tekoälyn merkitys nykyajan tietokonepelissä on suurempi kuin voisi aluksi luulla. Miellämme usein pelitekoälyn liittyvän lähinnä tietokoneen ohjaamien peliobjektien liikutteluun ja hallintaan, mutta tekoälyllä on monissa peleissä muitakin tehtäviä. Emme välttämättä edes aina tiedosta tekoälyn olemassaoloa ja toisinaan voi olla vaikeasti tunnistettavissa, onko jonkin pelitapahtuman takana minkäänlaista älykkyyttä vai ei.

Tässä luvussa kerron joistakin tyypillisimmistä peleissä käytetyistä tekoälytekniikoista. Kaiken kaikkiaan tekniikoita on lukematon määrä, mutta kaikki eivät ole kovinkaan laajassa käytössä. Pelinkehittäjät pyrkivät jatkuvasti innovoimaan ja keksimään uusia pelikokemuksia ja niiden myötä syntyy jatkuvasti myös uusia tekoälytekniikoita.

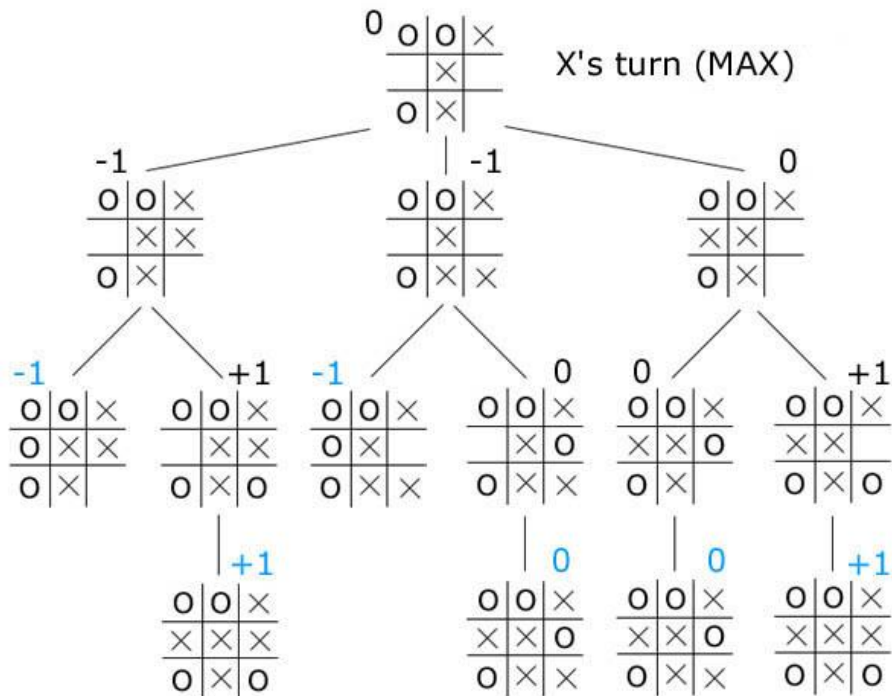
3.2.1 Pelien pelaaminen

Pelien pelaaminen on vanha ja pitkälle kehittynyt tekoälytutkimuksen osa-alue. Kuten on jo aiemmin mainittu, varhaisimmat niistä tehtiin jo 50-luvulla ja ne pystyivät pelaamaan perinteisiä lautapelejä, kuten shakkia ja tammea. Pelejä pelaavat tekoälyt ovat sittemmin kehittyneet huimasti, mutta parantamisen varaa on yhä. Pelejä pelaavat tekoälyt kiinnostavat tutkijoita vieläkin, koska edistyksellisen pelitekoälyn kehittäminen on haasteellista. Tämän lisäksi pelit toimivat mainioina testiympäristöinä, koska ne antavat hyvin palautetta tekoälyn suorituksesta. (Russell & Norvig 2003, 162.)

Tietokonepeleissä tekoälyn ohjaamia pelaajaentiteettejä kutsutaan usein boteiksi (Randar 2012).

Käytettävien menetelmien tarpeellisuus riippuu pitkälti siitä, minkälaista peliä tekoälyn pitää osata pelata. Pongin kaltaisessa yksinkertaisessa tietokonepelissä luultavasti riittää, että tietokoneen ohjaama maila osaa jollakin tavoin seurata pallon liikettä. Modernissa tietokonestrategiapelissä tekoälyn on puolestaan varmastikin hyödynnettävä sekä useampia, että monimutkaisempia, menetelmiä. Vuoropohjaisissa peleissä, kuten lautapeleissä, tekoälyn toiminta perustuu pitkälti erilaisiin pelipuihin ja hakualgoritmeihin, joilla pelipuusta haetaan tietokoneelle pelitilanteen kannalta optimaalisin siirto (Adamchik 2009).

Esimerkiksi ristinolla voidaan ratkaista hyödyntämällä niin kutsuttua minimax-algoritmia. Minimax-algoritimia voidaan käyttää silloin, kun pelataan peliä, jossa yhden pelaajan etu on toisen pelaajan tappio. Tällaisia pelejä kutsutaan myös nollasummapeleiksi. Minimax-algoritmi on hakualgoritmi, joka osaa etsiä pelistä juuri sellaiset siirrot, mistä on kaikista eniten hyötyä. Menetelmän toimintaperiaate on, että jokaisella eri pelitapahtumalla on oma pistearvonsa, ja algoritmi hakee kaikki ne pelitapahtumat, joista saa eniten pisteitä. On syytä huomioida, että minimax ei sovi kovin monimutkaisten pelien ratkaisemiseen. Jo pelkästään ristinollassa on yhteensä 362880 mahdollista läpikäytävää pelitilannetta. (Adamchik 2009.) Kuvassa 12 on nähtävissä loppusuoralla olevan ristinollapelin pelipuu ja pisteet, jotka tietokonepelaaja voisi kustakin siirrosta saada minimaxin mukaan.



KUVA 12. Osittainen ristinollan pelipuu (Lin 2003)

Useissa tietokonepeleissä ja etenkin laajamittaisissa strategiapeleissä on usein tyypillistä, että tekoäly huijaa. Sen sijaan, että tekoäly pelaisi peliä samoilla ehdoilla kuin pelaaja, sille voidaan antaa tiettyjä etuuksia pelin alusta alkaen. Esimerkiksi strategiapelissä tekoälypelaajalle saatetaan antaa etukäteen tietoja muista pelaajista, kuten muiden pelaajien sijainnit pelikartalla. Huijaamalla tekoälypelaajasta tehdään kilpailukykyisempi taitavia ihmispelaajia vastaan ja pyritään tekemään tekoälystä toiminnallisesti yksinkertaisempi. Vaikka huijaaminen onkin usein tarpeen, sen pitää olla maltillista, eikä se saa olla ihmispelaajalle liian ilmeistä. Liian ilmeisesti huijaavaa tekoälyä pidetään yleisesti huonona tekoälynä. (Bourg & Seemann 2004, 3–4.)

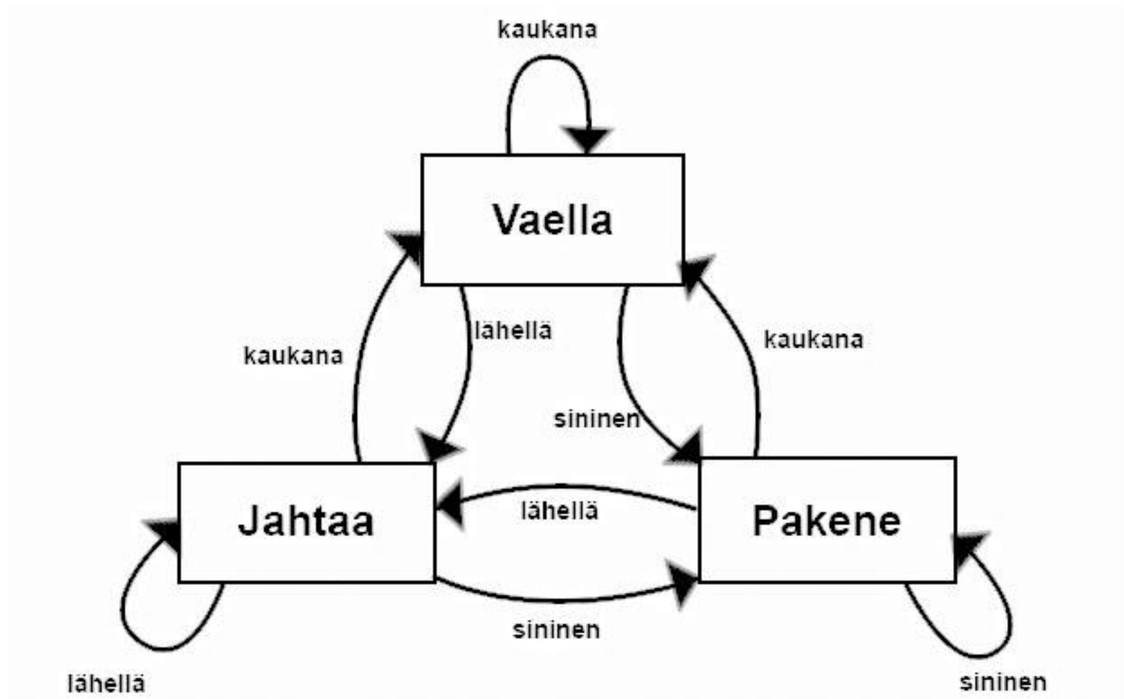
Yksi suurimmista pelejä pelaavien tekoälyjen ongelmista on, että ne ovat pitkälti erikoistuneita. Vaikka IBM Deep Blue pystyykin pelaamaan shakkia huippupelaajan tasolla, se tuskin pystyisi pelaamaan sujuvasti vaikkapa ristinollaa. Tämän vuoksi onkin alettu tutkia yleistä pelejä pelaavaa tekoälyä, englanniksi *general game playing* (GGP). Toimivan GGP- tekoälyn toteutus on hyvin haasteellista. Sen toiminta ei voi perustua tiettyihin peleihin tarkoitettuihin algoritmeihin, koska se ei voi ennen pelin alkamista tietää, minkälaista peliä sen pitää pelata. (Genesereth 2013.)

Vuonna 2013 kehitettiin tutkimusmielessä kiehtova tekoälyohjelma, joka oppi yrittämisen ja erehtymisen kautta pelaamaan vanhoja NESin eli Nintendo Entertainment Systemin pelejä. Ohjelman perusideana oli, että ohjelmalle syötettiin lyhyt pätkä syötetietoa siitä, kuinka ihmispelaaja pelaisi. Analysoimalla tätä syötetietoa ohjelman tuli päätellä, mikä pelatun pelin tavoite oli, ja kuinka se saavutettaisiin. Vaikka ohjelma oppikin lopulta pelaamaan melko sujuvasti erilaisia yksinkertaisia tasohyppelypelejä, kuten Super Mario Brothersia ja Hudson's Adventure Islandia, sen käytös Tetriksessä oli mitä mielenkiintoisin: tekoäly ei millään osannut päätellä, miten Tetristä pelataan. Lopulta tekoäly kuitenkin oivalsi, että ainoa tapa olla häviämättä Tetriksessä on lopettaa pelaaminen kokonaan. (Murphy 2013.)

3.2.2 Äärellinen tilakone

Äärellinen tilakone eli finite state machine (FSM) on niin sanottu abstrakti kone tai automaatti, joka koostuu äärellisestä määrästä ennalta määritettyjä tiloja. Äärelliseen tilakoneeseen sisältyvät ehdot, jotka määrittävät, missä tilassa tilakone on. Tilat itsessään puolestaan määrittelevät, kuinka tilakone käyttäytyy. Esimerkiksi Pac-Manin haamuviholliset ovat äärellisiä tilakoneita, joilla on kolme mahdollista tilaa: vapaa vaellus, pelaajan jahtaaminen ja pakeneminen. Äärellisiä tilakoneita käytetään peleissä hyvin laajamittaisesti niiden yksinkertaisuuden ja tehokkuuden vuoksi. (Bourg & Seemann 2004, 165.)

Kuvassa 13 on nähtävissä esimerkki mahdollisesta tilakaaviosta, joka Pac-Manin haamulla voisi olla. Lähtökohtaisesti haamu vain vaeltelee ympäriinsä, mikäli Pac-Man ei ole tarpeeksi lähellä. Jos Pac-Man on tarpeeksi lähellä, haamu aloittaa jahtaamisen. Mikäli haamu muuttuu siniseksi eli Pac-Man voi syödä sen, haamu lähtee karkuun. Mikäli haamu ei ole sininen, ja Pac-Man on liian kaukana, haamu siirtyy takaisin alkuperäiseen vaellustilaan.



KUVA 13. Äärellistä tilakonetta kuvaava tilakaavio

Äärellistä tilakonetta voidaan tarpeen tullen myös hienosäätää ja laajentaa. Lähtökohtaisesti äärelliset tilakoneet ovat melko deterministisiä, eli niiden toiminta on ennalta arvattavaa ja tilasiirrokset staattisia. Lisäämällä tilakoneen toimintaan esimerkiksi satunnaisuutta on mahdollista luoda arvaamattomampia ja luontevammin toimivia pelihahmoja. Yhdellä pelihahmolla voi olla samanaikaisesti myös useita eri tiloja, joiden avulla voidaan hienosäätää hahmon käyttäytymistä. (Bailiie de-Byl 2004, 237–238.)

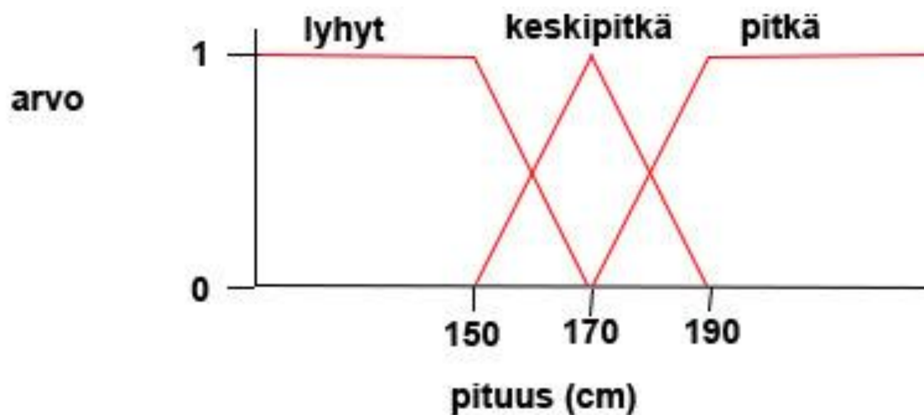
3.2.3 Sumea logiikka

Sumean logiikan avulla pyritään mallintamaan päätöksentekoa niissä tilanteissa, kun jokin väittämä ei välttämättä ole totuusarvoltaan suoranaisesti tosi tai epätosi, vaan jotakin siltä väliltä. Sumean logiikan mukaan jokin totuusarvo voi olla esimerkiksi ”lähes tosi”. Sumeassa logiikassa totuusarvot esitetään liukuvalla asteikolla 0:n ja 1:n eli epätoden ja toden välillä. (Bailiie de-Byl 2004, 30.)

Lisäämällä sumeaa logiikkaa äärellisiin tilakoneisiin pelikehittäjät pystyvät tekemään niiden toiminnasta arvaamattomampaa ja täten mahdollisesti kiinnostavampaa. Sumean logiikan avulla tilakoneisiin voi myös syöttää epämääräisempiä totuusehtoja. Tämä

vähentää myös niiden toteuttamiseen vaadittavaa työtaakkaa, koska jokaiselle arvolle ei tarvita omaa sääntöä. (Bourg & Seemann 2004, 4.)

Kuvassa 14 on kuvaaja, joka kertoo, kuinka pitkä henkilön on oltava ollakseen lyhyt, keskipitkä tai pitkä. Kuvaajan mukaan 150 cm pitkä henkilö on lyhyt, 170 cm pitkä taasen keskipitkä, ja näiden välillä oleva on jotain siltä väliltä. Esimerkiksi 160 cm pitkää henkilöä voisi kuvailla lähes keskipitkäksi. Tällaista todellisen numeroarvon sanallista nimeämistä kutsutaan sumeuttamiseksi, englanniksi fuzzification (Bourg & Seemann 2004, 193). Jos sumea arvo halutaan puolestaan muuttaa numeraaliseksi, prosessia kutsutaan täsmällistämiseksi, englanniksi defuzzification (mts. 203).



KUVA 14. Esimerkki sumean pituusmuuttujan saamista arvoista

Vaikka sumeaa logiikkaa käytetäänkin tietokonepeleissä, sen laajamittainen tai erityisen syventävä käyttö ei ole kovin yleistä, tai ainakaan tällaisesta ei ole juurikaan mainintoja. Joitakin esimerkkipelejä ovat muun muassa S.W.A.T. 2, Close Combat ja The Sims. S.W.A.T. 2:ssa sumeaa logiikkaa on käytetty vihollisten persoonallisuuksien mallintamiseen ja sen avulla niiden käytöksestä on saatu spontaanimpaa. Close Combat on puolestaan sotapeli, jossa sumean logiikan avulla käydään läpi lukuisia muuttujia, joiden perusteella päätetään, tehdäänkö jokin toiminto vai ei. Yksi tällainen muuttuja voi olla vaikkapa sotilaiden moraali. The Simsissä, kuten S.W.A.T. 2:ssa, sumeaa logiikkaa on käytetty pelihahmojen persoonallisuuksien mallintamiseen. Nämä persoonallisuudet puolestaan määrittelevät, miten hahmot ovat vuorovaikutuksessa keskenään ja ympäristönsä kanssa. (Pirovano 2012.)

3.2.4 Polunetsintä

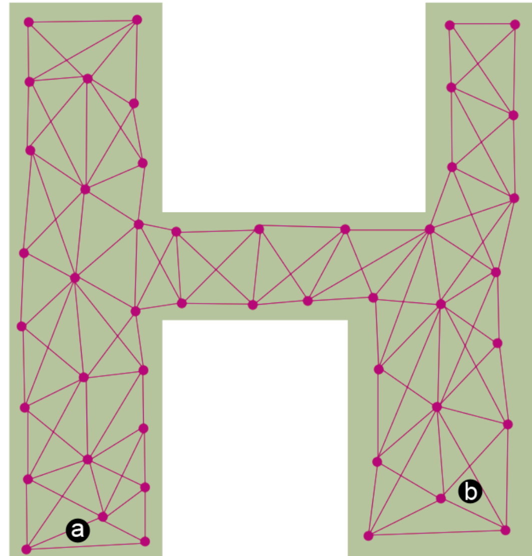
Polunetsinnällä – tai vaihtoehtoisesti reitinetsinnällä – tarkoitetaan yksinkertaisimmillaan prosessia, jossa pelihahmo siirretään alkusijainnistaan johonkin toiseen sijaintiin. Matemaattisesta näkökulmasta katsottuna polunetsinnässä on tarkoituksena löytää jokin kulkukelpoinen reitti kahden eri pisteen välillä. Se, mitä kulkukelpoisella reitillä tarkoitetaan ja minkälainen haetun reitin tulee olla, riippuu käyttötarkoituksesta. (Bourg & Seemann 2004, 96.)

Polunetsinnällä on paljon mahdollisia käyttötarkoituksia ja se on usein tärkeässä roolissa monissa tietokonepeleissä. Siksi sen toimintaan kiinnitetäänkin paljon huomiota. Erityisen merkittävä rooli sillä on strategiapeleissä, joissa armeijoiden on osattava kulkea maastossa ja kyettävä kiertämään ylitsepääsemättömät esteet. Myös toimintapelien vihollisten on kyettävä kulkemaan pelikentillä jäämättä jumiin esteisiin tai seiniin. (Bourg & Seemann 2004, 4.) Karkeasti sanoen voisi kai väittää, että polunetsintää tarvitaan aina, kun halutaan jonkin pelihahmon vaeltelevan pelissä itsenäisesti ilman pelaajan väliintuloa.

Tyypillisesti polunetsintäongelmia ratkaistaan erilaisten polunetsintäalgoritmien avulla. Kun polunetsintää lähdetään toteuttamaan, on tärkeää, että ongelmaa varten valitaan sopiva ratkaisutapa. Vaikka jotkut algoritmit suoriutuvatkin yhdessä tapauksessa erinomaisesti, ne eivät välttämättä toimi odotusten mukaisesti jossakin toisessa. (Bourg & Seemann 2004, 96.) Koska erilaisia käyttötilanteita on lukemattomia ja erilaisia ratkaisualgoritmeja vaikka kuinka, keskityn esittelemään näistä kolme: yksinkertaisen solmumenettelyn, Dijkstran algoritmin ja A* algoritmin.

Luultavasti yksinkertaisin keino toteuttaa toimiva polunetsintä on hyödyntää yksinkertaista solmurakennetta. Tällöin pelikentälle lisätään useita pisteitä eli solmuja, jotka liitetään toisiinsa suorilla linjoilla. Tällaisen rakenteen idea on siinä, että pelihahmo liikkuu pelimaailmassa eri solmujen välillä. Pelihahmo liikkuu aloitussolmusta seuraavaan solmuun ja siitä taas eteenpäin seuraavaan, kunnes hahmo saapuu tavoitesolmuun. Tämä menetelmä on toimiva ja verrattain helppo toteuttaa, mutta ongelmana on, että liikkeestä tulee kankeaa ja robottimaista. Tekniikkaa on kuitenkin mahdollista hienosäätää niin, että liikkeestä tulisi sulavampaa ja luontevamman näköistä. (Mc-

Shaffry 2012, 636–637.) Kuvassa 15 on esimerkki eräänlaisesta solmupisteillä toteutetusta navigaatioverkosta.



KUVA 15. Navigaatioverkko (Grenier 2011)

Dijkstran algoritmi on hakualgoritmi, joka hakee kaikki mahdolliset reitit aloituspisteestä lopetuspisteeseen ja valitsee niistä kaikkein lyhyimmän. Toisin kuin monet muut polunetsintäalgoritmit, Dijkstran algoritmi ylläpitää listaa laskemistaan etäisyyksistä, ja valitsee aina sen reitin, josta loppupisteeseen on kaikista lyhin matka. Aina, kun pelihahmo etenee pelissä yhden pisteen verran, algoritmi tarkistaa, onko nykyinen reitti yhä kaikista lyhyin. Jos se ei ole, hahmo etsii uuden reitin ja lähtee kulkemaan sitä pitkin. Dijkstran algoritmin suurin etu on, ettei mikään muu polunetsintäalgoritmi kykene löytämään yhtä optimaalisia polkuja. Toisaalta Dijsktran algoritmi on myös hyvin raskas ja sen käyttämiseen tarvitaan paljon prosessointitehoa (Rabin 2009, 572.)

```
lisää aloituspiste avoimeen listaan
while (avoin lista EI ole tyhjä)
{
      tämänhetkinen piste = edullisin piste avoimessa listassa
      if (tämänhetkinen piste = maalipiste)
      {
         reitti valmis
      }
      else
      {
         siirrä tämänhetkinen piste suljettuun listaan
         tutki nykyisen pisteen vieressä olevat pisteet
         for (vieressä olevat pisteet)
         {
            if (piste ei ole avoimessa listassa AND piste ei ole suljetussa listassa
            AND piste ei ole este)
            {
               siirrä piste avoimeen listaan ja laske arvo
            }
         }
      }
}
```

KUVA 16. A* hakualgoritmin pseudokoodia

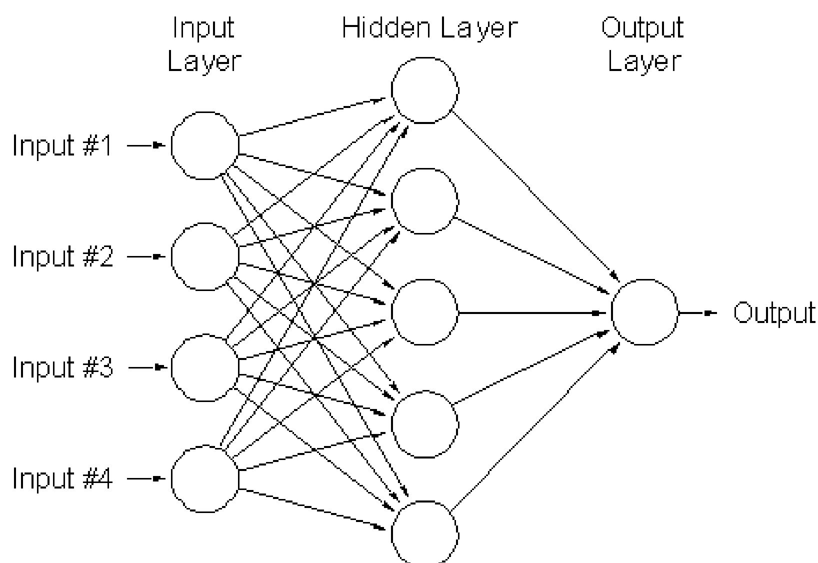
A* algoritmi on yksi pelialan käytetyimmistä polunetsintäalgoritmeista, ellei kaikista käytetyin. Se on tehokas, verrattain kevyt laitteistonkäytön suhteen ja tehoaa hyvin useimpiin polunetsintäongelmiin. Kuten Dijkstran algoritmin tapauksessa, pelimaailma jaetaan useisiin pisteisiin, joiden välillä hahmot voivat liikkua. Algoritmin tavoitteena on löytää optimaalisin reitti alkupisteestä loppupisteeseen. Toisin kuin Dijsktran algoritmi, A* algoritmi ei pyri jatkuvasti etsimään varmasti parhainta reittiä. Sen sijaan A* algoritmin toiminta perustuu heuristiikkaan. Tämä tarkoittaa käytännössä sitä, että algoritmi yrittää parhaansa mukaan vain päätellä, missä paras reitti saattaisi olla. Jos algoritmilla ei ole käytössään tarpeeksi taustatietoa, algoritmi pyrkii lähinnä arvailemaan parhaan reitin yritys ja erehdys- menetelmällä. (Bourg & Seemann 2004, 126–133.) A* algoritmi on laajennettu variaatio Dijkstran algoritmista, ja heuristiikan pettäessä A* algoritmi toimii samalla toimintaperiaatteella (LaValle 2012). Kuvassa 16 on pätkä A* algoritmin pseudokoodia.

3.2.5 Keinotekoiset neuroverkot

Keinotekoiset neuroverkot ovat toistaiseksi tietokonepeleissä melko harvinaisia ja epätavanomaisia, mutta ne ovat mielestäni kuitenkin mainitsemisen arvoisia. Uskon nimittäin, että niiden merkitys tulee tulevaisuudessa kasvamaan kaikilla tekoälyn osa-alueilla, mukaan lukien tietokonepeleissä.

Neuroverkkojen perusajatuksena on niiden nimen mukaisesti jäljitellä luonnollisten hermostojen toimintaa, vaikka todellisuudessa nykyisin kovin tarkkaan jäljittelyyn ei oikeasti pyritäkään. Syöttämällä neuroverkolle tarpeeksi havaintoaineistoa se ennen pitkää oppii tekemään tilannekohtaisia johtopäätöksiä. Esimerkiksi autopeleissä tietokoneen ohjaamien autojen pitäisi pystyä hienosäätämään ajotapaansa sen mukaan, onko lähettyvillä muita autoja, kuinka mutkikas tie on ja minkälaisessa maastossa ajetaan. Neuroverkot ovat myös yksi mahdollinen tapa lisätä tekoälyyn oppimista. Pelitekoäly voi pelin edetessä arvioida omaa suoriutumistaan ja epäonnistuessaan yrittää jotakin muuta lähestymistapaa. (Russell & Norvig 2003, 736–737.)

Kuvassa 17 on esimerkkimalli eräänlaisesta yksinkertaisesta neuroverkosta. Neuroverkot koostuvat erilaisista toisiinsa kytketyistä neuroneista, joiden välille on määritelty kytkentäkohtaiset painotukset. Neuroverkolle annetaan alustavasti syötetietoja (input), jonka jälkeen syötesignaali kulkeutuu neuroneiden välityksellä johonkin tulosneuroniin (output). Edellä mainitussa autopeliesimerkissä syötetieto voisi olla mikä tahansa huomioitava tekijä pelissä, kuten auton nykyinen nopeus tai sijainti tiellä, kun taas tuloksena voisi olla jokin auton ohjaamiseen liittyvä toiminto, kuten nopeudensäätö, ratin kääntö tai vaihteen vaihto. (mts. 737–738.)



KUVA 17. Keinotekoinen neuroverkko (Hjørland 2005)

Huolimatta neuroverkkojen valtaisasta potentiaalista tietokonepeleissä, niitä on hyödynnetty hyvin vähän kaupallisissa peleissä. Tähän on lukuisia eri syitä. Ensinnäkin

perinteisemmät tekoälyn ohjelmointitavat ovat helpompia toteuttaa ja toimivat useimmiten paremmin nykypäivän tyypillisimmissä tekoälyongelmissa. Toisekseen toimivan neuroverkon toteuttaminen ja ”kouluttaminen” on hyvin monimutkaista ja työlästä, koska useimmissa peleissä tekoälyn tulee huomioida valtava määrä syötetietoa. Neuroverkko täytyy myös opettaa uudelleen aina, kun pelin toimintaa muutetaan oleellisesti. Viimeisimpänä ongelmana on, että neuroverkon toiminnallisuuden testaaminen on haastavaa ja aikaa vievää. Neuroverkkoja on kuitenkin hyödynnetty onnistuneesti esimerkiksi *Colin McRae's Rally 2:ssa* (2000) ja *Black & White:*ssa (2001).

4 ESIMERKKIEN TOTEUTTAMINEN

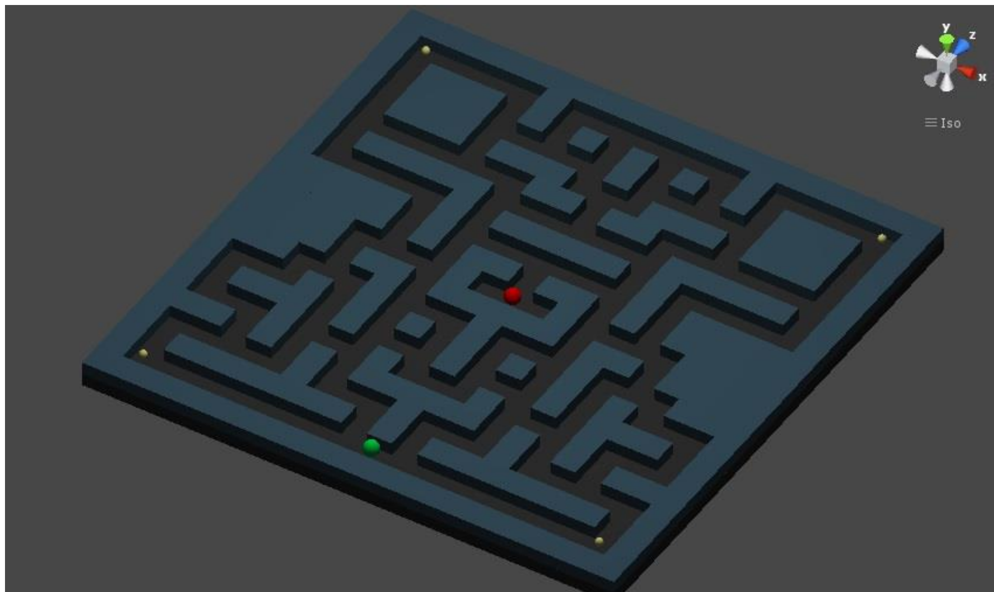
Tässä luvussa esittelen Unity-ohjelmistolla toteuttamiani pelidemoja, joissa keskeisenä pelimekaniikkana on jollakin tapaa pelin kulkuun vaikuttava tekoäly. Esimerkit itsessään ovat peleinä hyvin yksinkertaisia, koska niiden pääasiallinen tavoite on lähinnä esitellä tekoälyn toimintaa. Tämän vuoksi niiden pelisisältöön, grafiikoihin tai yleiseen ulkoasuun ei ole juurikaan panostettu. On myös syytä noteerata, että tekoälyn toteuttamisessa on keskitytty nimenomaan erilaisten menetelmien toteutettavuuteen ja soveltuvuuteen Unityssa, eikä niinkään itse tekoälyalgoritmien tutkimiseen.

Loppujen lopuksi päädyin tekemään kaksi erilaista esimerkkipeliä. Ensimmäinen esimerkki on Pac-Mania muistuttava sokkelopeli, jossa pelaajan pitää vältellä jahtaavaa vihollispalloa. Tämä esimerkki havainnoi tekoälyn toimintaa nopeatempoisessa pelissä, jossa tekoälyn pitää tehdä jatkuvasti nopeita päätöksiä. Toisena esimerkkinä on puolestaan kolmiulotteinen ristinolla. Tämä peli puolestaan havainnoi tekoälyn toimintaa strategisemmassa pelissä, jossa tekoälyllä on enemmän aikaa puntaroida erilaisia vaihtoehtoja.

Ohjelmointikielenä olen käyttänyt pääosin C sharpia (C#). Toinen varteenotettava vaihtoehto Unityssa olisi ollut JavaScript. JavaScriptilla on suurempi suosio Unity-yhteisössä ja siihen on helpommin saatavilla esimerkkejä, mutta päädyin kuitenkin lopulta valitsemaan C#:n. Päädyin tähän lähinnä sen takia, että halusin harjoitella C#-ohjelmointia.

4.1 Sokkeloesimerkki

Ensimmäinen toteuttamani esimerkki on yksinkertainen sokkelopeli, joka koostuu sokkelon lisäksi pelaajan ohjaamasta vihreästä pallosta, tekoälyn ohjaamasta punaisesta pallosta, sekä sokkeloon sijoitetuista ”*power-upeista*”. Kutsuttakoon niitä tässä yhteydessä vaikkapa voimapisteiksi. Pelin ideana on paeta punaista palloa, joka jahtaa pelaajaa jatkuvasti. Tilanne kuitenkin muuttuu hetkellisesti päinvastaiseksi, jos pelaaja onnistuu keräämään yhdenkin edellä mainituista voimapisteistä. Tällöin vihollispallo lopettaa pelaajan jahtaamisen ja yrittää sen sijaan paeta pelaajaa.



KUVA 18. Kolmiulotteinen sokkelo

Kuten kuvasta 18 näkee, esimerkissä käytetyt objektit koostuvat lähinnä erimuotoisista suorakulmioista ja palloista. Ne kaikki saa kätevästi luotua suoraan Unityssa, ilman erillistä 3D-mallinnusohjelmaa. Toisaalta pelin ulkonäköä olisi voinut helposti parantaa käyttämällä Unityn Asset Storesta löytyviä ilmaisia 3D-malleja. Yksinkertaiset 3D-muodot täyttävät kuitenkin tehtävänsä.

Pelin toteuttamiseen tarvitaan siis käytännössä kaksi erilaista tekoälyominaisuutta: reitinetsintä ja tilakone. Reitinetsintä on oleellinen, jotta vihollinen osaa kulkea sokkelossa törmäilemättä päin seiniä, tai vielä pahempaa, seinien läpi. Tilakoneen tehtävänä on puolestaan kaikessa yksinkertaisuudessaan vaihtaa vihollispallon käyttäytymistä pakenemistilan ja jahtaamistilan välillä.

4.1.1 Reitinetsinnän toteuttaminen

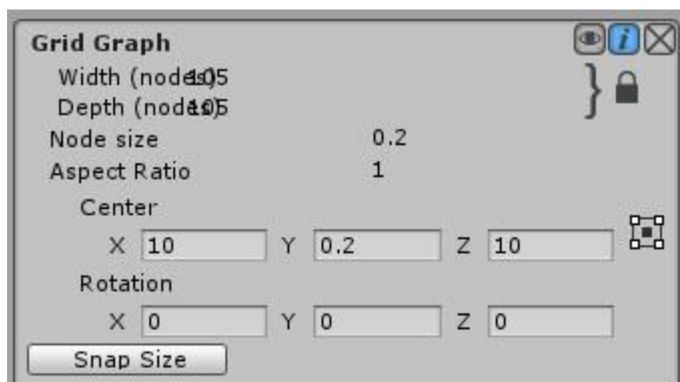
Ennen kuin reitinetsintää voi alkaa toteuttaa, pitää ensin päättää, minkälaista reitinetsintää halutaan käyttää. Kaikki reitinetsintäalgoritmit eivät tietenkään sovellu kaikkiin mahdollisiin reitinetsintäongelmiin, mutta toisaalta ongelman laajuudesta riippuen on turha lähteä rakentamaan mitään yltiömäisen monimutkaista algoritmia, etenkin jos tarkoituksena on toteuttaa se lähes tai täysin kokonaan itse. Tässä esimerkissä monimutkaisuudella ei kuitenkaan ole niin suurta merkitystä, koska en rakenna tekoälyalgoritmia itse.

Unityssa itsessäänkin on jonkinlainen navigaatioverkkoihin perustuva reitinetsintäominaisuus, mutta kyseinen ominaisuus on käytettävissä vain Unityn maksullisessa Pro-versiossa. Päädyinkin lopulta käyttämään reitinetsinnän toteutuksessa *A* Pathfinding Project*-laajennusta. Unityn Asset Storesta kyseisestä laajennuksesta löytyy vain maksullinen Pro-versio, mutta ilmaisen Free-version voi ladata projektin kotisivuilta (http://arongranberg.com/astar/). Ilmaisversio on samanlainen kuin maksullinenkin versio, mutta siinä on vähemmän ominaisuuksia ja sen kaupalliseen käyttöön liittyy rajoituksia.

Reitinetsinnän toteuttaminen alkaa siitä, että laajennus *importataan* eli lisätään projektiin (Assets->Import Package->Custom Package…) ja luodaan Unity-projektiin kaksi uutta *layeria*: yksi lattiaa varten ja yksi seiniä varten (Edit->Project Settings->Tags). Koska sokkelossa on vain yksi lattiataso eikä lainkaan korkeuseroja, oman layerin luominen lattialle ei ole välttämättä aivan pakollista, mutta se on kuitenkin hyvä lisätä varmuuden vuoksi. Layerien tarkoituksena on lähinnä kertoa tekoälylle, mitkä objektit ovat esteitä ja mitkä eivät.

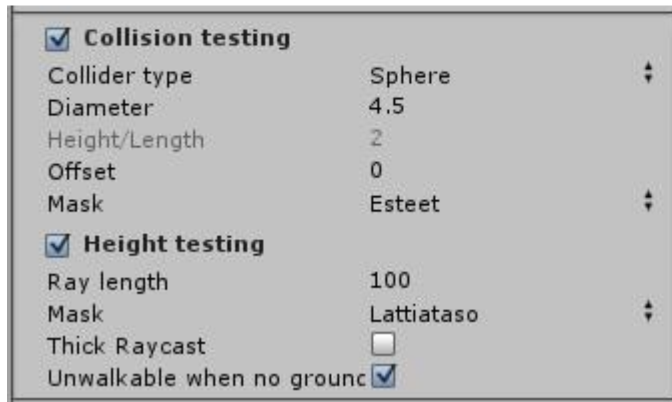
Seuraavaksi lisätään itse A*-toiminnallisuus projektiin. Tämä tapahtuu lisäämällä *sceneen* tyhjä peliobjekti, johon liitetään laajennuksen mukana tullut reitinetsintäkomponentti (Component->Pathfinding->Pathfinder). Tämän jälkeen pitää vielä luoda pisteverkko, josta tekoäly alkaa polkuja etsiä: tämä tapahtuu valitsemalla peliobjektin inspectorista ”Add New Graph” -vaihtoehto. Tämän jälkeen pitää vielä valita minkälainen pisteverkko halutaan: yksinkertainen ruudukko (Grid Graph) soveltuu tähän tehtävään mainiosti.

Kun uusi pisteverkko on luotu, inspectoriin aukeaa iso liuta säädeltäviä asetuksia, joista pisteverkkoa on mahdollista hienosäätää. Näistä oleellisimmat ovat kuitenkin ruudukon sijainti ja koko sekä *collision testing* eli törmäyksen testauksen asetukset. Koska tässä esimerkissä oleva sokkelo koostuu 21x21 kokoisesta ruudukosta, reitinetsintään riittää samankokoinen pisteverkko, jossa jokainen piste vastaa yhtä sokkelon ruutua. Jos tekoälylle haluaa kuitenkin antaa enemmän liikkumavaraa ja tehdä pallon liikkeestä mahdollisesti sulavampaa, pisteitä voi olla myös enemmän – itse päädyin käyttämään 105x105 kokoista pisteverkkoa. Pisteverkko on hyvä sijoittaa sokkelon päälle niin, että se on korkeussuunnassa hiukan lattiatason yläpuolella. Muuten tekoäly saattaa erheellisesti luoda virheellisen polun tai epäonnistua siinä kokonaan. Kuvassa 19 näkyvät esimerkissäni käyttämäni ruudukon koko- ja sijaintiasetukset.



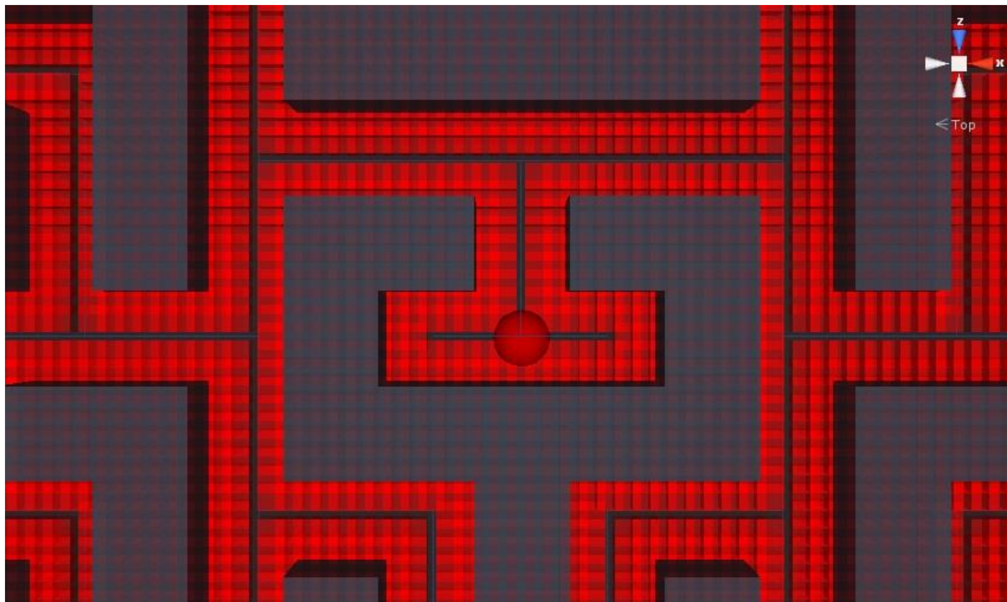
KUVA 19. Pisteverkon koko- ja sijaintiasetukset

Collision testing (kuva 20) on reitinetsinnän toiminnan kannalta hyvin tärkeä asetus. Sen pohjalta tekoäly nimittäin testaa, mitkä pisteet ovat kulkukelvottomia. Koska tekoälyn ohjaama objekti on pallo, *collider typeksi* voidaan tässä kohtaa asettaa *Sphere.* Diameter-asetuksessa pitää puolestaan määritellä testauksessa käytettävän pallon halkaisija, ja maskiksi asetetaan se layer, johon esteet kuuluvat. Jos lattiatason layerin haluaa huomioida polun laskennassa, sen voi lisätä Height testing-asetuksen maskiksi.



KUVA 20. Esteiden testaamisen asetukset

Kun kaikki tarvittavat asetukset on säädetty, voidaan polun luontia testata klikkaamalla inspectorin alareunassa olevaa *Scan*-painiketta. Tällöin valmiin pisteverkon pitäisi ilmestyä Unityn editoriin kuvan 21 mukaisesti. Tällöin ruudukossa pitäisi näkyä kulkukelvottomat pisteet punaisina palikkoina ja kulkukelpoiset reitit yhtenäisinä sinisinä viivoina. Jos ruudukossa ei näy minkäänlaista polkua tai polut menevät miten sattuu, vika on luultavasti törmäyksen testauksen asetuksissa tai väärin määritetyissä layereissa. Jos koko ruudukkoa ei taasen näy editorissa ollenkaan, vika saattaa olla Unityn omissa *gizmojen* näkyvyyteen vaikuttavissa asetuksissa.



KUVA 21. Valmis pisteverkko

Kun pisteverkko on valmis, pitää vihollisen ohjaamaan palloon vielä lisätä toiminnallisuus liikkua verkon eri pisteiden välillä. Tämän saavuttamiseksi palloon lisätään

ensimmäisenä *Seeker*-komponentti (Component->Pathfinder->Seeker). Tämän jälkeen objektiin tarvitaan vielä yksi erillinen skripti, jossa määritellään loput vihollispallon tarvitsemista tiedoista ja pyydetään *Seeker*-skriptiä etsimään reitti haluttuun määränpäähän. Ennen kuin tämä viimeinen skripti luodaan, voidaan vihollispalloon jo tässä vaiheessa lisätä *Character Controller*-komponentti, jota sen liikutteluun tarvitaan (Component->Physics->Character Controller).

```
public void Start () {

    targetPosition = new Vector3(pelaajago.transform.position.x,
    pelaajago.transform.position.y,pelaajago.transform.position.z);
    seeker = GetComponent<Seeker>();
    controller = GetComponent<CharacterController>();

    seeker.StartPath (transform.position,targetPosition, OnPathComplete);
}

public void OnPathComplete (Path p) {
    if (!p.error) {
        path = p;
        currentWaypoint = 0;
    }
}
```

KUVA 22. Luodaan ensimmäinen polku

Vaikka pelistä löytyykin jo valmis pisteverkko ja *Seeker*-komponentti joka osaa hakea erilaisia polkuyhdistelmiä kyseisestä pisteverkosta, vaaditaan vielä viimeinen skripti, jonka avulla tekoäly liitetään vihollisen ohjaamaan palloon. Tätä varten luodaan *AStarAI.cs*-skripti, jossa vihollista käsketään liikkumaan. Ennen kuin skriptiin alkaa lisätä mitään muuta koodia, aivan skriptin alkuun tulee lisätä rivi ”*using Pathfinding”*. Muuten pathfinding-laajennuksen mukana tulleet ominaisuudet eivät toimi koodissa. Tämän jälkeen voidaan suorittaa tarvittavat alustukset ja lopulta kutsua *Seeker*-skriptiä kuvan 22 mukaisesti. Kaikki mitä metodi *seeker.startPath* vaatii toimiakseen, on liikuteltavan objektin sijainti (transform.position) ja kohdepiste (targetPosition). Lopuksi kutsutaan *OnPathComplete*-funktiota, jossa varmistetaan, että polku on löydetty. Tässä koodinpätkässä polunlaskenta on *Start()*-funktion sisällä, joka tarkoittaa, että polku lasketaan silloin, kun peli alkaa. Uusi polku pitää kuitenkin muistaa säännöllisin väliajoin laskea uudelleen, koska kohdepiste – eli pelaaja – liikkuu jatkuvasti.

```
//Direction to the next waypoint
Vector3 dir = (path.vectorPath[currentWaypoint]-
transform.position).normalized;
dir *= speed * Time.fixedDeltaTime;
controller.SimpleMove (dir);
//Check if we are close enough to the next waypoint
//If we are, proceed to follow the next waypoint
if (Vector3.Distance (transform.position,
path.vectorPath[currentWaypoint]) < nextWaypointDistance) {
    currentWaypoint++;
    return;
}
```

KUVA 23. Vihollispallon liike

Nyt kun skripti osaa tallentaa haetun polun *path*-muuttujaan, tarvitaan enää koodinpätkä, joka saa pallon liikkumaan. Tähän tarvittava koodi näkyy kuvassa 23. Käytännössä pallo vain etsii suunnan, missä seuraava piste on, ja liikkuu sitä kohti ennalta määrätyllä nopeudella, kunnes se lähtee tavoittelemaan seuraavaa pistettä. Tässä kannattaa myös kiinnittää huomiota toisen *if*-lauseen sisällä olevaan *nextWaypointDistance*- muuttujaan. Kyseinen muuttuja alustetaan skriptin alussa ja se määrittää, kuinka monen pisteen yli objekti saa ”hypätä”. Jos tällaisia pisteitä on liikaa, pallo saattaa yrittää oikoa pisteestä toiseen seinien läpi, jolloin se jää jumiin. Tältä voi välttyä lisäämällä enemmän pisteitä pisteverkkoon tai pienentämällä *nextWaypointDistancen* arvoa.

4.1.2 Tilakoneen toteuttaminen

Tilakone saattaa äkkiseltään kuulostaa jotenkin äärimmäisen hienolta ja monimutkaiselta ohjelmointiviritelmältä, mutta etenkin tämän esimerkin puitteissa kyse on lähinnä siitä, että vihollispallo ohjelmoidaan käyttäytymään kahdella mahdollisella eri tavalla: pakenemaan pelaajalta tai jahtaamaan pelaajaa. Reitinetsinnän toteuttamisen kautta pelissä on valmiina jo oikeastaan yksi vaihe, nimittäin pelaajan jahtaaminen. Nyt tekoälylle pitää luoda ainoastaan toinen käyttäytymismalli, ja tehdä mahdolliseksi vaihdella näiden kahden eri tilan välillä. Tämän toteuttamiseksi luodaan kaksi uutta apuskriptiä: *muutos.cs* ja *poiminta.cs*.

Kuten peli-idean esittelyvaiheessa tuli jo ilmi, tarkoituksena on, että vihollinen lähtee pelaajalta karkuun silloin, kun pelaaja kerää edes yhden sokkelosta löytyvistä voima-

pisteistä. Pelimekaniikka on oikeastaan samanlainen kuin *Pac-Manissa*, jossa kaikki haamut muuttuvat hetkellisesti sinisiksi, jona aikana *Pac-Man* voi syödä ne. Poimintaskriptin tarkoituksena onkin lähinnä huolehtia siitä, että voimapisteet ovat poimittavissa, ja että poimimisen seurauksena vihollispallon tila muuttuu.

Poimintaskripti näkyy kokonaisuudessaan kuvassa 24. Skriptissä itsessään ei tapahdu oikeastaan mitään ihmeellistä: kun pelaaja törmää voimapisteeseen, voimapiste tuhoutuu ja samalla kutsutaan vihollispalloon liittyviä funktioita *muuntaudu* ja *menePakoon.* Niiden kutsumiseen täytyy käyttää Unityn *SendMessage*-metodia, koska ne sijaitsevat eri peliobjektissa kuin poimintaskripti.

```
public class Poiminta : MonoBehaviour {

    public GameObject vihollinen;

    void OnTriggerEnter(Collider hitti) {
        if(hitti.gameObject.tag == "kerailtava"){

            Destroy(hitti.gameObject);
            vihollinen.SendMessage("Muuntaudu");
            vihollinen.SendMessage("menePakoon");
        }
    }
}
```

KUVA 24. Poimintaskripti

Muutosskriptissä tapahtuu puolestaan jo aavistuksen verran enemmän kuin poimintaskriptissä. Nimensä mukaisesti se sisältää varsinaiset muutokset tilojen välillä. Tämän lisäksi skripti muuttaa vihollispallon väriä riippuen siitä, missä tilassa se on: kuten *Pac-Manissa*, pallo muuttuu siniseksi silloin, kun se pakenee pelaajaa. Kun se taas jonkin ajan kuluttua jatkaa pelaajan jahtaamista, se muuttuu taas punaiseksi. Muuttuja *onkoMuutos* on numeroarvo, joka nousee, kun pelaaja poimii voimapisteen, ja laskee itsestään aina, kun aikaa kuluu. Muuttuja *muutosKytkin* on lähinnä totuusarvomuuttuja, joka varmistaa että tila joko on tai ei ole päällä. Käytännössä sitä ei oikeastaan tarvitsisi, koska saman voi ilmoittaa *onkoMuutos*-muuttujan avulla. Muutosskripti näkyy kuvassa 25.

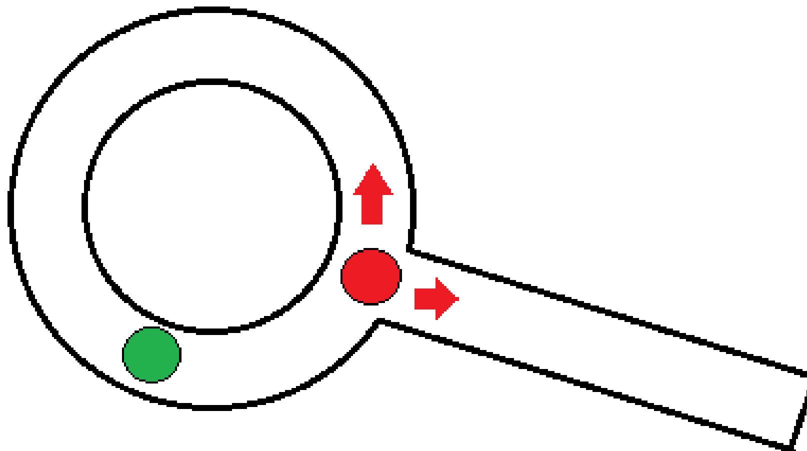
```
void Update () {

    if(onkoMuutos > 0)
    {
        renderer.material.color = Color.cyan;
        onkoMuutos--;
    }
    else
    {
        renderer.material.color = Color.red;
        if (muutosKytkin = true){
            muutosKytkin = false;
            gameObject.SendMessage("jatkaJahtausta");
        }}}

void Muuntaudu() {
    onkoMuutos = 500;
    muutosKytkin = true;
}
```

KUVA 25. Muutosskripti

Nyt vihollispallo muuttaa jo väriä tilastaan riippuen, mutta toistaiseksi tilamuutos ei vaikuta millään tavoin sen liikkumiseen. Tämän korjaamiseksi täytyy muokata reitinetsintäskriptiä niin, että myös se ottaa huomioon tilojen muutokset. Juuri tämän takia poimintaskripti kutsuukin kahta funktiota yhden sijaan.



KUVA 26. Kumpi reitti on parempi?

Tässä vaiheessa törmätään kuitenkin visaiseen ongelmaan. Vaikka reitti johonkin ennalta määrättyyn pisteeseen onkin suhteellisen yksinkertaista, sellaisen välttely ei ole helppoa, etenkään jos välteltävä objekti liikkuu koko ajan. A*-algoritmin toiminta on

ainakin teoriassa mahdollista muuttaa käänteiseksi niin, ettei se pääse ikinä perille, mutta vaikka tämä käyttäytyminen vastaa käytännössä kohdepisteen välttelyä, se toisi mukanaan liudan erilaisia ongelmia. Yksi suurimmista ongelmista on, että reitinetsintäalgoritmit eivät useimmiten osaa tehdä hyviä ratkaisuja ”pitkällä tähtäimellä”. Yksi tällainen ongelmatilanne on kuvassa 26, jossa vihollispallo saattaisi yrittää paeta pelaajaa umpikujaan. Tässä sokkeloesimerkissä tämä ongelma ei ole kovinkaan vakava, koska sokkelossa on hyvin vähän umpikujia.

Esimerkin kannalta ongelmallista on kuitenkin, että vaikka *A* Pathfinding Project* sisältääkin erillisen funktion nimenomaan pakenemista varten, se on saatavilla vain Pro-versiossa. Lähdekoodin muokkaaminen itse olisi puolestaan hyvin haastavaa. Yksi vaihtoehto olisi myös tehdä oma polunetsintäalgoritmi pakenemista varten, mutta se olisi puolestaan melko turhauttavaa. Päätinkin toteuttaa pakenemisen kiertoratkaisulla, jossa pakeneva vihollinen valitsee määränpääkseen yhden sokkelon kulmista sen mukaan, mistä pelaaja on kaikista kauimpana. Tämä ratkaisu toimii jotenkuten, joskin siinä on kaksi heikkoutta: ensinnäkin vihollinen saattaa yhtäkkiä yllättäen vaihtaa mieltään ja alkaa liikkua aivan eri suuntaan, ja toisekseen se ei yhtään osaa tehdä reitinvalintaa pelaajan sijainnin mukaan ja saattaa vahingossa paeta suoraan pelaajaa päin yrittäessään päästä määränpäähänsä.

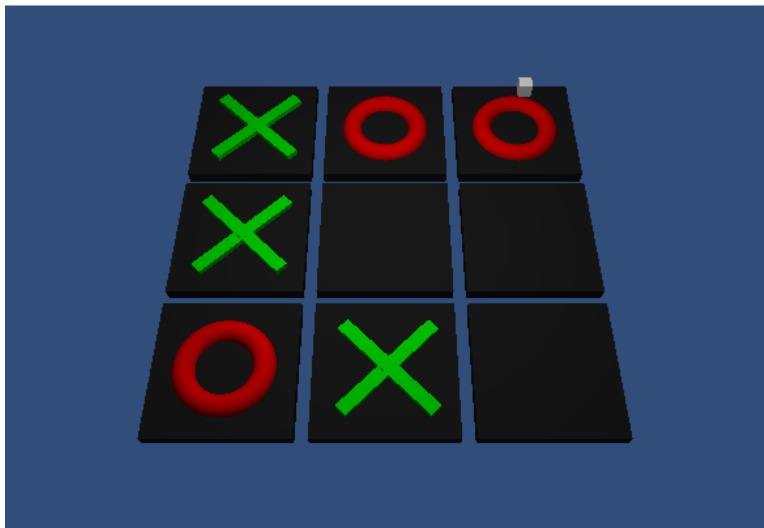
```
while (tekoTila == 1)
{
    int randomPakopaikka = Random.Range (0,3);
    switch (randomPakopaikka)
    {
        case 1:
            targetPosition = new Vector3(19f,0.7f,19f);
            break;
        case 2:
            targetPosition = new Vector3(1f,0.7f,19f);
            break;
        case 3:
            targetPosition = new Vector3(19f,0.7f,1f);
            break;
        default:
            targetPosition = new Vector3(1f,0.7f,1f);
            break;
    }
    etaisyys = Vector3.Distance (targetPosition, transform.position);
    if (etaisyys > 1)
    {
        etaisyys = Vector3.Distance (targetPosition, pelaajago.transform.position);
        if (etaisyys > 5)
            {
            tekoTila = 2;
            seeker.StartPath (transform.position,targetPosition, OnPathComplete);
            }
    }
}
```

KUVA 27. Haetaan pakoreitti

Käytännössä pakeneminen perustuu siis siihen, että polunetsintä arpoo yhden sokkelon kulmapisteistä sattumalta ja alkaa etsiä reittiä sinne, mikäli pelaaja on tarpeeksi kaukana kyseisestä pisteestä. Jos pelaaja on liian lähellä kyseistä pistettä, tai vihollinen on jo melkein perillä kyseisessä pisteessä, arvotaan uusi piste. Kuten tavallisessa jahtaustapauksessakin, uusi reitti etsitään säännöllisin väliajoin uudelleen. Pakoreitin hakemiseen tarvittava koodi näkyy kuvassa 27. Ratkaisu ei ole erityisen elegantti, mutta täyttää tehtävänsä osana kokonaisuutta.

4.2 Ristinolla

Toisen esimerkkipelini aiheeksi valitsin ristinollan. Aihevalinta perustui siihen, että vaikka ristinolla onkin sekä yksinkertainen pelata että verrattain yksinkertainen toteuttaa, siitä käy kuitenkin melko hyvin ilmi tekoälyn toiminnallisuus. Toisin kuin monissa muissa lautapeleissä, siirrot on melko helppo pisteyttää eikä siinä ole mitään poikkeustilanteita, jotka pitäisi huomioida.



KUVA 28. Ristinollan pelitilanne

Pelin idea on siis varsin yksinkertainen: peliruudukko koostuu yhdeksästä tyhjästä ruudusta, johon pelaajat vuorotellen merkkaavat oman merkkinsä eli joko ristin (X) tai nollan (O). Se pelaaja, joka ensimmäisenä saa kolme omaa merkkiään vierekkäin, voittaa. Jos kaikki ruudut täyttyvät, eikä kumpikaan pelaaja saa kolmen suoraa, peli

päättyy tasapeliin. Kuvassa 28 on ruudunkaappaus yhdestä mahdollisesta pelitilanteesta.

Kuten sokkeloesimerkkikin, peli on kolmiulotteinen ja koostuu lähinnä yksinkertaisista Unitylla luoduilla muodoista. Poikkeuksena on nollan pelimerkki, jota ei ole suoraan saatavilla Unityssa ja josta ei edes Unityn Asset Storesta löytynyt ilmaisia versioita. Päädyin nopeasti tekemään itselleni siitä 3D-mallin *SketchUpissa*, jonka ilmainen Pro-version kokeiluaika sallii 3D-mallien tallentamisen myös sellaisissa muodoissa, jotka Unity hyväksyy. Käytännössä saman olisi voinut tehdä millä tahansa ilmaisella mallinnusohjelmalla, kuten vaikkapa *Blenderillä.*

Aloitin pelilaudan rakentamisen luomalla 9 erillistä ruutuobjektia, jotka kaikki ovat omia objektejaan eivätkä vain yhden peliobjektin eri instansseja. Loin myös uuden *tagin* (Edit->Project Settings->Tags) nimeltä *ristiRuutu* ja lisäsin sen jokaiseen ruutuun erikseen. Tämän tarkoituksena on lähinnä ilmoittaa, että ruudut ovat tyhjiä peliruutuja. Jo tässä vaiheessa on hyvä huomata, että jos kyseessä olisi yhtään laajempi peli, ei jokaisen ruutua kannattaisi pitää välttämättä omana peliobjektinaan, sillä jo yhdeksän objektin hallinta erikseen voi käydä työlääksi. Tämän esimerkin puitteissa se kuitenkin helpottaa asioita jonkin verran.

Peliin jokaiseen ruutuun on myös liitetty skriptitiedosto *ruutuTiedot.cs.* Tämän skriptin tarkoitus on lähinnä ilmoittaa pelille mikä ruudun *id* on. Jokaisella peliruudulla on oma *id* ja peli käyttää tätä tunnusta tunnistamaan mikä ruutu on kyseessä, missä se sijaitsee ja onko se tyhjä vai pelattu ruutu. Skripti on nähtävillä kokonaisuudessaan kuvassa 29. Koodi käyttää apuna *id:n* selvittämisessä peliobjektin nimeä, sillä ruudun nimessä oleva numero on jokaisen ruudun kohdalla aina yhden suurempi, kuin mitä sen *id* on. Esimerkiksi ruutu_1:n id on 0, ruutu_2:n id on 1, jne. Kun ruudun id on saatu selville, skripti lähettää sen eteenpäin *varaaRuutu()*-funktiolle.

```
public class ruutuTiedot : MonoBehaviour {

    private string ruutuNimi;
    private string[] splitteri;
    private int ruutuNro;

    void Start () {
        ruutuNimi = gameObject.name;
        splitteri = ruutuNimi.Split('_');
        ruutuNro = int.Parse(splitteri[1]);
    }

    void tarkistaRuutu(){
        int ruutuIndeksi = ruutuNro - 1;
        GameObject.Find("Pelinydin").SendMessage("varaaRuutu", ruutuIndeksi);
    }
}
```

KUVA 29. Ruudun tiedot

Peliruudun valinta toimii pelissä niinkin yksinkertaisesti, että pelaajan pitää vain klikata peliruudulla tyhjää ruutua, jolloin siihen tulee pelaajan pelimerkki. Ensin koodi tarkistaa, onko valittu ruutu tyhjä käyttäen *tarkistaRuutu*()-funktiota. Tämä funktio käyttää apunaan edellä läpikäytyä *ruutuTiedot.cs* skriptiä. Jos ruutu on tyhjä, se merkitään varatuksi, ja siihen piirretään pelaajan merkki. Klikkauksen tunnistamiseen käytetään apuna *Raycastingia.* Käytännössä se tarkoittaa, että hiiren kursorista ”ammutaan” näkymätön säde ja katsotaan, törmääkö se mihinkään – tässä tapauksessa peliruutuun. Klikkauksen tunnistamiseen liittyvä koodi on nähtävissä kuvassa 30.
Tekoälyn siirrot toimivat samalla toimintaperiaatteella kuin ihmispelaajankin siirrot, mutta hiiren kursorin sijaan tekoälyllä on oma ”valintakuutio” joka vaihtaa paikkaa sen mukaan, minkä siirron tekoäly aikoo tehdä.

```
else if (turn==-1){ // pelaaja vuoro (X)
if (Input.GetButtonDown("Fire1")) {

    //Etsitään Raycastilla valittu ruutu ja valitaan se, jos validi
    RaycastHit hitInfo = new RaycastHit();
    bool hit = Physics.Raycast(Camera.main.ScreenPointToRay(Input.mousePosition),
    out hitInfo);
    if (hit)
    {
        if (hitInfo.transform.gameObject.tag == "ristiRuutu")
        {
            //Debug.Log ("Homma toimii");
            if(turn == -1)
            {
                Vector3 paikka = new Vector3(-5.5F,7.5F,-4.5F);
                paikka = paikka+hitInfo.transform.position;
                Instantiate(rnRisti, paikka, transform.rotation);
                hitInfo.transform.gameObject.tag = "ristiX";
                hitInfo.transform.gameObject.SendMessage("tarkistaRuutu");
            }
        }
    }
```

KUVA 30. Klikkauksen tunnistaminen

Tekoälyn toimintaperiaate tässä esimerkissä perustuu kolmen eri funktion toimintaan. Näitä funktioita ovat *TakeWin(),TakeBlock()* ja *TakeRandom().* Idea näiden takana on varsin yksinkertainen ja vastaa osittain sitä ajatusmallia, jolla ihmispelaajakin pelaisi ristinollaa. Käytännössä tekoäly ensin katsoo, voiko se tehdä siirron, jolla se voittaa pelin. Jos sellaista siirtoa ei ole, se katsoo, voiko se estää pelaajaa tekemästä peliä ratkaisevaa siirtoa. Jos sellaistakaan siirtoa ei ole, niin tekoäly valitsee jonkin sattumanvaraisen ruudun pelilaudalta.

```
bool TakeWin(){
    int winningIndex;
    int pojot = 0;
    bool bIsWinningMove;
    bIsWinningMove = false;
    winningIndex = 0;
    for (int i = 0; i != (scoreSums.Length); i++){
        pojot = scoreSums[i];
        if (pojot == 2) // This is a win!
        {
            bIsWinningMove = true;
            break;
        }
        winningIndex = winningIndex+1;
    }
    if (!bIsWinningMove) return false;
    int[,] potentialPositions = new int[8,3] { {0,1,2}, {3,4,5}, {6,7,8},
    {0,3,6}, {1,4,7}, {2,5,8}, {0,4,8}, {6,4,2} };
    int spot = 0;
    for (int o = 0; o < 3; o++){
        spot = potentialPositions[winningIndex,o];
        if (boardState[spot] == 0) {
            //boardState[spot] = turn;
            tekoalySiirto(spot);}}
    return true;}
```

KUVA 31. Tekoälyn voittosiirto

Tarkastellaanpa sitten lähemmin *TakeWin()*-funktiota, jolla tekoäly etsii ja valitsee pelin ratkaisevan siirron (kuva 31). Muuttuja *bIsWinningMove* on totuusarvomuuttuja, joka kertoo tekoälylle, onko tutkitulla siirrolla mahdollista voittaa, vai ei. Kyseisen muuttujan arvo perustuu siihen, löytyykö *scoreSums*-taulukosta yhtään pisteriviä, jossa tekoälyllä olisi 2 pistettä. Kaksi pistettä vastaa siis kahta vierekkäin olevaa pelimerkkiä, kolmas piste olisi täten voitto. Jos nykytilanteessa ei ole yhtään 2 pisteen riviä, *bIsWinningMove* palauttaa epätoden ja yrittää seuraavaa funktiota. *TakeBlock()* toimii samalla toimintaperiaatteella kuin *TakeWin(),* mutta sen sijaan, että tekoäly vertailisi mahdollisia voittorivejä omiin pisteisiinsä, se vertailisi niitä pelaajan pisteisiin.

```
void TakeRandom() {
    int boardIndex = 0;
    List<int> freeSpotArray = new List<int>();
    int boardStateVal = 0;

    for (int i = 0; i != (boardState.Length); i++){
        boardStateVal = boardState[i];
        if (boardStateVal == 0) {
            freeSpotArray.Add(boardIndex);
        }
        boardIndex = boardIndex + 1;
    }

    // Valitse vapaista ruuduista yksi satunnainen
    int newIndex = Random.Range(0, freeSpotArray.Count);
    //boardState[newIndex] = turn;
    tekoalySiirto(newIndex);
}
```

KUVA 32. Satunnaisen ruudun valinta

Satunnaisen vapaan ruudun valinta eroaa jonkin verran voittavan tai torjuvan siirron valinnasta. Sen toimintaperiaate on kuitenkin varsin yksinkertainen. Käytännössä funktio *TakeRandom()* vain hakee kaikki tyhjät ruudut, pinoaa niiden tunnisteet eli id:t yhteen listaan ja lopuksi valitsee sattumalta niistä yhden. Satunnaisen luvun valintaan on käytetty Unityn omaa *Random.Range*-metodia. Kuten kuvasta 32 näkyy, kyseinen metodi on hyvin helppokäyttöinen – sille pitää vain parametreina syöttää pienin ja suurin arvottavissa oleva luku, jonka jälkeen se osaa satunnaisesti luoda jonkun kokonaisluvun näiden arvojen väliltä.

5 PÄÄTÄNTÖ

Opinnäytetyöni päätavoitteena oli toteuttaa erilaisia tekoälyä hyödyntäviä esimerkkipelejä, jotka käyttävät Unity-pelimoottoria. Vaikka tekoälyn toteuttaminen ja siihen liittyvät algoritmit eivät mitään rakettitiedettä olekaan, sai asiaan paremmin perehtyessään todeta niiden taustalla olevan parhaimmillaan melko monimutkaista matematiikkaa. Etenkin reitinetsinnässä teoriaa olisi riittänyt materiaalia useammankin opinnäytetyön tarpeiksi. Omat esimerkkinikin vain raapaisevat tekoälyn pintaa aihealueena, mutta se kielii toisaalta myös aihealueen laajuudesta.

Kuinka paljon Unitylla työskentely sitten loppujen lopuksi helpottikaan tekoälyn toteuttamista? Loppujen lopuksi voisi sanoa, että ei juuri ollenkaan. Ainakaan mikäli

sellaisen toteuttamiseen ei tee rahallisia investointeja. Unity ei itsessään tarjoa tekoälyn toteuttamista varten juuri minkäänlaisia työkaluja ja Unityyn saatavilla olevat ilmaiset laajennukset ovat nekin melko rajoitteellisia ja ne sisältävät myös joitakin pikkuvikoja. Toisaalta Unitylla työskentely helpottaa pelientekoa kaikilta muilta osin, jonka ansiosta tekoälyn toteuttamiseen keskittyminen on helpompaa.

Omia esimerkkejäni ajatellen tulee jo mieleen joitakin vaihtoehtoisia toteutusideoita, joita olisin voinut vaihtoehtoisesti kokeilla. Sokkeloesimerkin toteuttamisessa olisi voinut kokeilla yksinkertaisen reitinetsinnän toteuttamista itsenäisesti sen yksinkertaisuuden vuoksi, ja toisaalta myös koska A* algoritmi oli kyseiseen esimerkkiin ehkä vähän liiankin yliampuva. Toisaalta vihollisen pakenemiskäyttäytymistäkin olisi voinut parantaa niin, ettei se vahingossakaan yrittäisi paeta suoraan pelaajaa kohti. Lautapeliesimerkissä olisi voinut puolestaan pyrkiä kehittelemään paljon etevämpää tekoälyä, joka osaisi pelata ristinollaa erikokoisilla pelilaudoilla perinteisen 3x3 laudan lisäksi.

Reitinetsinnän lisäksi sokkeloesimerkkiä olisi voinut parantaa myös tilakoneen osalta. Peliin olisi voinut lisätä enemmän vihollisia ja jokaiselle olisi voinut antaa omanlaisensa, hiukan muista eroavan tilan. Näin vihollispalloille olisi *Pac-Mania* mukaillen saanut eroavat persoonallisuudet. Tällä tavoin esimerkin olisi saanut myös vaikuttamaan enemmän valmiilta peliltä.

Nämä esimerkit ovat kuitenkin vain melko pinnallisia, eivätkä anna kovin hyvää käsitystä siihen mihin nykytekoäly oikeasti pystyy. Toteuttamani tekoälyt ovat myös huomattavasti alkeellisempia ja suoraan sanottuna tyhmempiä, kuin mitä nykypelien tekoälyltä voisi parhaimmillaan odottaa. Niistä saa kuitenkin hyvän käsityksen siitä, millä toimintaperiaatteella pelitekoäly toimii ja mistä niiden toteuttamisessa on hyvä lähteä liikkeelle.

Ei olekaan mitenkään vaikeaa uskoa, että asiansa osaavilla tekoälyohjelmoijilla on pelialalla kysyntää. Valmiiden tekoälylaajennusten ostaminen ja vanhojen tekoälyohjelmien kierrättäminen tuntuukin olevan nykypelientuotannossa arkipäivää. Tämän lisäksi etenkin isot pelistudiot tuntuvat pääosin panostavan esimerkiksi pelien näyttävyyteen enemmän, jolloin tekoälyn kehittäminen ja prioriteeteissa taka-alalle. Omien

kokemusteni perusteella pelialalla ollaankin jokseenkin jälkeenjääneitä tekoälyn saralla: sen sijaan, että pelistudiot kilpailisivat parhaasta tekoälystä, ne kilpailevat siitä, missä tekoälyssä on kaikista vähiten moitittavaa.

Uskon kuitenkin, että tekoälyn kehitys lähtee tulevaisuudessa uuteen nousuun. Tietokoneiden ja pelikonsolien tullessa yhä tehokkaammiksi ja peligrafiikoiden tullessa yhä vaikeammiksi parantaa, tekoäly saattaa olla hyvinkin se tekijä, jolla pelit tulevaisuudessa tulevat erilaistumaan toisistaan. Uudet ja helppokäyttöisemmät kehitystyökalut antanevat kehittäjille myös paremmat edellytykset hyödyntää toistaiseksi melko harvinaisiakin tekoälymenetelmiä, kuten neuroverkkoja. Tulevaisuudessa tullaan varmasti kehittelemään myös lisää tehokkaampia ja älykkäämmältä vaikuttavia tekoälyalgoritmeja.

Aika kuitenkin näyttää, milloin muutokset näkyvät itse tietokonepeleissä. Nykyiset olemassa olevat tekoälymenetelmät nimittäin täyttävät useimmiten tarkoituksensa riittävän hyvin. En kuitenkaan pitäisi minkäänlaisena mahdottomuutena sitäkään mahdollisuutta, että tulevaisuuden pelitekoälyt saattavat hyvinkin luoda jopa täysin uusia peligenrejä.

LÄHTEET

Adamchik, Victor S. 2009. Game Trees. WWW-dokumentti. http://www.cs.cmu.edu/~adamchik/15-121/lectures/Game%20Trees/Game%20Trees.html. Ei päivitystietoja. Luettu 13.5.2013.

Bailiie de-Byl, Penny 2004. Programming Believable Characters for Computer Games. Hingham: Charles River Media / Cengage Learning.

Birch, Chad 2010. Understanding Pac-Man Ghost Behavior. WWW-dokumentti. http://gameinternals.com/post/2072558330/understanding-pac-man-ghost-behavior. Päivitetty 2.12.2010. Luettu 6.5.2013.

Bond, Allison 2010. Reality Checkup: Medical Artificial Intelligence Still a Hard Sell in the Clinic. Scientific American. WWW-dokumentti. http://www.scientificamerican.com/article.cfm?id=artificial-intelligence-medical-tests-software-diagnosis. Päivitetty 12.1.2010. Luettu 22.4.2013.

Booth, Michael 2009. The AI systems of Left 4 Dead. PDF-dokumentti. http://www.valvesoftware.com/publications/2009/ai_systems_of_l4d_mike_booth.pdf. Ei päivitystietoja. Luettu 7.5.2013.

Bourg, David M. & Seemann, Glenn 2004. AI for Game Developers. Sebastopol: O'Reilly Media.

Brookshear, J. Glenn 2003. Tietotekniikka. Helsinki: Edita Prima Oy.

Champandard, Alex J. 2007. Evolving with Creatures' AI: 15 Tricks to Mutate into Your Own Game. WWW-dokumentti. http://aigamedev.com/open/review/creatures-ai/. Päivitetty 1.10.2007. Luettu 7.5.2013.

Copeland, Jack 2000a. Strong AI, Applied AI, and CS. WWW-dokumentti. http://www.alanturing.net/turing_archive/pages/reference%20articles/what_is_AI/What%20is%20AI02.html. Päivitetty 6.12.2011. Luettu 18.3.2013.

Copeland, Jack 2000b. Is Strong AI Possible? WWW-dokumentti. http://www.alanturing.net/turing_archive/pages/reference%20articles/what_is_AI/What%20is%20AI13.html. Päivitetty 6.12.2011. Luettu 18.3.2013.

Copeland, Jack 2000c. Early AI Programs. WWW-dokumentti. http://www.alanturing.net/turing_archive/pages/reference%20articles/what_is_AI/What%20is%20AI04.html. Päivitetty 6.12.2011. Luettu 18.3.2013.

Dominguez, James 2012. What's so special about Half-Life anyway?. WWW-dokumentti. http://www.smh.com.au/digital-life/games/blogs/screenplay/whats-so-special-about-halflife-anyway-20121002-26vyw.html. Päivitetty 2.10.2012. Luettu 7.5.2013.

Genesereth, Michael 2013. General Game Playing. WWW-dokumentti. http://logic.stanford.edu/classes/cs227/2013/logistics/information.html. Ei päivitystietoja. Luettu 14.5.2013.

Grenier, Michael 2011. Pathfinding concept, the basics. WWW-dokumentti. http://mgrenier.me/2011/06/pathfinding-concept-the-basics/. Päivitetty 30.6.2011. Luettu 19.8.2013.

Griliopoulos, Dan 2013. GDC 2013: The AI tricks behind XCOM, Assassins Creed 3 and Warframe. WWW-dokumentti. http://www.pcgamer.com/2013/03/26/gdc-2013-the-ai-tricks-behind-xcom-assassins-creed-3-and-warframe/. Päivitetty 26.3.2013. Luettu 5.4.2013.

Gunkel, David J. 2012. Machine Question - Introduction. PDF-dokumentti. http://machinequestion.org/pdf/gunkel_MQ_intro.pdf. Päivitetty 13.6.2012. Luettu 19.3.2013.

Hayes, James 2012. Turing Test 2012: friend or faux. WWW-dokumentti. http://eandt.theiet.org/magazine/2012/07/friend-or-faux.cfm. Päivitetty 17.7.2012. Luettu 10.4.2013.

History of Automata 2009. Gearworx. WWW-dokumentti. http://www.handworx.com.au/gearworx/history/ancient.html. Päivitetty 24.7.2009. Luettu 5.3.2013.

History of Computers and Computing 2013. Norbert Wiener. WWW-dokumentti. http://history-computer.com/ModernComputer/thinkers/Wiener.html. Päivitetty 13.2.2013. Luettu 4.3.2013.

Hjørland, Birger 2005. Knowledge representation. WWW-dokumentti. http://www.iva.dk/jni/lifeboat/info.asp?subjectid=92. Päivitetty 13.1.2008. Luettu 15.7.2013.

Honda 2011. Honda Unveils All-new ASIMO with Significant Advancements. WWW-dokumentti. http://world.honda.com/news/2011/c111108All-new-ASIMO/. Päivitetty 22.11.2011. Luettu 24.4.2013.

IBM 2003. Deep Blue. WWW-dokumentti. http://www-03.ibm.com/ibm/history/ibm100/us/en/icons/deepblue/. Ei päivitystietoja. Luettu 11.3.2013.

Imperial College London 2006. Simple Reflex Agents. WWW-dokumentti. http://www.doc.ic.ac.uk/project/examples/2005/163/g0516334/sra.html. Päivitetty 2.6.2006. Luettu 23.4.2013.

Johnston, John 2008. Allure of Machinic Life : Cybernetics, Artificial Intelligence and the New Al. Cambridge: MIT Press.

Jones, M. Tim 2003. AI Application Programming. Herndon: Charles River Media / Cengage Learning.

Kumar, Ela 2008. Artificial Intelligence. New Delhi: I.K. International Publishing House Pvt. Ltd.

LaValle, Steven M. 2006. A-star. Planning Algorithms. WWW-dokumentti. http://planning.cs.uiuc.edu/node45.html. Päivitetty 20.4.2012. Luettu 21.5.2013.

Lin, Yosen 2003. Computer Science Game Trees. WWW-dokumentti. http://www.ocf.berkeley.edu/~yosenl/extras/alphabeta/alphabeta.html. Päivitetty 20.12.2005. Luettu 20.5.2013.

Love, Dylan 2011. The 10 Most Important Computers From The Last 30 Years. WWW-dokumentti. http://www.businessinsider.com/most-important-computers-2011-8?op=1. Päivitetty 12.8.2011. Luettu 17.4.2013.

Machines like us 2007. The end of AI winter? WWW-dokumentti. http://machineslikeus.com/news/end-ai-winter. Päivitetty 26.10.2007. Luettu 12.3.2013.

Massaad, Roy 2011. Dune 2. WWW-dokumentti. http://www.gamebuzz.me/gamebuzz.php/2011/05/13/dune-2. Päivitetty 13.5.2011. Luettu 7.5.2013.

McCarthy, John 2007. What is Artificial Intelligence? Basic Questions. WWW-dokumentti. http://www-formal.stanford.edu/jmc/whatisai/node1.html Päivitetty 12.11.2012. Luettu 11.3.2013.

McShaffry, Mike 2012. Game Coding Complete (4th Edition). Boston: Course Technology / Cengage Learning.

Murphy, Tom 2013. The First Level of Super Mario Bros. is Easy with Lexicographic Orderings and Time Travel... after that it gets a little tricky. PDF-dokumentti. http://www.cs.cmu.edu/~tom7/mario/mario.pdf. Päivitetty 1.4.2013. Luettu 20.5.2013.

Niemueller, Tim & Widyadharma, Sumedha 2003. Artificial Intelligence – An Introduction to Robotics. PDF-dokumentti. http://www.niemueller.de/uni/roboticsintro/AI-Robotics.pdf. Päivitetty 8.7.2003. Luettu 16.4.2013.

Old-computers 2013. Digital History: Space Invaders. WWW-dokumentti. http://www.old-computers.com/history/detail.asp?n=1&t=4. Ei päivitystietoja. Luettu 6.5.2013.

Parker, Laura & King, Darryn 2008. Why Pong scored so highly for Atari. WWW-dokumentti. http://www.guardian.co.uk/technology/2008/apr/17/games.atari. Päivitetty 17.4.2008. Luettu 7.5.2013.

Perkowitz, Sidney 2004. Digital People: From Bionic Humans to Androids. Washington: Joseph Henry Press.

Pfeifer, Rolf & Scheier, Christian 1999. Understanding Intelligence. Cambridge: MIT Press.

Pfeifer, Rolf, Bongard, Josh & C. Brooks, Rodney 2006. How the Body Shapes the Way We Think. Cambridge: MIT Press.

Pirovano, Michele 2012. The use of Fuzzy Logic for Artificial Intelligence in Games. PDF-dokumentti. http://homes.di.unimi.it/~pirovano/pdf/fuzzy_ai_in_games.pdf. Päivitetty 7.12.2012. Luettu 20.5.2013.

Palace, Bill 1996. Data Mining: What is Data Mining? WWW-dokumentti. http:// http://www.anderson.ucla.edu/faculty/jason.frand/teacher/technologies/palace/datamining.htm. Päivitetty 4.11.1997. Luettu 17.8.2013.

Rabin, Steve 2009. Introduction to Game Development (2nd Edition). Herndon: Course Technology / Cengage Learning.

Randar 2012. The Bot FAQ. WWW-dokumentti. http://www.randars.com/bots/botfaq.html. Päivitetty 27.10.2012. Luettu 22.5.2013.

Retro Gamer 2013. Space Invaders:The Original Game. WWW-dokumentti. http://www.retrogamer.net/show_image.php?imageID=2852. Ei päivitystietoja. Luettu 7.5.2013.

Retro.mmgn.com 2012. Pac Man Arcade screen shot. WWW-dokumentti. http://retro.mmgn.com/Arcade/Gallery/Pac-Man-Arcade-screen-shot. Päivitetty 31.7.2012. Luettu 7.5.2013.

Russell, Stuart & Norvig, Peter 2003. Artificial Intelligence - A Modern Approach, Second Edition. New Jersey: Pearson Education, Inc.

Schwab, Brian 2008. AI Game Engine Programming (2nd Edition). Boston: Course Technology / Cengage Learning.

Shi, Zhongzhi 2011. Advanced Artificial Intelligence. River Edge: WSPC.

Sreedhar, Suhas 2007. Checkers, Solved! WWW-dokumentti. http://spectrum.ieee.org/computing/software/checkers-solved. Päivitetty 1.7.2007. Luettu 17.4.2013.

Stahl, Ted 2013. Chronology of the History of Video Games. WWW-dokumentti. http://www.thocp.net/software/games/golden_age.htm. Päivitetty 15.3.2013. Luettu 6.5.2013.

Steam 2013. Half-Life. WWW-dokumentti. http://store.steampowered.com/app/70/?snr=1_7_15__13. Ei päivitystietoja. Luettu 8.5.2013.

Thrun, Sebastian 2005. AI - The Next 25 Years. WWW-dokumentti. http://www.ai.rutgers.edu/aaai25/. Päivitetty 28.9.2005. Luettu 9.4.2013.

Thursten, Chris 2013. Reinstall: Black and White. WWW-dokumentti. http://www.pcgamer.com/2013/03/24/reinstall-black-and-white/2/. Päivitetty 24.3.2013. Luettu 8.5.2013.

Wexler, James 2002. Artificial Intelligence in Games: A look at the smarts behind Lionhead Studio’s “Black and White” and where it can and will go in the future. PDF-dokumentti. http://www.cs.rochester.edu/~brown/242/assts/termprojs/games.pdf. Päivitetty 11.5.2002. Luettu 7.5.2013.

Vision Recognition 2003. Vision & AI. WWW-dokumentti. http://www.mbhs.edu/~lpiper/computer_vision03/visionai.html. Päivitetty 12.6.2003. Luettu 16.4.2013.