Osaamista
ja oivallusta
tulevaisuuden
tekemiseen

Erika Tarsala

Tekoälyn toteutus 2D-tasohyppelypelissä

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tietotekniikka

Insinöörityö

5.12.2021

Tekijä Otsikko Sivumäärä Aika	Erika Tarsala Tekoälyn toteutus 2D-tasohyppelypelissä 54 sivua 5.12.2021
Tutkinto	insinööri (AMK)
Tutkinto-ohjelma	Tietotekniikka
Ammatillinen pääaine	Ohjelmistotekniikka
Ohjaaja	Lehtori Antti Laiho

Tekoäly on laaja ja jatkuvasti kehittyvä ala, jonka soveltaminen videopeleissä eroaa merkittävästi sen muista käyttökohteista. Tämän insinöörityön tavoitteena on tutustua tekoälyyn käsitteenä ja sen eri osa-alueisiin sekä erityisesti tekoälyn hyödyntämiseen videopeleissä ja niiden kehityksessä. Tutkimuksessa tarkastellaan käytetyimpiä tekoälymenetelmiä peleissä, kuten äärellistä tilakonetta ja reitinhakua. Lisäksi käydään hieman läpi, mihin suuntaan kehitys on alalla menossa.

Teoriaosuuden lisäksi tutkimuksen tueksi suunniteltiin ja toteutettiin myös yksinkertainen tasohyppelypeli Unitylla ja C#-ohjelmointikielellä. Siinä tarkastellaan muutamaa yleisintä tapaa hyödyntää tekoälymenetelmiä tämänkaltaisessa pelissä. Monia näistä menetelmistä voidaan soveltaa riippumatta pelin monimutkaisuudesta tai siitä, onko se toteutettu 2D- vai 3D-ympäristössä. Perusperiaatteiltaan algoritmien toimintatavat ovat kuitenkin samanlaisia riippumatta näistä seikoista. Esimerkkipelissä keskitytään kuitenkin melko yksinkertaisiin tekniikoihin, jotka soveltuvat käytettäväksi kyseisessä peligenressä.

Pelissä toteutettiin yksinkertainen tekoäly tietokoneohjattuun hahmoon yhdistelemällä erilaisia tekniikoita itsenäiseen liikkumiseen ja ympäristön havaitsemiseen. Lisäksi rakennettiin äärellinen tilakone kahta eri tekniikkaa käyttäen. Yksittäisellä pelihahmolla on useita erilaisia tiloja, kuten hyökkäys ja partiointi, ja se voi olla vain yhdessä niistä kerrallaan. Lisäksi tutustuttiin hieman A*-reitinhakualgoritmin toimintaan käytännössä.

Lopputuloksena saatiin rakennettua toimiva peli siihen soveltuvaa tekoälyä hyödyntäen. Käytetyt tekoälymenetelmät saatiin toimivaan suunnitellun mukaisesti. Peli soveltuu myös hyvin jatkokehitykseen, eli sitä voitaisiin laajentaa lisäämällä esimerkiksi enemmän kenttiä ja erilaisia vihollisia, joiden toiminnan ohjaamisessa voisi hyödyntää myös muita kuin tässä projektissa käytettyjä tekoälyalgoritmeja. Myös proseduraalista generointia olisi mahdollista toteuttaa tällaisessa pelissä.

Avainsanat	tekoäly, videopelit, tietokonepelit, 2D, tasohyppely, Unity

Author	Erika Tarsala
Title	Implementing AI in 2D Platformer Game
Number of Pages	54 pages
Date	5 December 2021
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Professional Major	Software Engineering
Instructor	Antti Laiho, Senior Lecturer

Artificial Intelligence is a broad and constantly evolving field, and the way to harness AI in video games differs significantly from that regarding many other applications. The purpose of this Thesis is to examine the concept of Artificial Intelligence and its various aspects, and especially the use of AI in video games and their development process. The paper explores the most popular ways of applying AI in video games, the main trends and in which direction the game development is going in the field.

In addition to the theoretical part, a simple 2D Platformer game made with Unity and C# was implemented to support the research. In the game, a look at some of the most common ways to utilize AI methods in this particular level and genre of a video game is taken. Many of the methods can be applied to either a 2D or 3D environment and to a simple or a more complex game, as the basic principles of the algorithms are similar regardless of these factors.

A simple AI to a non-player character was implemented in the game by combining a few different techniques to detect the environment, a Finite State Machine in two different ways, and finally A* Pathfinding algorithm was also studied in practice.

As a result, a functional game with some suitable AI methods was created. The game is also well suited for further development. For example, it could be expanded by adding more levels and various enemies which could use different AI algorithms than the ones used in the present project. It would also be possible to implement procedural generation in a game of this kind.

Keywords	AI, artificial intelligence, video games, platformer

Sisällys

Käsitteet

Asset	Pelikehittäjien tai artistien luoma ilmainen tai maksullinen sisältö, jota voi ladata omaan käyttöön Unity Asset Storesta.
C#	”C sharp”, Microsoftin .NET-alustalle kehitetty olio-ohjelmointikieli.
Chattibotti	Chatbot, tekstiä tuottava ohjelma, jota käytetään esimerkiksi yritysten verkkosivuilla asiakaspalvelutehtävissä ihmiskontaktin korvaajana tai täydentäjänä.
Collider	Unityn komponentti törmäysten havaitsemiseksi fyysisten objektien välillä.
Enum	Lueteltu tyyppi ohjelmoinnissa.
FSM	Finite State Machine, äärellinen tilakone.
Gizmo	ns. “vempain”. Tässä insinöörityössä sillä tarkoitetaan Unityn työkalua visuaaliseen vian- tai asetusten määritykseen.
NPC	Non-playable character, tietokoneohjattu hahmo videopeleissä.
Platform	Tasohyppelypelin liikkumisalue.
Rigidbody	Mahdollistaa Unityn fysiikkamoottorin ominaisuudet peliobjektille.
Trigger	Colliderin ominaisuus, jolla saadaan havaittua vuorovaikutus fyysisten objektien välillä ja suoritettua kyseisessä tilanteessa haluttu koodi.
Unity	Pelimoottori, jolla voidaan kehittää 2D- ja 3D-pelejä.

1 Johdanto

Tekoäly on niin sanottua simuloitua älykkyyttä, eli sen tarkoituksena on jäljitellä ihmismäistä käyttäytymistä. Nykyisin käytettävien tekoälysovellusten päätavoite on kuitenkin pitkälti erilaisten tosielämän haasteiden ratkominen sekä manuaalisesti tehtävän työn ja inhimillisten erehdysten vähentäminen. Lyhyesti sanottuna, sovellusten halutaan helpottavan jokapäiväistä elämää. Ihmisen ja koneen vuorovaikutuksen toivotaan olevan mahdollisimman vaivatonta. [1; 2; 3.]

Tekoälytutkimus ja videopelit ovat molemmat jatkuvassa kehityksessä olevia aloja, joiden suosio myös kasvaa koko ajan. Yleisin tekoälyn soveltamistapa videopeleissä on erilaisten tietokoneohjattujen hahmojen eli NPC-hahmojen toiminnan ohjaaminen. Tekoälyn avulla niistä saadaan luotua realistisempia ja mielenkiintoisempia. [4.] Tämän opinnäytetyön käytännön toteutuksessa käsitellään erilaisia tapoja hyödyntää tekoälyä videopeleissä ja erityisesti juuri NPC-hahmojen toiminnassa.

Tekoälyyn tutustutaan ensin yleisellä tasolla sen historian ja kehityskaaren kautta, minkä jälkeen perehdytään tarkemmin tekoälyyn käsitteenä ja sen eri osa-alueisiin. Siitä jatketaan tekoälyn käyttöön videopeleissä, erilaisiin yleisimmin käytössä oleviin algoritmeihin sekä tulevaisuuden näkymiin.

Lopussa on selonteko tutkielman tueksi tehdystä 2D-tasohyppelypelistä, jossa on hyödynnetty muutamaa erilaista tekoälymenetelmää. Tarkoituksena oli kokeilla tekoälyn toteutusta myös käytännössä ja ennen kaikkea sitä, miten NPC-hahmot saadaan yksinkertaisessakin videopelissä vaikuttamaan älykkäiltä.

Peliprojektin vaiheista käydään ensin läpi suunnitelma, eli kuvataan hieman tarkemmin, millaisesta pelistä on kyse ja mitä tekoälyalgoritmeja siinä on tarkoitus käyttää. Sen jälkeen tehdään pieni katsaus varsinaisen pelin toteutusvaiheeseen, jonka jälkeen pureudutaan yksityiskohtaisemmin itse tekoälymenetelmien käyttöön.

Lopuksi analysoidaan vielä pelissä käytettyjen tekoälymenetelmien toimintaa, niiden toteutuksessa ilmenneitä haasteita sekä mahdollisia jatkokehitysajatuksia.

2 Tekoälyn historia ja kehitys

2.1 Tekoälyn historiaa

Yleisesti tekoälyn isänä pidetään Alan Turingia, englantilaista matemaatikkoa. Hän julkaisi vuonna 1950 artikkelin kehittämästään käytännön kokeesta, jonka tarkoitus oli selvittää, kykeneekö tietokone ihmisenkaltaiseen ”ajatteluun”. Jälkeenpäin koe on tunnettu Turingin testinä, jota käytetään edelleen yhtenä tekoälyn mittarina. [5.] Turing oli myös yhdessä kollegansa David Champernownen kanssa työstänyt shakkialgoritmin, joka sai nimekseen Turochamp yhdistelmänä molempien sukunimistä. Turochampia voidaan pitää historian ensimmäisenä tietokoneshakkiohjelmana, vaikka se tuolloin olikin pelkkä kynällä ja paperilla toteutettava algoritmi. Algoritmin logiikka perustui muutamaan shakin perussääntöön ja se pystyi päättelemään ainoastaan kaksi siirtoa ennakoivasti. Vertauksena maailman parhaat pelaajat miettivät noin kolmesta viiteen siirtoa eteenpäin, mutta etukäteen on mahdollista päätellä jopa 14 seuraavaa siirtoa. [6.]

Vuonna 1956 eli kaksi vuotta Turingin kuoleman jälkeen amerikkalainen tietojenkäsittelytieteilijä John McCarthy keksi termin ”artificial intelligence” eli tekoäly, ehdottaessaan sitä käsittelevän tutkimusseminaarin järjestämistä Dartmouth Collegessa. McCarthy teki merkittävän uran tietotekniikan ja tekoälytutkimuksen parissa ja kehitti muun muassa LISP-ohjelmointikielen vuonna 1958, josta tuli käytetyin ohjelmointikieli AI-sovelluksissa. [7.]

Ensimmäisen kokeellisen yhden neuronin eli perceptronin neuroverkon kehitti Frank Rosenblatt vuonna 1957, ja 60-luvulta alkaen tekoälytutkimus keskittyikin pitkälti juuri neuroverkkoihin. [8.] Näihin aikoihin kehitettiin myös minimax-algoritmi ja alfa-beta-karsinta, jotka loivat pohjan videopeleissä edelleen käytössä oleville tekoälymenetelmille. [9.]

70-luvulla tekoälytutkimus kärsi jonkinasteisesta inflaatiosta valtionrahoituksen puutteessa, mutta 80-luvulla se koki uuden tulemisen asiantuntijajärjestelmien parissa. Järjestelmät olisivat kuitenkin vaatineet laitteistolta niin paljon, etteivät ne pystyneet pitkään kilpailemaan suosiotaan kasvattaneiden ja edullisten kotitietokoneiden kanssa.

Vasta internetin kasvun myötä ja erityisesti vuosituhannen vaihteessa tekoälytutkimus lähti todelliseen nousukiitoon. [8.]

2.2 Tekoälyn määritelmä

Yksinkertaisesti sanottuna tekoälyn voisi sanoa olevan älykkyyttä, joka ei ole biologista. Tekoälyn määrittely sen sijaan on haastavaa, koska sillä voidaan tarkoittaa eri asioita tilanteesta riippuen, ja tutkimusalueena se myös elää ainaisessa muutoksessa. Teknologian kehityksen myötä tekoäly määritellään jatkuvasti uudelleen, eikä edes alan tutkijoilla ole olemassa sille yhtä ainoaa tarkkaa ja vakiintunutta määritelmää. [1.] Yleisin käsitys on, että tekoäly on ihmisenkaltaista kykyä suoriutua jostakin tehtävästä. Ideaalitilanteessa tekoälyn pyrkimyksenä on enemmänkin päätellä ja toimia rationaalisesti, mikä sulkee pois inhimillisten erehdysten mahdollisuuden. [2.]

Kun koneet voivat nähdä, kuulla, ymmärtää ja oppia vastaavalla tavalla kuin ihminen, niillä voidaan katsoa olevan tekoälyä. Tietokoneen etu ihmiseen verrattuna on kuitenkin sen mahdollisuus prosessoida valtavia tietomääriä lyhyessä ajassa. Jo vuonna 1997 IBM:n kehittämä shakkitietokone Deep Blue päihitti senhetkisen maailmanmestarin Garri Kasparovin normaalipituisilla turnausajoilla käydyssä ottelussa. [10.]

Turingin testi

Aiemmin mainitussa Turingin testissä ihminen haastattelee tietokoneen välityksellä kahta osanottajaa, joista toinen on ihminen ja toinen tietokone. Testin katsotaan olevan läpäisty, mikäli haastattelija ei keskustelun perusteella kykene erottamaan tietokonetta ihmisestä. [11.] Joltakin kannalta katsottuna testin onkin jo läpäissyt muutaman kerran chattibotti, joka johtaa haastattelijaa harhaan juttelemalla ihmismäisesti, eli esimerkiksi aiheen vierestä, vitsailemalla tai tekemällä kirjoitusvirheitä. Tulokset ovat kuitenkin kiisteltyjä. [11.]

Tämän päivän tekoälyä arvioitaessa Turingin testi on melko rajoittunut ja myös kritisoitu työkalu [12.], eikä se monessakaan sovelluksessa ole edes käyttökelpoinen arvioimaan tekoälyn kyvykkyyttä. Testi arvioi pikemminkin ihmismäistä kykyä ajatella kuin varsinaista älykkyyttä. Testin läpäisy voidaan siis käytännössä tulkita niin, että kone osaa ajatella, kunhan ihminen ei tunnista sitä koneeksi. Lisäksi testin keksimisestä on kulu-

nut jo yli 70 vuotta, eikä se näin ollen osaa alkuperäisessä muodossaan huomioida nykyisten tietokoneiden kykyä käydä läpi suuria määriä tietoa valtavalla nopeudella ihmiseen verrattuna, saati tekoälyn mahdollisuutta hyödyntää erilaisia sensoreita ympäröivän maailman aistimiseen. Nykyisten tekoälysovellusten tarkoituksena on pääasiassa käytännöllisten, arkea helpottavien ja todellisen maailman ongelmien ratkominen, ja ihmisten ja sovellusten välisen vuorovaikutuksen toivotaan olevan ennen kaikkea mahdollisimman vaivatonta ja hyödyllistä. Ihmismäinen ajattelu ei siis tällaisissa tilanteissa ole useinkaan edes päätavoitteena. [3.] Turingin testin lisäksi modernia tekoälyä voidaankin arvioida myös esimerkiksi mittaamalla sen suorituskykyä, datakapasiteettia sekä opettelu- ja päättelynopeutta. [13.]

Vahva ja heikko tekoäly

Heikolla tekoälyllä tarkoitetaan tilannetta, jossa kone analysoi yhtä ongelmaa kerrallaan tietyn ohjelmoidun logiikan mukaisesti. Heikon tekoälyn pohjalla on siis aina enemmän tai vähemmän ihmisen määrittämiä ohjeita ja ehtoja, eli niin sanottua simuloitua älykkyyttä. Kaikki nykyään käytössä olevat tekoälymenetelmät ovat heikkoa tekoälyä. [14.]

Vahvasta tekoälystä puhuttaessa viitataan sen sijaan jo lähes inhimilliseen älykkyyteen, eli ainakin jonkin asteisen tietoisuuden omaaviin koneisiin, jotka kykenevät suoriutumaan useista erilaisista tehtävistä samoin kuin ihminen. Toistaiseksi vahva tekoäly on olemassa vasta teoreettisella tasolla, eikä sellaisen kehittäminen ole kovin realistista lähitulevaisuudessakaan. [15.]

Näiden lisäksi tunnetaan myös termi supertekoäly (Super AI, Artificial Superintelligence), joka vastaa enemmänkin tieteiselokuvien uhkakuvia äärimmilleen kehittyneestä, ihmisen päihittävästä ja lopulta koko ihmiskunnan sukupuuttoon syöksevästä tekoälystä. [14.]

Tekoälyn suuntaukset

Tekoälyä hyödynnetään nykypäivänä laajalti muun muassa logistiikan, terveydenhuollon ja erityisesti teknologiateollisuuden alalla. Itseohjautuvat autot ja älykotijärjestelmät kehittyvät ja yleistyvät jatkuvasti. Suorittimien laskentateho, koneoppimisessa hyödynnettävä datan määrän lisääntyminen sekä algoritmien saatavuus on mahdollistanut tekoälyn nopean kehityksen 2010-luvulta lähtien. [1.]

Yhteisiä tekoälylle tyypillisiä ominaisuuksia ovat autonomisuus ja adaptiivisuus. Autonomisella tarkoitetaan kykyä suoriutua monimutkaisistakin tehtävistä ilman jatkuvaa ohjausta. Adaptiivisuus taas on kokemukseen perustuvaa oppimista ja sen hyödyntämistä tulevissa tehtävissä. [1.]

Koneoppiminen, neuroverkot ja syväoppiminen

Koneoppiminen on yksi tämän hetken hallitsevimmista tekoälyn suuntauksista. Sitä käytetään esimerkiksi kuvien ja puheen tunnistuksessa ja tuottamisessa, haku- ja käännöskoneissa, kohdennetussa mainonnassa, sovellusten sisällön personoinnissa, videopeleissä, robotiikassa ja monenlaisissa datan analysointityökaluissa. Koneoppimisen avulla sovelluksista on mahdollista saada reaktiivisia ja mukautumiskykyisiä. [1.]

Koneoppimisen yhteydessä puhutaan usein ohjatusta ja ohjaamattomasta oppimisesta sekä vahvistusoppimisesta. Kaikkia käytössä olevia menetelmiä ei kuitenkaan voida ryhmitellä tiukasti vain yhteen näistä kategorioista, joten lisäksi on olemassa erilaisia välimuotoja. Koneoppimisessa sovelletaan hyvin pitkälti vanhoja tilastotieteen menetelmiä, kuten lineaarista regressiota ja Bayesin tilastoja. [1.]

Ohjatussa oppimisessa (engl. supervised learning) koneelle annetaan syötteenä opetusdata, joka sisältää myös vaadittavan ratkaisun. Tätä koneoppimisen muotoa sovelletaan monessa käytännön ongelman opettamisessa, kuten roskapostin suodatuksessa. [1.]

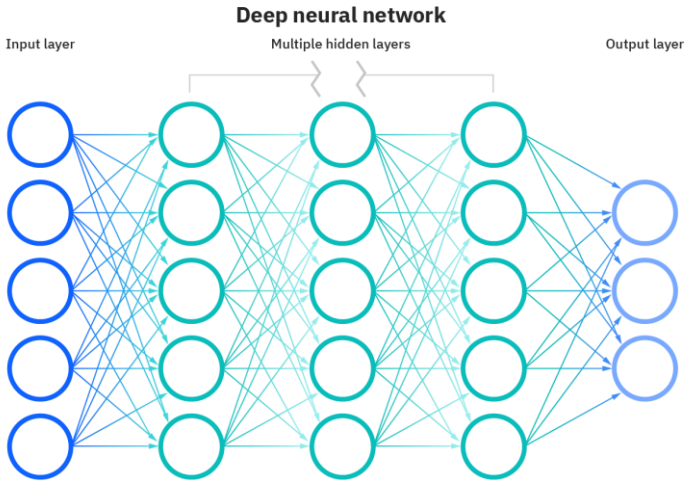
Ohjaamaton oppiminen (engl. unsupervised learning) taas tarkoittaa, että kone prosessoi ja analysoi opetusaineistoa muodostaen ratkaisut itsenäisesti koneoppimisalgoritmien perusteella. Ohjaamatonta oppimista voidaan hyödyntää esimerkiksi datan analysoinnissa ja kuvantunnistuksessa. [1.]

Vahvistusoppimisessa (engl. reinforcement learning) kone saa palautetta suorituksestaan ja pystyy sen pohjalta parantamaan sitä. Oppiminen tapahtuu siis yrityksen, erehdyksen ja niiden pohjalta kertyvän kokemuksen kautta. Vahvistusoppimista käytetään paljon esimerkiksi itseohjaavien autojen kehityksessä. [1.]

Neuroverkot on koneoppimisen alalaji. Nimensä mukaisesti ne jäljittelevät toiminnallaan ihmisaivojen rakennetta. Keinotekoiset neuronit ovat kytkeytyneenä toisiinsa ver-

kon tavoin, lähettäen ja vastaanottaen signaaleja keskenään. Neuroverkot koostuvat useista kerroksista, joissa syötteenä saatu data siirtyy aina eteenpäin seuraavalle kerrokselle. [16.]

Syväoppiminen taas on neuroverkkojen alalaji, jonka nimikin viittaa juuri neuroverkkojen kerroksiin ja syvyyteen. Kerrokset koostuvat kolmesta erilaisesta tyypistä: syötekerroksesta, piilokerroksesta ja ulostulokerroksesta. Piilokerroksia voi olla useita. [16.]



Kuva 1. Neuroverkon rakennetta [16.]

Kuva 1 havainnollistaa neuroverkon toimintaa. Syötekerrokselle (input layer) annetaan data, joka käsitellään välissä olevissa kerroksissa. Ratkaisu tulee ulostulokerroksesta (output layer). Syväoppiminen hyödyntää keinotekoisten neuroverkkojen hierarkkista tasoa suorittaakseen koneoppimisprosessin, eli tieto kulkee kaikkien kerrosten läpi mahdollistaen monimutkaistenkin rakenteiden ja säännönmukaisen tiedon oppimisen. Esimerkiksi Googlen hakualgoritmi hyödyntää neuroverkkoteknologiaa. [16.]

Yksi suurimpia tekoälyn haasteita on luonnollisen kielen ymmärtäminen. Tietokoneet ymmärtävät määrättyjä sanoja, joten on vaikeaa opettaa niitä tunnistamaan puheesta lukuisia erilaisia äänensävyjä, aksentteja ja muita vivahteita. Toisin sanoen kieltä on hyvin helppo tulkita väärin. Kielellisesti haastavaa sekä puheen että tekstin tulkitsemisessa tai tuottamisessa on myös asioiden yhdistäminen, kuten lauseiden liittyminen toisiinsa tai useamman kuin yhden merkityksen omaavat sanat tai lauseet. Esimerkiksi kysymyksen ymmärtäminen on tietokoneelle vaikein osuus, kun taas ihmiselle se on helpoin. [17.] Neuroverkkoja ja syväoppimista hyödyntämällä on kuitenkin saavutettu jo

merkittävää kehitystä muun muassa puheen- ja tekstintunnistuksessa sekä kielen tuottamisessa. [16.]

3 Tekoäly videopeleissä

3.1 Pelitekoälyn tavoitteet

Videopeleissä on yleensä syytä tietää ennalta, mitä pelaaja tulee kokemaan. Tekoälyn pitää siis olla tämän suhteen ennustettavissa. Esimerkiksi neuroverkkojen jatkuva mukautuminen ja oppiminen kaikesta mahdollisesta voisi aikaansaada täysin odottamattomia tilanteita, mitkä pahimmillaan saattaisivat pilata pelikokemuksen tai jopa aiheuttaa pelin toimimattomuuden. [18.]

Yleisesti ottaen pelien tarkoituksena on tarjota pelaajalle viihdettä ja haastetta, joten myös peleissä käytetyn tekoälyn päätavoite on ennen kaikkea mahdollisimman mukaansatempaava pelikokemus. Jos esimerkiksi vihollisten päihitys olisi jatkuvasti käytännössä mahdotonta, kokemus voisi lähinnä turhauttaa, ja pelaaminen todennäköisesti loppuisi melko lyhyeen. Vaikeustason tulisi kasvaa pelin edetessä, ja erityisen haastavassakin pelissä pelaajan olisi hyvä saada aika ajoin myös onnistumisen kokemuksia. Pelitekoäly poikkeaa siis monen todellisen maailman sovelluksen tavoitteesta, koska useimmiten se ei niin sanotusti saa olla liian hyvä. Tekoälyn avulla halutaan enemmänkin luoda todentuntuinen maailma, käytännössä illuusio aidosta kokemuksesta, jolloin pelaaja voi tuntea kilpailevansa inhimillistä, mielenkiintoista vastustajaa vastaan. Pelitekoäly muodostuu ainakin toistaiseksi ennalta määriteltyjen ehtojen ja rajojen mukaan ja keskittyy pääasiassa ympäristön tarkkailuun ja päätöksentekoon sekä toimenpiteisiin niiden perusteella. [19.]

Toisin kuin todellisessa maailmassa käytettävissä tekoälysovelluksissa, videopeleissä tämä ympäristön tarkkailu voidaan pitää melko yksinkertaisena, koska tekoälyn ei tarvitse analysoida sitä vaikkapa ottamalla kuvia ja vertaamalla niitä verkosta löytyvään dataan koneoppimista hyödyntämällä. Esimerkiksi vihollisen havaitsemiseen ei tarvita kuvantunnistusalgoritmeja, koska sen olemassaolo on jo tiedossa ja tieto voidaan välittää suoraan päätöksentekoprosessiin. [19.]

Koneoppimista käytetään vielä suhteellisen vähän itse pelin toimintalogiikassa, kuten NPC-hahmoissa [20.]. Sen sijaan kasvavassa määrin sitä hyödynnetään varsinaisessa pelikehityksessä sekä tekoälyalgoritmien kehitysprosesseissa muun muassa tekoälyn suorituskyvyn testaamiseen [17.]. Alphabet Inc -yhtiön DeepMind on rakentanut tekoälyn, joka pystyy pelaamaan videopelejä ihmisenkaltaisesti. Se on jo päihittänyt ammattipelaajia kiinalaisen GO-lautapelin lisäksi myös Blizzardin StarCraft II -strategiapelissä. Koneoppimiseen perustuva tekoäly koulutetaan käyttämällä vahvistus- ja syväoppimisen yhdistelmää esimerkiksi tekoälyn pelaamisella itseään vastaan nopeutettuna. [21.]

3.2 Peleissä käytettäviä algoritmeja

Pelitekoälyn ydinkomponentit ovat erilaiset reitinhaku- ja käyttäytymisalgoritmit. Niitä sovelletaan lähinnä NPC-hahmojen toiminnassa, jotta hahmoista on mahdollista saada älykkäältä vaikuttavia ja tarpeeksi haastavia vastustajia pelaajalle. [19.]

Seuraavaksi tutustutaan tarkemmin muutamaan yleisimpään tapaan toteuttaa tekoälyä videopeleissä.

Reitinhaku

Reitinhaulla (engl. pathfinding) tarkoitetaan reitin etsimistä lähtöpisteestä päätepisteeseen eli esimerkiksi pelihahmon kulkureittiä määrättyyn kohteeseen. Reitinhaun tarkoituksena on tehdä liikkumisesta realistista, kun hahmo osaa väistellä tiettyjä kohteita tai löytää nopeimman reitin päämääräänsä. Erityisesti sitä käytetään NPC-hahmoilla erilaisissa toiminta-, seikkailu- tai ajopeleissä. Myös esimerkiksi strategiapeleissä pelaajan liikuteltavissa olevat hahmot käyttävät reitinhakua, kun pelaaja asettaa niille päämäärän pelikentällä. [19.]

Reitinhaku on yksi käytetyimmistä tekoälytekniikoista videopeleissä, joten sen avuksi on kehitetty useitakin eri algoritmeja mahdollisimman optimaalista suoritusta varten kussakin tilanteessa. Yleisimmät niistä ovat A*-algoritmi ja Dijkstran algoritmi. [22.]

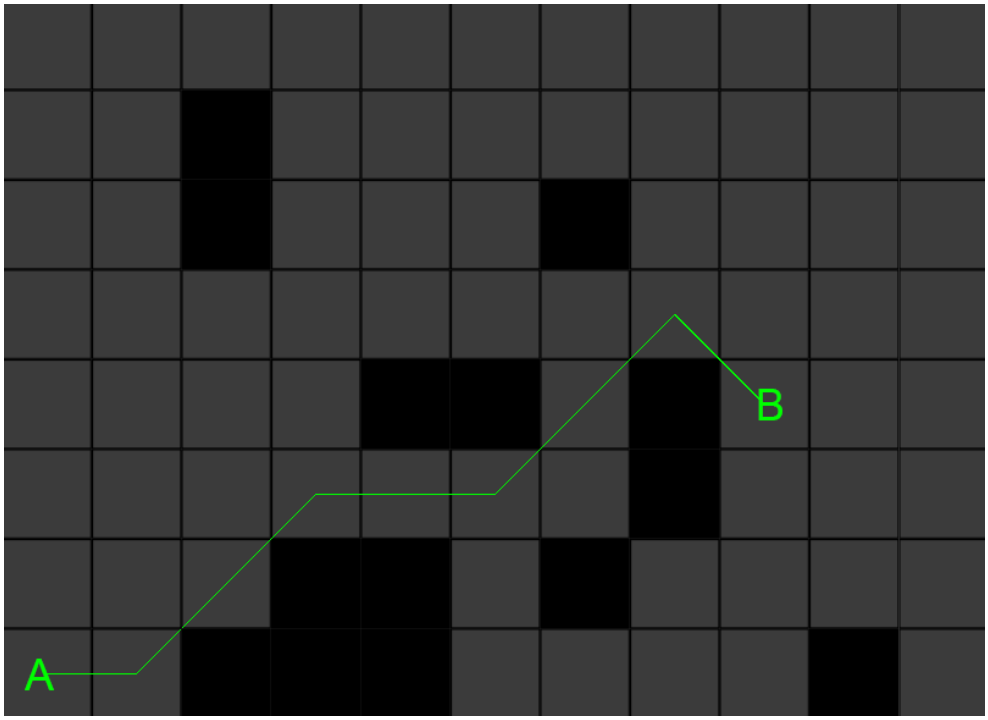
A*-algoritmi

A* on heuristiikkaan perustuva reitinhakualgoritmi. Sillä tarkoitetaan, että se löytää jatkuvalla etäisyyden arvioinnilla lyhyimmän reitin alkupisteestä loppupisteeseen. Se on tarkkuutensa ja suorituskykynsä ansiosta yksi käytetyimmistä reitinhakumenetelmistä. Sen tavoitteena on tunnistaa kuljettavissa ja ei-kuljettavissa olevat alueet ja löytää niiden perusteella lyhyin reitti kahden pisteen välillä. A* arvioi jo alkusolmussa optimaalisen reitin päämäärään, mutta päivittää arviota joka pisteessä uudelleen ja palaa joskus jopa takaisin löytääkseen vaihtoehtoisen, alkuperäistä paremman reitin. [22.]

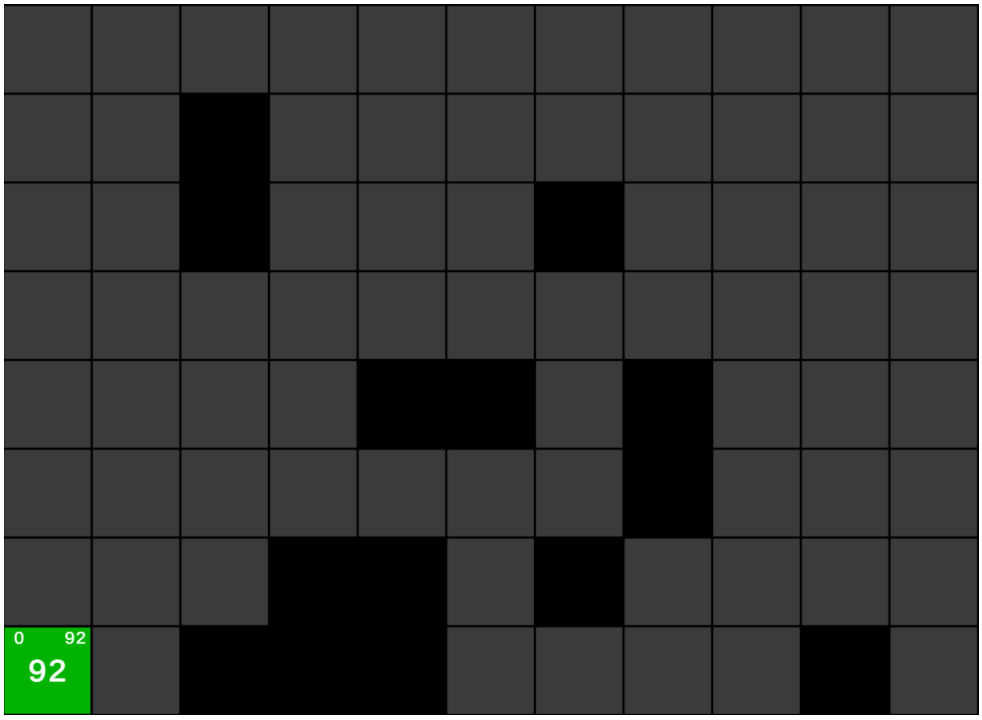
A*-algoritmissa jaetaan tarvittava pelialue ruudukkoon, jossa jokainen ruutu eli solmu (engl. node) vastaa yksikköä. Pelikentästä tulee siis kaksiulotteinen taulukko, jossa jokaisesta solmusta voi liikkua kahdeksaan eri suuntaan, mikäli kaikki viereiset solmut ovat kuljettavissa olevaa aluetta. Solmulla on kolme arvoa: ensimmäisenä on G-arvo, joka on kulkemisen hinta aloitussolmusta. Suoraan vieressä sijaitsevaan solmuun kulkemisen hinta voi olla kokonaislukuina vaikkapa 10, jolloin viistoon kulkemisen hinta olisi 14. Pelityypistä ja tavoitteista riippuen voi olla syytä huomioida myös painovoiman, vertikaalisen liikkeen ja kuljettavan maaston vaikutukset hintaan. Esimerkiksi vedessä kulkiessa hinta voisi olla suurempi ja alamäessä pienempi. [22.]

H-arvo on heuristinen hinta viimeiseen solmuun kuljettavasta kokonaismatkasta eli käytännössä lopullinen hinta-arvio päätepisteen saavuttamiselle. Kolmantena on vielä F-arvo, joka on G- ja H-arvojen yhteenlaskettu summa. F-arvon avulla algoritmi pystyy priorisoimaan solmut, jotka ovat lähimpänä päämäärää. Mitä lähempänä arvioitu liikkeen hinta H on todellista liikkeen hintaa, sitä tarkempi lopullinen reitti tulee olemaan. Jos arviota ei ole asetettu, generoitu reitti ei välttämättä ole lyhyin, mutta todennäköisesti lähellä sitä. Algoritmi valitsee seuraavan kuljettavan solmun pienimmän F-arvon mukaan. Jos saman F-arvon omaavia solmuja on useampia, valitaan se, jolla on pienin H-arvo. Kuljettaessa päätepistettä kohti G-arvo siis kasvaa jatkuvasti samalla, kun H-arvo pienenee. [22.]

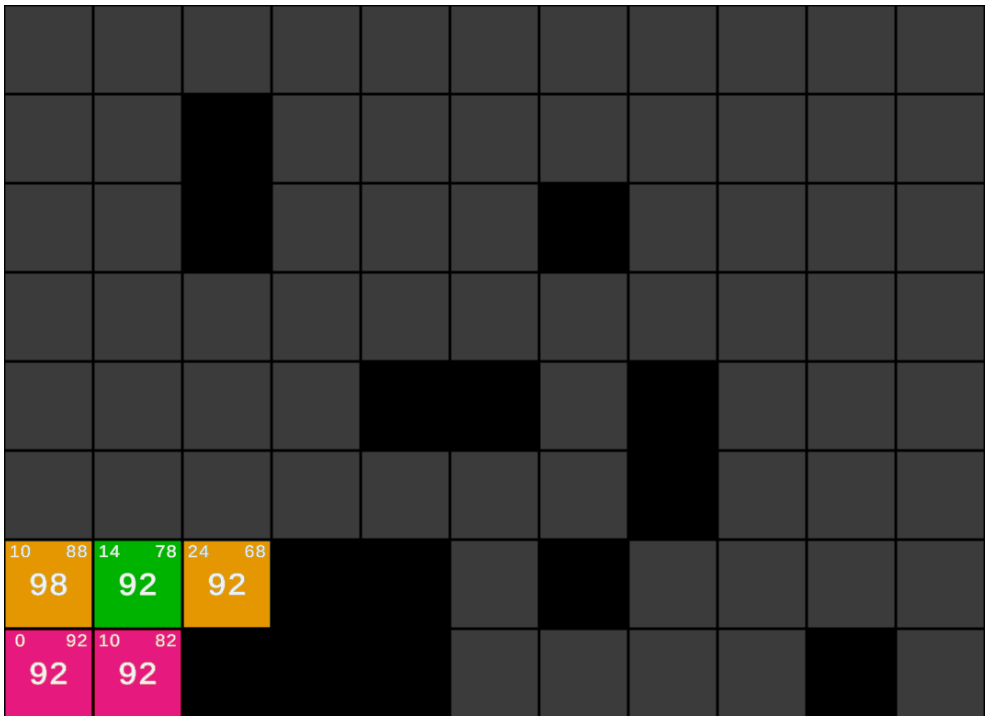
Algoritmissa on kaksi listaa solmuille. Avoin lista on kaikille kuljettavissa oleville solmuille, jotka pitää käydä läpi, ja suljettu lista jo läpikäydyille solmuille. Reitinhaku jatkuu niin kauan, kun nykyinen solmu vielä löytyy avoimelta listalta tai lista on tyhjä (= reittiä ei löytynyt). [22.]



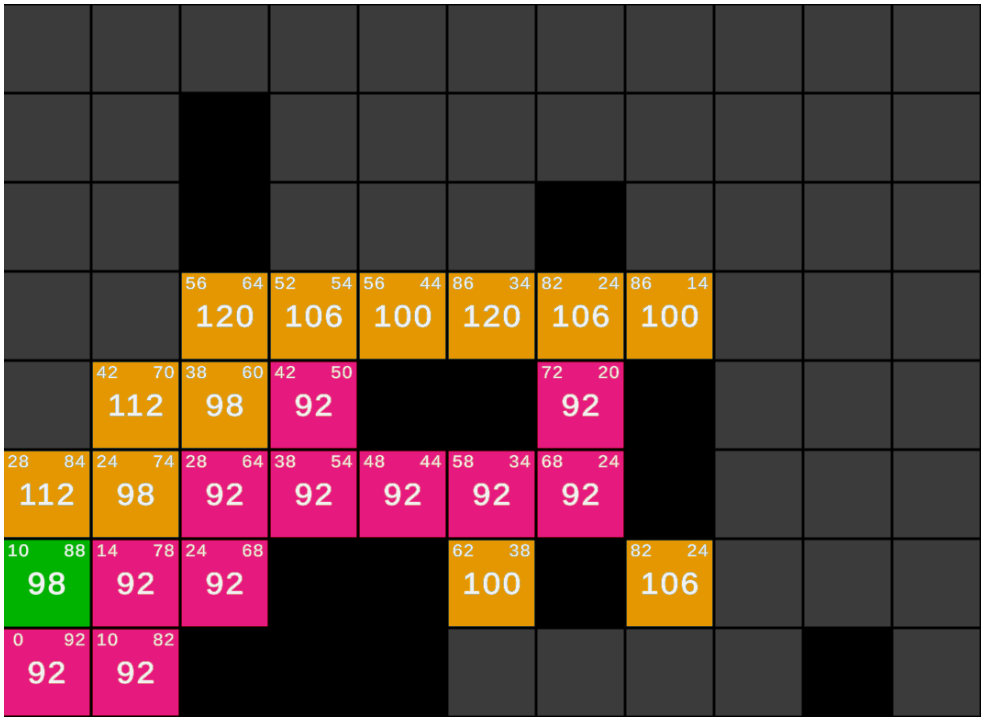
Kuva 2. Nopein reitti pisteestä A pisteeseen B.



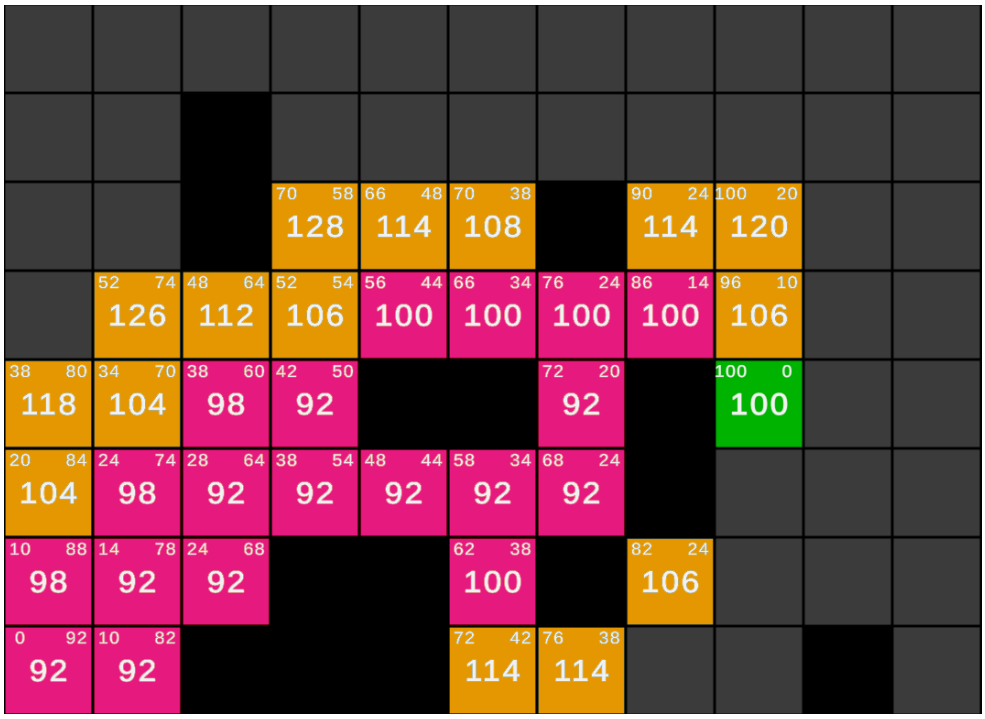
Kuva 3. Ensimmäisen solmun arvot.



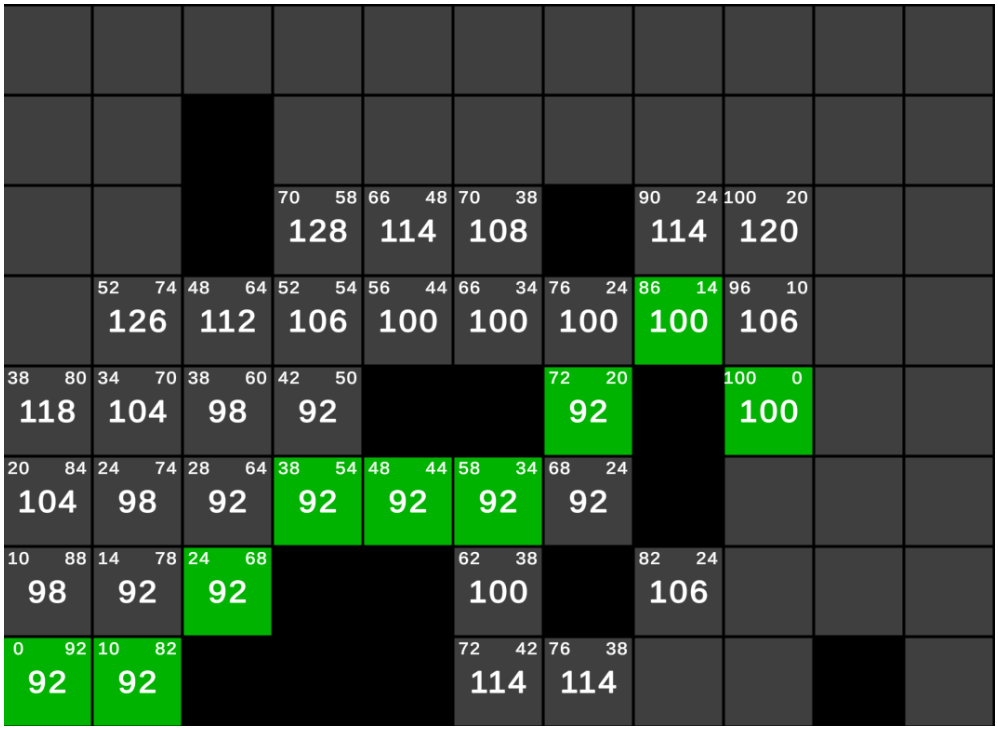
Kuva 4. Algoritmi on tutkinut viereiset solmut ja siirtynyt seuraavaan.



Kuva 5. Algoritmi on palannut takaisinpäin löytäessään alkupäästä pienempiä F-arvoja.



Kuva 6. Kaikki läpikäydyt solmut.



Kuva 7. Kuljettu reitti.

Kuvat 2-7 havainnollistavat reitinhakualgoritmin toimintaa. Kuvassa 2 algoritmi on laskenut nopeimman reitin pisteestä A pisteeseen B. Kuva 3 näyttää ensimmäisen solmun

arvot. G-arvo on 0 (vasen yläkulma), H-arvo 92 (oikea yläkulma) ja F-arvo 92. Kyseisestä solmusta voi liikkua kolmeen eri suuntaan. Kuvassa 4 algoritmi on tutkinut viereiset solmut ja siirtynyt seuraavaan. Kuvassa 5 nähdään, kuinka käytyään läpi useita solmuja, algoritmi on palannut takaisinpäin löytäessään sieltä pienempiä F-arvoja. Punaiset solmut ovat suljetulla listalla, eli jo läpikäytyjä solmuja. Kuvassa 6 näkyvät kaikki solmut, jotka käytiin läpi päämäärän saavuttamiseksi ja kuvassa 7 lopullinen kuljettu reitti.

Dijkstran algoritmi

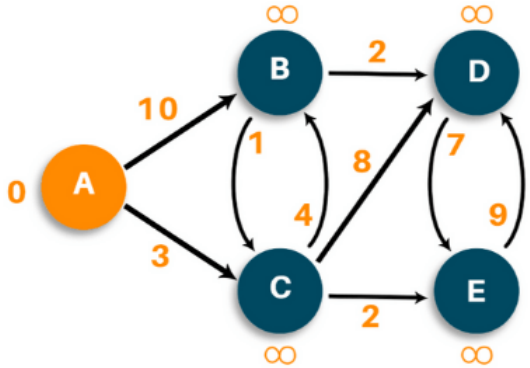
Dijkstran algoritmi etsii parhaan reitin lähtöpisteestä päätepisteeseen läpikäymällä koko graafin, kunnes lopettaa sen reitin löydettyään. Se vie siis enemmän laskentatehoa kuin esimerkiksi A*-algoritmi, koska laskettavien solmujen lukumäärä on suurempi. [23.]

Dijkstran algoritmia kuvataan painotetun graafin avulla, joka koostuu solmuista ja kaarista (engl. edge). Painotetulla tarkoitetaan, että jokaisella kaarella on paino (engl. weight) eli hinta, joka tarkoittaa, kuinka kauan kestää kulkea sitä pitkin seuraavaan solmuun joko ajallisesti tai matkallisesti. Reitinhaussa halutaan löytää lyhyin tai nopein reitti yhdestä solmusta toiseen. [23.]

Algoritmi laskee jokaiselle solmulle hinnan, joka alussa on aloitussolmulla 0 ja kaikilla muilla solmuilla ääretön. Solmut jaetaan kahteen joukkoon: läpikäydyt ja läpikäymättömät. Alkutilanteessa siis kaikki solmut ovat läpikäymättömien solmujen joukossa. [23.]

Algoritmi toistaa kaksivaiheista prosessia. Ensin päivitetään arvioitu hinta naapurisolmuille, jonka jälkeen valitaan seuraava läpikäytävä solmu. Aloitussolmu merkitään etsityksi, ja solmut, joiden arvioitu hinta on edelleen ääretön, merkitään läpikäymättömiksi. Seuraava kuljettava solmu on aina läpikäymätön solmu, jolla on pienin arvioitu hinta, joten valinta tehdään sen perusteella. [23.]

Kuva 8 (alla) havainnollistaa Dijkstran algoritmin toimintaa visuaalisesti. Alussa alkusolmun A arvo on 0 ja kaikilla muilla solmuilla B, C, D ja E se on ääretön ∞. Lisäksi näkyy kuljettavan matkan hinta jokaisen solmun välillä. Seuraava askel tässä olisi päivittää viereisten solmujen hinnat niin, että B saa arvon 10 ja C saa arvon 3. [23.]



Kuva 8. Dijkstran algoritmin toiminta [23.]

Utility AI

Utility AI (myös utility system) eli niin kutsuttu hyödyllisyysjärjestelmä perustuu nimensä mukaisesti hyödyllisimmän vaihtoehdon valitsemiseen useista suoritettavissa olevista toiminnoista tietyssä tilanteessa. Vihollisella voi esimerkiksi olla valittavanaan erilaisia taikoja, jotka kaikki tuottavat eri määrän vahinkoa pelaajaan. Loogisesti ajateltuna paras vaihtoehto olisi aina valita voimakkain taika, mutta se ei välttämättä johda parhaaseen lopputulokseen kyseisellä hetkellä. Ensinnäkin taioilla voi olla eripituiset suoritusajat. Osumisprosentti voi vaihdella taikojen välillä tai etäisyys pelaajaan vaikuttaa taian onnistumiseen. [24.]

Algoritmi laskee jokaisen suoritettavissa olevan toiminnan ”hyötypisteet” (engl. utility score) juuri kyseisellä hetkellä, ja osaa sen perusteella päättää parhaan mahdollisen toiminnan suoritettavaksi. Suurin osa peleistä ei kuitenkaan ole deterministisiä, joten tarkan hyödyllisyyden määrittäminen ei yleensä ole mahdollista. Laskenta perustuukin osittain todennäköisyyksiin. Yleisin tapa tähän laskutoimitukseen on kertoa kyseisen toiminnan hyötypisteet jokaisen mahdollisen ulostulon todennäköisyydellä, ja sitten laskea yhteen saadut arvot. Näin saadaan muodostettua odotettavissa oleva hyöty (engl. expected utility) kyseiselle toiminnalle. [24.]

Toinen esimerkki Utility AI:n käytöstä voisi olla esimerkiksi vihollisen hyökkäystilanne. NPC:n on mahdollista hyökätä kenen tahansa kimppuun, mutta se valitsee kohteet sen perusteella, miten kaukana ne ovat. Toisin sanoen hyöty kasvaa sitä myötä, kun NPC:n ja kohteen välinen etäisyys pienenee. [24.]

Tähän järjestelmään pohjautuen esimerkiksi peliin Dragon Age: Inquisition on rakennettu monimutkainen käyttäytymisen päätösjärjestelmä (engl. behaviour decision system). Pelaajalla on tekoälyä käyttäviä NPC-kumppaneita, jotka avustavat esimerkiksi taistelutilanteissa. Jokaisella kumppanihahmolla on käytettävissään erilaisia kykyjä, joiden hyödyllisyys voidaan määrittää joka tilanteessa. Ne voivat käyttää taistelussa sekä aseita että taikoja, mutta lisäksi niiden täytyy osata säännöstellä resurssejaan, kuten elämäpisteitä, taikavoimaa ja kestävyyttä. Hahmon selviytymisen kannalta on siis tärkeää tietää, kannattaako tietyllä hetkellä uhrata esimerkiksi kestävyyttä jonkin toiminnan suorittamiseksi. Kerrallaan voi suorittaa vain yhden toiminnan. Lisäksi eri toiminnoilla on erilaiset hyötyarvot, eli jotkin kyvyt ovat hyödyllisempiä kuin toiset. Joillakin kyvyillä voi olla myös useita eri tarkoituksia. Esimerkkinä soturihahmolla on kyky nimeltä ”charging bull”, joka on erinomainen käytettäväksi taistelutilanteessa, mutta tämän lisäksi suureksi hyödyksi myös pakenemistilanteessa. Utility AI:ta ei siis käytetä pelkästään selvittämään, onko jonkin toiminnan suorittaminen kannattavaa, vaan myös osoittamaan, miten kykyjä voidaan käyttää hahmon hyödyksi parhaalla mahdollisella tavalla perustuen senhetkiseen tilanteeseen kokonaisuudessaan. [25.]

Tätä varten peliin on määritelty datarakenne, niin sanottu käyttäytymisen palanen (engl. behaviour snippet), joka on olennainen osa järjestelmän toimintaa. Jokainen tällainen palanen (jäljempänä snippet) sisältää tarvittavan tiedon kyvyn arvioimista varten tietyllä ajanhetkellä ja lisäksi hyöty-yhtälön sen suorittamisen kannattavuuden laskemiseksi. Esimerkiksi ase voi sisältää yhden tai useamman snippetin, joiden avulla hahmo tietää, miten asetta käytetään. Kun ase otetaan käyttöön, snippetit rekisteröidään hahmolle hyötyjärjestelmän kautta. Jos taas kyseinen ase poistetaan käytöstä, rekisteröinti perutaan. Tämän avulla järjestelmä pystyy suorittamaan laskennan saatavilla olevista toiminnoista. Yhdellä hahmolla on keskimäärin 10-20 snippetiä käytössä kerrallaan. Hyödyn määrittämiseen käytetään käytöspuun muunnosta, jota kutsutaan arviointipuuksi (engl. evaluation tree). Sen tehtävänä on käytännössä muodostaa hyötypisteet kyseessä olevalle snippetille. Tätä pistemäärää hyödynnetään myöhemmin, kun arvioidaan suhteellista hyötyä kahden snippetin välillä. [25.]

Useimmat pelihahmojen käyttämät kyvyt vaativat suorittamista varten kohteen. Tiettyä taikaa ei esimerkiksi kannata tuhlata viholliseen, jolla on korkea resistanssi tai immuniteetti kyseistä taikatyyppiä kohtaan, koska tällöin vain tuhlataan resursseja. Myös kohteen valinta on siis kriittinen toimenpide hyödyn laskentaa tehtäessä. Kaikki mahdolliset kohteet pitää siis ottaa huomioon ja valita ne, jotka tarjoavat suurimman hyötyarvon.

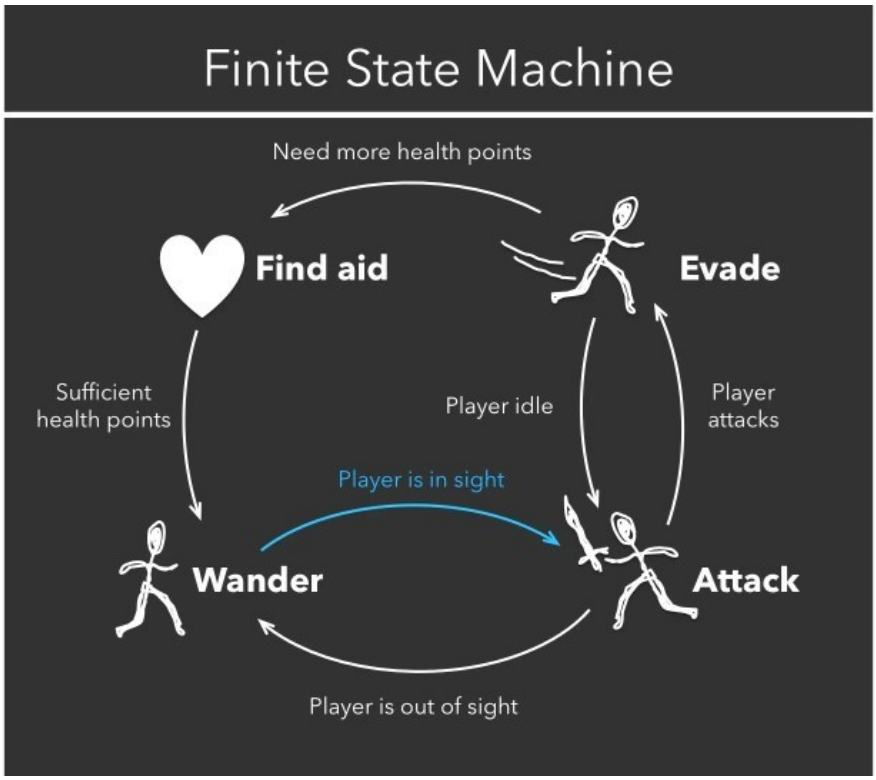
Hahmon päivityssilmukassa pyörii jokaisen rekisteröidyn snippetin arviointipuu, jonka tuottama pistemäärä sekä kohde talletetaan snippetin kanssa summaustaulukkoon. Kun kaikki puut ovat suorittaneet laskennan, suurimman pistemäärän saanut snippet valitaan suoritettavaksi. Myös suoritusta varten on oma käytöspuunsa, joka kertoo, miten toiminta pitäisi suorittaa. Esimerkiksi jonkin hyökkäystoiminnan suoritus voi vaatia, että hahmon tulee olla paitsi tietyn etäisyyden päässä kohteesta, myös kohteen takana, ja vasta sen jälkeen kohdetta isketään aseella. [25.]

Äärellinen tilakone

Tilakoneen tarkoituksena on määritellä kaikki mahdolliset tilanteet, jotka esimerkiksi NPC-hahmo voi kokea, sekä toiminta näissä tilanteissa. Lyhyesti sanottuna se on siis ryhmä sääntöjä, miten tulee käyttäytyä missäkin tilanteessa. [19.]

Äärellinen tilakone (engl. finite state machine) koostuu ennalta määritetystä, eli rajallisesta määrästä tiloja, joista vain yksi voi olla kerrallaan aktiivinen. Kukin tila määrittää, mihin tilaan se siirtyy seuraavaksi vastaanottamansa tiedon perusteella. [19.] Näin on mahdollista luoda monimutkaiselta vaikuttava pelihahmo, joka voi muuttaa käyttäytymistään useita kertoja ja toimia yllättävälläkin tavalla. Kun tähän lisätään vielä satunnaisuutta eri tilojen kestoihin tai tilan vaihtamiseen vaikuttaviin ehtoihin, hahmosta tulee vielä realistisemman oloinen. Tilakonetta voidaan hyödyntää muihinkin pelin toiminnallisuuksiin kuin NPC-hahmoihin. Esimerkkinä sen voisi toteuttaa myös pelaajahahmolle, jolloin NPC:n tai jonkin muun peliobjektin voisi ohjelmoida reagoimaan eri tavoin sen perusteella, missä tilassa pelaaja milloinkin on. Tämän lisäksi tilakonetta voitaisiin käyttää esimerkiksi pelin äänimaailman toteutuksessa.

Mitä enemmän yksityiskohtia NPC:n toiminnalle keksitään, sitä monimutkaisempi tilakoneesta tulee. Useiden eri tilojen lisäksi voidaan esimerkiksi määrittää, miten kauan viholliset jahtaavat pelaajaa, jäävätkö ne paikoilleen jahtaamisen loputtua vai palaavatko takaisin alkuperäiseen pisteeseensä.



Kuva 9. Esimerkki äärellisestä tilakoneesta [4].

Kuvassa 9 on esimerkki äärellisen tilakoneen toiminnasta. NPC on ensin vaeltelutilassa (Wander), josta se siirtyy hyökkäystilaan (Attack) nähdessään pelaajan. Jos pelaaja ei enää ole näkyvissä, NPC siirtyy takaisin vaeltelutilaan. Pelaajan hyökätessä NPC saattaa siirtyä välttelytilaan (Evade) ja takaisin hyökkäystilaan, mikäli pelaaja lopettaa hyökkäämisen. Jos NPC:n elämäpisteet laskevat tarpeeksi alas, se siirtyy etsi apua -tilaan (Find aid), josta takaisin vaeltelutilaan, mikäli elämäpisteitä on taas tarpeeksi. [4.]

Hierarkkinen tilakone

Jos tilakoneessa on paljon erilaisia tiloja, kannattaa siitä tehdä hierarkkinen. Tällöin samankaltaiset tilat voidaan jakaa alitiloiksi yhden päätilan alle. Esimerkkinä hyökkäystila voitaisiin jakaa kahdeksi tai useammaksi alitilaksi erilaisille hyökkäyksille. Myös ylimääräiset ja päällekkäiset siirtymiset tilojen välillä saadaan vältettyä tilahierarkiaa soveltamalla. [19.]

Tilakone ei välttämättä ole toimivin ratkaisu aivan jokaisessa peligenressä. Jos esimerkiksi strategiapelissä vihollinen käyttäytyisi aina samalla tavalla samassa tilanteessa, pelaaja oppisi hyvin nopeasti, miten tietokone voitetaan. Koko pelin idea häviäisi sen

myötä. Tämä ennustettavuuspiirre onkin monessa tapauksessa yksi tilakoneen heikkouksista. [4.]

Käytöspuut

Monimutkaisempaa käyttäytymistä tavoiteltaessa voi esimerkiksi tulla tarve toteuttaa samoja toimintoja tilasta riippumatta tai useassa eri tilassa. Ehtojen lisääminen jokaiseen tilaan erikseen on kuitenkin työlästä. Tällainen rakenne on helpompi toteuttaa käytöspuiden avulla. Se on nimensä mukaisesti puurakenne, jonka suorittaminen aloitetaan juurisolmusta. Jokaisella solmulla voi olla rajattomasti alisolmuja. Solmut palauttavat yhden kolmesta arvosta: onnistui (engl. succeeded), epäonnistui (engl. failed) ja käynnissä (engl. running). Päätössolmujen sijaan on ”koristelijasolmuja” (engl. decorator), joilla on vain yksi lapsisolmu. Jos niiden toiminta onnistuu, ne suorittavat lapsisolmun toimet. Toimintaa suorittavat solmut palauttavat käynnissä-arvon. Käytöspuun avulla ei tarvitse toteuttaa täsmällisiä siirtymisiä tilasta toiseen. Sen sijaan oikea päätöksenteko toteutetaan senhetkisen tilanteen perusteella riippumatta siitä, mitä tilaa oltiin aiemmin suorittamassa. [19.]

Hyvä esimerkki monimutkaisen käytöspuun toteutuksesta löytyy vuonna 2014 julkaistusta pelistä Alien: Isolation. Siinä nähdään, kuinka tekoälyn avulla on mahdollista luoda pelaajalle kiehtova ja arvaamaton ympäristö. [26.]

Alien: Isolationissa on kaksi erillistä tekoälyä. Toinen seuraa esimerkiksi pelaajan liikkeitä ja toinen kontrolloi vihollisen käyttäytymistä. Ohjaajatekoäly (engl. director AI) on passiivinen ohjain, joka on vastuussa pelikokemuksen luomisesta. Ohjaaja tietää jatkuvasti sekä pelaajan että vihollisen sijainnin jakamatta kuitenkaan pelaajan tarkkaa sijaintia viholliselle. Ohjaajatekoäly hallinnoi myös niin sanottua uhkamittaria, joka on lähinnä pelaajan odotettavissa olevien stressitasojen määrä. Se koostuu useista eri tekijöistä, kuten vihollisen lähistöllä tai sen näkökentässä vietetystä ajasta. Kun uhkamittarin lukema nousee tietylle tasolle, ohjaajatekoäly antaa viholliselle vihjeen pelaajan sijainnista osoittaen sen oikeaan suuntaan. [26.]

Sen lisäksi, että vihollisen tekoäly (engl. alien AI) saa ohjaajatekoälyltä tietoa uhkatasosta, se myös reagoi pelaajan käyttäytymiseen. Vihollisen tekoälyjärjestelmästä löytyy myös Utility AI:ta hyödyntävä niin sanottu työjärjestelmä, joka priorisoi listalta löytyviä tehtäviä ja kertoo sijainnin, jossa ne suoritetaan. Järjestelmän perusteella vihollisella on

kaksi eri tilaa: aktiivinen ja passiivinen. Aktiivisessa tilassa vihollinen havainnoi pelaajan liikkeitä tai sijaintia ja reagoi esimerkiksi tämän aiheuttamiin ääniin. Passiivinen tila aktivoituu uhkatason ollessa tarpeeksi pitkään korkealla, jolloin vihollinen vetäytyy syrjään piileksimään. [26.]

Vihollisen tekoäly käyttää laajaa käytöspuuta, jossa on yli 100 solmua ja 30 valintasolmua. Valintasolmut tekevät päätöksen siitä, mitä käyttäytymistä kulloinkin suoritetaan. Osa käytöspuun solmuista on pelin alkaessa lukittuna. Kun ne pikkuhiljaa aukenevat pelin edetessä, saadaan luotua illuusio oppivasta, pelaajan käytökseen sopeutuvasta tekoälystä. Lisäksi vihollinen käyttää reitinhakutekniikoita havaitakseen erilaisia pelaajan aiheuttamia ääniä. Tähän tapaan toteutetulla tekoälyllä saadaan pidettyä pelaaja jatkuvassa jännityksessä. [26.]

Kuva 10. Tilanne pelistä Alien: Isolation. [26.]

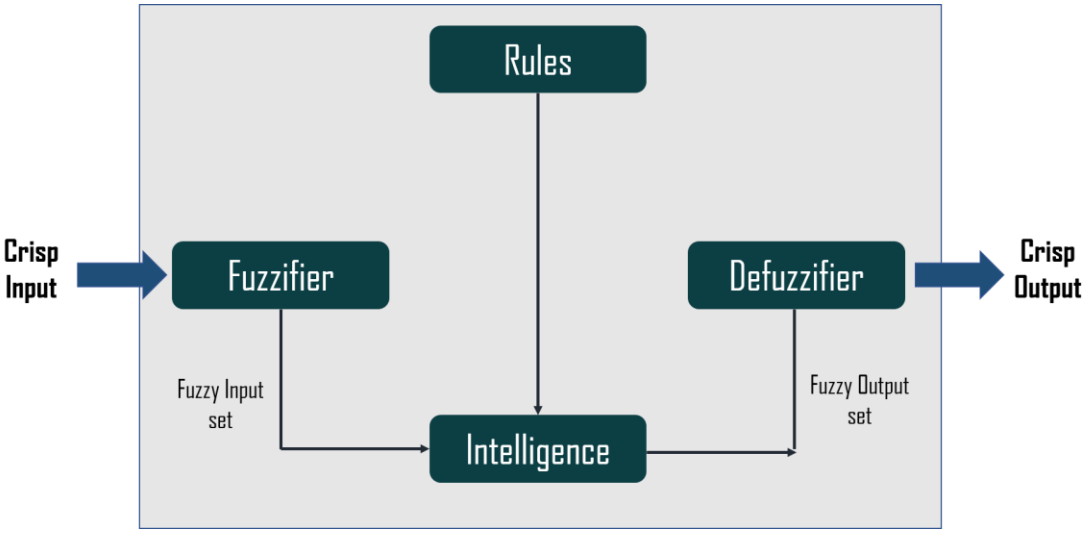
Kuvassa 10 näkyy pelaaja sekä Alien, pelaajaa etsivä vihollinen. Kuvan tilanteessa viholliselta voisi vielä olla mahdollista piiloutua, kunhan sen tekee tarpeeksi äänettömästi.

Sumea logiikka

Sumea logiikka (engl. fuzzy logic) on myös yksi tapa toteuttaa esimerkiksi NPC:n käyttäytymistä. Se on päättelymenetelmä epätarkan tiedon käsittelemiseen eli muistuttaa ihmisen tapaa päätellä asioita. Sumeaa logiikkaa soveltamalla pelihahmoista voi siis tehdä realistisempia ja enemmän ihmisenkaltaisia. [27.]

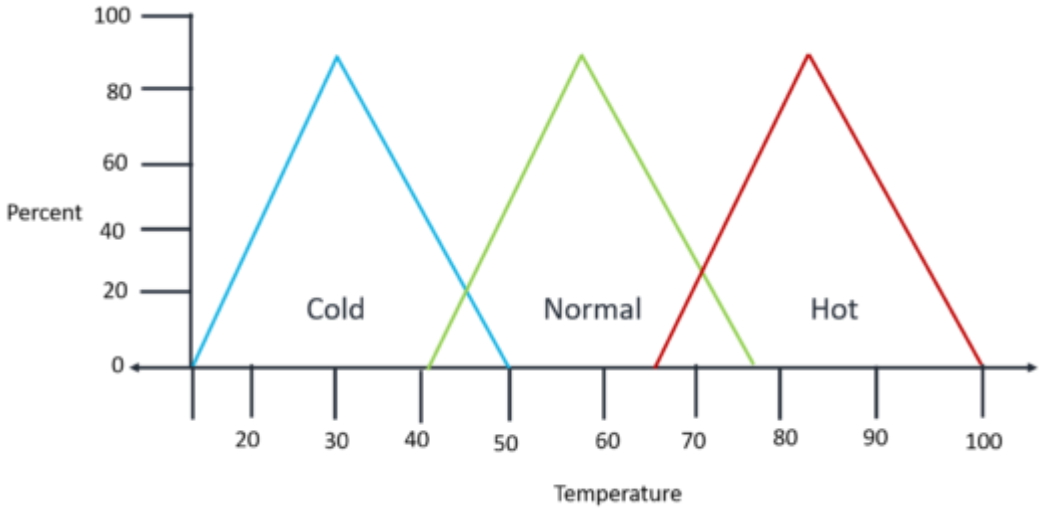
Perinteisessä tarkassa logiikassa totuusarvomuuttujan (engl. boolean) arvo vastaukselle kysymykseen ”onko lämmin?” voi olla joko tosi tai epätosi. Sumeassa logiikassa vastaus voi kuitenkin olla kaikkea näiden väliltä, kuten ”erittäin kylmä”, ”hieman kylmä”, ”kohtalaisen kuuma” tai ”en osaa sanoa”. Käytännössä sumean logiikan tarkoituksena on määrittää totuuden aste. [27.]

Kuva 11 (alla) havainnollistaa sumean logiikan toimintaa. Ensin on määritelty säännöt, joiden perusteella päätökset tehdään. Sumentamisessa (engl. fuzzification) syötteet eli tarkat (engl. crisp) arvot muunnetaan sumeiksi joukoiksi. Päätöksenteko tapahtuu päättelymoottorissa (engl. fuzzy inference engine) muuttujien arvojen perusteella. Ulostulona saadaan myös sumeita joukkoja. Ne täsmällistetään (engl. defuzzification) lopuksi tarkoiksi arvoiksi, jotta tietokone voi käsitellä ne. [27.]



Kuva 11. Sumean logiikan vaiheet. [27.]

Sumean logiikan muuttuja voi kuulua yhteen tai useampaan sumeaan joukkoon samanaikaisesti. Jäsenyyden astetta voidaan kuvata jäsenyysfunktion avulla. Kuvassa 12 (alla) näkyy esimerkki jäsenyysfunktioista. Kuvaajista nähdään, että lämpötila voi olla paitsi kylmä, kuuma tai normaali, myös asettua johonkin näiden arvojen välimaastoon, eli kuulua kahteen joukkoon samanaikaisesti. [27.]



Kuva 12. Jäsenyysfunktiot, jotka kuvaavat kylmää, normaalia ja kuumaa lämpötilaa. [27.]

Sumeaa logiikkaa voisi hyödyntää esimerkiksi tilakoneen laajentamisessa sumentamalla sääntöjä, joiden perusteella tilaa vaihdetaan, tai itse tilojen toimintalogiikkaa. Vihollinen saattaisi vaikkapa havaita pelaajan kuuntelemalla tämän aiheuttamia ääniä, ja lähteä etsimään pelaajaa, kun tämä on aiheuttanut tarpeeksi ääntä. Jahtaustilassa vihollisen nopeus muuttuisi riippuen pelaajan etäisyydestä. Hyökkäystilassa voisi olosuhteista riippuen tapahtua useita erilaisia toimintoja, kuten kasvava aggression määrä sen mukaan, mitä enemmän vihollista vahingoitetaan. Aggression määrän voisi jakaa esimerkiksi kolmeen eri sumeaan joukkoon: ärtynyt, kiukkuinen ja raivokas, joille määritetään jäsenyysfunktion avulla jäsenyysaste. Kiukkuisuuden saavuttaessa tietyn asteen, vihollinen yrittäisi tainnuttaa pelaajan, ja kiukkuisuuden huipun saavuttaessaan paiskata pelaajan kauemmas itsestään. Kun taas saavutetaan tarpeeksi suuri aggression taso, se voisi kutsua apujoukkoja. Lisäksi vihollinen voisi päättää lähteä pakenemaan, kun sen elämäpisteet ovat laskeneet alle tietyn rajan.

3.3 Tekoälyn tulevaisuus pelikehityksessä

Tekoälyn kehittymisen myötä koneoppiminen on hyvin todennäköisesti yhä enemmän keskiössä tulevaisuuden pelikehityksessä. Etenkin pelin kehitysvaiheessa tekoälyn avulla voisi olla mahdollista automatisoida suurikin osa vaadittavasta työstä, kuten NPC-hahmojen ja muiden objektien luomisprosessi ja yksilöllisen peliympäristön rakentaminen. [28.] Vahvistusoppimista käyttämällä tekoälyä voidaan palkita toivotusta käytöksestä, jolloin se oppii oikean tavan ja järjestyksen suorittaa tietyt toiminnot. Tätä voitaisiin periaatteessa hyödyntää minkä tahansa strategiaa vaativan käyttäytymisen opetuksessa, esimerkiksi reitinhaussa tai hyökkäystilanteessa. [19.]

Koneoppimisen avulla tulevaisuudessa on todennäköisesti mahdollista luoda pelejä, jotka edetessään kehittyvät ja mukautuvat pelaajan tyyliin ja joiden laajuus, monimutkaisuus ja visuaalisuus vastaavat todellista maailmaa. NPC-hahmojen toiminta olisi vähemmän ennalta arvattavaa ja myös niiden päihittämisen vaikeusaste voisi nousta sitä mukaa, kun pelaajan taidot karttuvat. Tekoälyn huippu pelikehityksessä olisi todellinen jatkuvasti muuttuva ja kehittyvä hahmo, joka reagoisi tapahtumiin inhimillisesti. [4.]

Tulevaisuuden tekoäly voisi siis mahdollistaa sen, että pelit näyttävät paremmilta, ovat realistisempia ja lisäksi myös paljon edullisempia tuottaa. [4.]

Proseduraalinen generointi tekoälyn avulla

Useissa peleissä on jo pitkään käytetty proseduraalista generointia, eli pelisisällön automaattista tuottamista algoritmien avulla manuaalisen luomisen sijaan. Tätä hyödynnetään esimerkiksi pelikenttien luomisessa, mutta myös muun pelisisällön tuottamisessa. [17.]

Tekoälyn hyödyntäminen proseduraalisessa generoinnissa on vielä kohtalaisen tuore asia. Oppiva tekoäly voisi tulevaisuudessa kuitenkin rakentaa vaikka kokonaisia pelejä hyödyntäen muun muassa proseduraalista generointia. Jotain tämänkaltaista on jo hieman tehtykin: Nvidian kehittämä tekoäly loi PacMan-pelin uudelleen vain katsomalla sen pelaamista. [28.] Eräässä toisessa tutkimuksessa tekoäly loi käytännössä kokonaan uuden pelin, kun se katsoi Super Marion, Mega Manin ja Kirby's Adventuren pe-

laamista. Tuloksena oli Megamania muistuttava peli. Tällaisesta menetelmästä käytetään nimitystä konseptin laajennus (engl. conceptual expansion). [29.]

Kenttien ja tasojen lisäksi olisi mahdollista generoida käytännössä mitä tahansa rakennuksista vaihteleviin maastoihin ja NPC-hahmoihin. Koneoppimista voitaisiin hyödyntää esimerkiksi pelityylin tai pelaajan mieltymysten mukaisesti. Tulevaisuudessa tekoäly oppii todennäköisesti mukauttamaan peliä täysin reaaliaikaisesti. Proseduraalinen sisällöntuotanto myös säästää huomattavasti aikaa pelikehityksessä. Esimerkiksi yhden rakennuksen tekeminen peligraafikolta voi helposti viedä yli 20 tuntia. [18.]

Nvidian kehittämä tekoäly generoi peligrafiikoita hyödyntäen syväoppimista. Tekoäly käyttää pelimoottoria ja videomateriaalia opetellakseen mielikuvituksellisen kaupunkikorttelin luomiseen vaadittavat toimenpiteet. Samaa tekniikkaa voidaan käyttää myös vaikkapa täysin uusien ihmiskasvojen luomiseen. Lista toki on loputon: tekoäly voi tehdä metsiä, vuoria, eläimiä, avaruusaluksia, erilaisia rakennuksia tai mitä tahansa, mistä löytyy kuva- tai videomateriaalia. Pelisuunnittelija määrittelee vain haluamansa parametrit, kuten talon ulkomitat sekä ikkunoiden ja ovien lukumäärän. Tekoäly antaa tämän jälkeen erilaisia vaihtoehtoja, joita pystyy halutessaan muokkaamaan. [30.]

Kuva 13. Nvidian tekoälyn generoimia ihmiskasvoja. [31.]

Kuvassa 13 näkyy Nvidian neuroverkon generoimia erittäin realistisia ihmiskasvoja. Tekoäly luo täysin uusia kasvoja aidoista kuvista oppimansa tiedon perusteella.

Borderlands-peleissä käytetään proseduraalista generointia aseiden luomiseen, mikä tarkoittaa, että pelissä on lähestulkoon loputtomasti vaihtelua erilaisten aseiden suhteen. Melkein jokainen ase luodaan satunnaisesti valtavasta määrästä eri komponentteja. Myös niiden ominaisuudet luodaan täysin sattumanvaraisesti. [32.]

Edinburghin yliopiston ja Method Studiosin yhteistyössä toteuttama tekoäly on koulutettu realistisen liikkeen saloihin katsomalla videoleikkeitä liikkeenkaappauksista, joista se näkee erilaisia tapoja liikkua. Tämän perusteella se pystyy luomaan erittäin aidon näköisiä animaatioita. [33.]

4 Tekoälyn toteutus 2D-tasohyppelyssä

4.1 Pelin tavoitteet

Tarkoituksena oli tehdä retrotyylinen 2D-tasohyppelypeli, jossa pelaaja voi liikkua sekä maassa että seiniä pitkin. Peli rakennettiin käyttämällä Unity-pelimoottoria ja C#-ohjelmointikieltä.

Pelikenttien ja hahmojen luomisessa käytettiin tyyliin sopivaa pikseligrafiikkaa. Niistä osa on tehty itse Adobe Photoshopilla ja osa on valmiita assetteja, jotka on ladattu Unityn Asset Storesta. Se on Unityn oma asset-kirjasto, jossa on kehittäjien ja artistien tuottamaa ilmaista tai maksullista sisältöä, jota voi ladata omaan käyttöönsä.

Pelaajahahmo on soturi, joka pystyy tuhoamaan vihollisia joko miekalla tai hyppimällä niiden päälle. Pelissä on erilaisia NPC-hahmoja eli esimerkiksi vihollisia, joita täytyy eliminoida tai paeta päästäkseen pelissä eteenpäin. Näiden lisäksi on ystävällisiä NPC-hahmoja, jotka voivat olla vaikkapa eläimiä. Kuoltuaan NPC:t pudottavat pelaajan kerättäväksi erilaisia esineitä, joista saa esimerkiksi lisää elämäpisteitä tai hetken kestäviä erikoisvoimia. Pelin edetessä on tarkoitus, että pelaaja oppii hyödyntämään näitä optimaalisella tavalla, kuten keräämällä tiettyjä lisävoimia antavia esineitä vasta juuri ennen voimakkaan vihollisen kohtaamista. Keräiltäviä esineitä on mahdollista löytää myös esimerkiksi laatikoista ja pensaista, joita pystyy rikkomaan.

Peli muuttuu vaikeammaksi tasolta toiselle siirryttäessä, mikä tarkoittaa käytännössä sitä, että pelikentällä liikkuminen on haastavampaa ja viholliset voimakkaampia.

Käytettävät tekoälymenetelmät

Pelin tavoitteena oli erityisesti selvittää, minkälainen tekoäly tulee kyseeseen tämänkaltaisessa tasohyppelypelissä. Tekoälyä päätettiin soveltaa NPC-hahmoihin, jotta niistä välittyisi pelaajalle aidompi tunne.

Ensimmäinen toteutustapa on itsenäisesti liikkuva hahmo, joka kulkee tietyn alueen sisällä välillä pysähtyen sekä tunnistaen mahdolliset reunat ja muut esteet. Hahmosta on kaksi eri versiota: vihollis-NPC ja ystävällinen NPC. Ensin mainittu eroaa toisesta siten, että se hyökkää pelaajan kimppuun, kun pelaaja on tarpeeksi lähellä. Lisäksi sillä on enemmän elämäpisteitä ja toiminnallisuuksia.

Toinen suunniteltu tekoälyn sovellus on FSM eli rajallinen tilakone, joka on mahdollista toteuttaa useillakin vaihtoehtoisilla tavoilla. Yksinkertaisimmillaan tilakone voidaan rakentaa pelkillä if-lauseilla tai käyttäen enumia ja switch-rakennetta. Jopa Unityn animaattori voi käytännössä toimia tilakoneena tai käytöspuuna, kun tiettyä tilaa vastaavaan animaatioon lisätään erilaisia käyttäytymisiä. Monimutkaisempi versio onnistuu esimerkiksi hyödyntämällä periytymistä tai rajapintoja. Tässä tapauksessa päädyttiin kokeilemaan tilakoneen toimintaa kahdella eri tavalla: enumilla ja periytymisellä.

Kolmanneksi tekniikaksi valikoitui A*-reitinhakualgoritmi, joka on tarkoitus toteuttaa ainakin lentävään NPC-hahmoon. Tällöin ei ole tarvetta huomioida esimerkiksi painovoiman vaikutuksia.

4.2 Pelin toteutus

Pelikenttä luotiin Unityn Tilemap-komponentilla. Liikkumisen toteutuksen helpottamiseksi pinnasta päätettiin tehdä täysin tasainen. Liikkuminen tällaisessa pelissä myös näyttää paremmalta tasaisella kuin epätasaisella alustalla. Tilemap-komponentti jaettiin useaan kerrokseen. Itse liikkumistasot (engl. platform) ovat tasaisia, mutta etu- ja taka-alalla on lisäksi maastoon vaihtelevuutta tuovia elementtejä, joissa ei ole törmäyksentunnistusta. Myös taustat on jaettu useaan kerrokseen syvyysvaikutelman aikaansaa-

miseksi. Kamerakomponenttia ohjaava skripti liikuttaa niitä eri nopeuksilla eli parallaksia vieritystä (engl. parallax scrolling) toteuttaen.

Pelaajahahmon toimintoja ohjaa skripti, joka vastaa esimerkiksi liikkumisesta. Pelaaja voi juosta, hyppiä sekä ilmassa että seiniä pitkin, syöksyä nopeasti eteenpäin tietyin väliajoin ja lisäksi liukua alas seinää pitkin. Sekä pelaaja- että NPC-hahmojen liikkumisessa on hyödynnetty Unityn Rigidbody 2D -komponenttia, joka mahdollistaa Unityn fysiikkamoottorin ominaisuudet pelihahmolle. Tällöin liikkuminen näyttää realistiselta.

Kaikki pelissä käytettävät grafiikat ovat spritejä eli 2D-grafiikkaobjekteja. Pelaaja- ja NPC-hahmoille on myös luotu useita eri animaatioita, kuten juokseminen, hyppääminen ja hyökkäys.

Pelissä on tarkistuspisteitä (engl. checkpoint), jotka toimivat myös uudelleensyntymispisteinä (engl. spawning point) pelaajan kuollessa. Kuoltuaan pelaaja ilmestyy aina sen tarkistuspisteen kohdalle, jonka on viimeksi ohittanut.

Kuva 14. Näkymä pelistä.

Kuvassa 14 on näkymä pelikentästä. Pelaajahahmo on keskellä, ja liikkumistasoja on tässä kohtaa usealla eri korkeudella.

4.3 Yksinkertainen tekoälyn toteutus NPC-hahmolle

Ajatus NPC-hahmolla olevasta tekoälystä voi olla jo pelkästään se, että hahmo liikkuu itsenäisesti rajatulla alueella mahdollisia esteitä väistellen. Yksinkertaisimmillaan tämä voidaan toteuttaa tekemällä kaksi tai useampia liikkumispisteitä. Liikkumispisteet ovat peliobjekteja, joiden välille NPC:n liikkuminen määritetään skriptissä. Esimerkiksi lentävän vihollisen voi ohjelmoida liikkumaan useiden pisteiden välillä ja eri korkeuksilla.

Kuva 15. NPC liikkuu kahden ennalta määritetyn pisteen välillä.

Kuvassa 15 nähdään, miten liikkumispisteet sijaitsevat pelikentällä. NPC on ohjelmoitu liikkumaan niiden välisellä alueella pysähtyen välillä hetkeksi. Aina pisteelle päästyään se kääntyy ympäri ja lähtee kulkemaan vastakkaiseen suuntaan.

Liikkumispisteiden sijaan tai lisäksi liikkumiseen saadaan luotua adaptiivisuutta käyttämällä Raycastia tai Linecastia. Molemmat ovat Unityn Physics-luokasta löytyviä metodeja, joita käytetään törmäysten havaitsemiseen fyysisten objektien välillä. Periaatteessa niitä voisi ajatella vaikkapa NPC:n silminä, koska niiden avulla NPC saadaan huomaamaan esimerkiksi alueen reunat ja seinät. Tämä toimii hyvin ilmassa leijuvalla platformilla tai kahden seinän välisellä alueella. Tällöin ei tarvitse erikseen edes määritellä liikkumispisteitä, koska NPC voidaan ohjelmoida kääntymään aina toiseen suuntaan esteen havaitessaan.

Raycast ”ampuu” säteen lähtöpisteestä annettuun suuntaan. Oletusarvoisesti säteen pituus on ääretön, mutta sitä voi myös halutessaan muuttaa. Säde havaitsee peliobjektit, joihin on liitetty Collider-komponentti törmäyksentunnistusta varten. Jos halutaan havaita useita collidereita säteen varrella, voidaan käyttää RaycastAll()-funktiota.

Linecast on hyvin samankaltainen Raycastin kanssa. Vähäinen ero löytyy määrittelystä: Raycastissa määritellään alkupiste ja suunta, kun taas Linecastissa määritellään alku- ja loppupiste. Linecast tutkii, onko näiden kahden pisteen välillä fyysistä objektia.

Myös pelaajan havaitsemiseen voidaan käyttää Raycastia. Sen avulla on helppoa toteuttaa NPC, jolta voi vaikkapa piiloutua erilaisten, ”näköesteinä” toimivien objektien taakse. Raycast siis osuu tähän NPC:n ja pelaajan välillä sijaitsevaan objektiin pelaajan sijaan. Tämä on myös yksi tapa toteuttaa esimerkiksi hyökkäysetäisyys eli alue, jonka sisällä NPC voi havaita pelaajan. Raycastilla voidaan määrittää, miten kaukaa NPC pelaajan havaitsee. Näin NPC voidaan ohjelmoida jahtaamaan pelaajaa niin kauan kuin sen ja pelaajan välinen etäisyys ei ylitä määritettyä raja-arvoa.

Pelaajan havaitsemisalue voidaan toteuttaa myös esimerkiksi collider-komponentin avulla. Komponentissa on asetus ”is trigger”, jolla voidaan määritellä, käyttäytyykö se tavallisena colliderina vai jonkin toiminnan laukaisevana tekijänä. Kun pelaaja astuu alueelle, NPC:n skriptiin sisältyvä OnTriggerEnter2D()-funktio saa siitä tiedon. Silloin voidaan suorittaa halutut toimenpiteet kuten hyökkäys pelaajaa kohti.

Kuva 16. Havaitsemisalue

Kuvassa 16 näkyvä vihreä laatikko on alue, jonka sisällä NPC havaitsee pelaajan (Aggro Area). Se on peliobjekti, johon on liitetty Box Collider 2D -komponentti törmäyksentunnistukseen. Sen voi säätää juuri tarvittavan kokoiseksi.

NPC:lle voidaan myös määritellä satunnainen aika liikkumista ja paikallaanoloa varten, ja se vaihtelee näiden kahden tilan välillä. Näitä kaikkia menetelmiä yhdistelemällä on hahmoon saatu luotua hieman eloa ja tekoälyn tuntua, kuitenkin käyttämättä varsinaisia tekoälyalgoritmeja.

```
groundDetected = Physics2D.Raycast(groundCheck.position, Vector2.down, ground-
CheckDistance, whatIsGround);

wallDetected = Physics2D.Raycast(wallCheck.position, transform.right, wallCheck-
Distance, whatIsGround);
```

Esimerkkikoodi 1. Raycast maan ja seinien havaitsemiseen.

Esimerkkikoodissa 1 määritetään Raycast muuttujien groundDetected ja wallDetected arvoiksi. Ensimmäinen tarkkailee, onko vastassa vielä maata, jotta tiedetään kääntyä reunan tullessa vastaan. Toinen tekee saman seinien suhteen. Molemmissa annetaan parametreina sijainti eli lähtöpiste, suunta, välimatka ja LayerMask. Unityn jokaisessa peliobjektissa voidaan määritellä, mihin layeriin eli kerrokseen se kuuluu. LayerMask-parametrin avulla voidaan kertoa Raycastille, mitä layereita kyseisessä tilanteessa tulee käyttää. WhatIsGround-muuttujan avulla määritellään siis, mikä tai mitkä peliobjektit ovat maata. GroundCheck ja wallCheck ovat Unityssa luodut peliobjektit, jotka sijoitetaan sopiviin kohtiin havainnoinnin suorittamista varten (ks. kuva 17).

Kuva 17. Raycastien havainnollistamista Unityssa.

Kuvassa 17 näkyvät seinien ja maan havaitsevien Raycastien sijainnit. Unityn gizmot ovat työkaluja visuaaliseen vian- tai asetusten määritykseen. Niiden avulla näiden muuttujien arvoja ja peliobjektien sijaintia on helppo säätää sopivaksi Unityn puolella, kun ne voi nähdä konkreettisesti. Kuvassa näkyvä pystyviiva havaitsee maan ja NPC:n edessä oleva vaakaviiva puolestaan seinät. GroundCheckDistance ja wallCheckDistance -muuttujissa määritellään näiden pituus.

4.4 Äärellisen tilakoneen toteutus enum-luokalla

Yksinkertainen tilakone voidaan toteuttaa esimerkiksi enum-luokkaa ja switch-rakennetta käyttämällä. Kun muuttujien arvot tiedetään ennalta, voidaan käyttää enum-luokkaa eli lueteltua tyyppiä. Se on rajapinnan ja normaalin luokan lisäksi oma luokkatyyppinsä, joka määritellään avainsanalla enum (enumeration). Oletusarvoisesti enumin arvot ovat int-tyyppiä.

```
private enum npcStates
    {
        Idle,
        Patrol,
        Attack,
        Dead
    }

    private npcStates currentState;
```

Esimerkkikoodi 2. Enum-arvojen määrittely

Esimerkkikoodissa 2 määritetään enumin arvot jokaista haluttua NPC:n tilaa kohden ja npcStates-tyyppinen muuttuja currentState pitämään kirjaa kulloinkin käynnissä olevasta tilasta.

```
private void Update()
    {
        switch (currentState)
        {
            case npcStates.Idle:
                IdleState();
                break;
            case npcStates.Patrol:
                PatrolState();
                break;
            case npcStates.Attack:
                AttackState();
                break;
            case npcStates.Dead:
                DeadState();
                break;
        }
    }
```

Esimerkkikoodi 3. Switch-case-rakenne

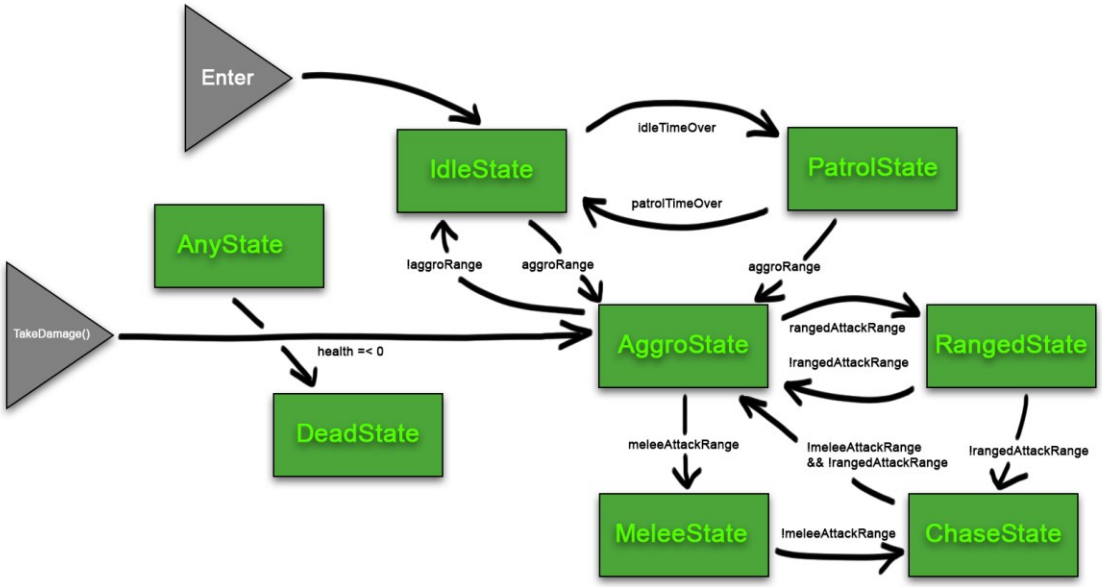
Esimerkkikoodissa 3 näkyy Update()-funktio, jossa käytetään if-lauseiden sijaan switch-case-lausetta. Se sisältää neljä ”casea”, yksi kutakin enumin arvoa varten. Jokaisessa casessa kutsutaan haluttua funktiota kyseistä tilaa vastaavalle käyttäytymiselle. Itse tilan vaihtaminen tapahtuu näiden funktioiden sisällä.

4.5 Äärellisen tilakoneen toteutus periytymisellä

Jos pelissä ei ole kovinkaan useita erilaisia NPC-hahmoja, eikä niillä myöskään ole lukuisia määriä erilaisia tiloja, tilakoneen voi vielä hyvin toteuttaa pelkillä ehtolauseilla yhdessä ainoassa skriptissä. Tällaisesta ratkaisusta voi kuitenkin tulla hankala ja epä-

selvä, jos peliin lisätään monia, eri tavoilla käyttäytyviä hahmoja. Lopputuloksena yksittäisetkin skriptit voivat paisua satojen rivien pituisiksi.

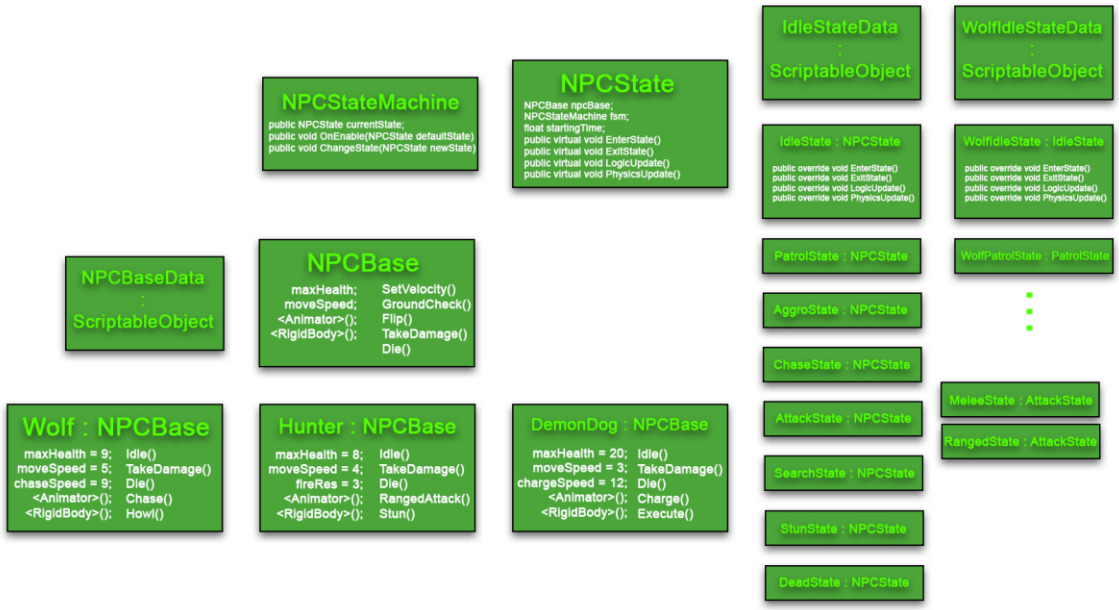
Yksi huomattavasti selkeämpi tapa on käyttää periytymistä (engl. inheritance). Sen avulla voidaan luoda niin sanottu kantaluokka (engl. base class), josta voi periytyä useita eri aliluokkia (engl. derived class). Kantaluokassa määritellään kaikki yhteiset muuttujat ja funktiot, jotka myös periytyvät luokat saavat käyttöönsä. Kantaluokan funktion määrittelyssä käytetään avainsanaa ”virtual”, jolloin niiden toimintoja voidaan tarvittaessa muuttaa tai laajentaa ylikirjoittamalla niitä aliluokissa. Olio-ohjelmoinnissa tätä kutsutaan polymorfismiksi. [34.]



Kuva 18. Tilakaavio

Kuvassa 18 näkyy yhden pelissä käytettävän NPC:n kaikki tilat sekä ehdot niiden välisille siirtymisille. Oletustilana on IdleState, jolloin NPC pysyy paikallaan. Kun tiettyjen raja-arvojen sisään määritelty satunnaisaika on ohi, NPC siirtyy PatrolState-tilaan, jolloin se liikkuu. Molemmista tiloista voi tapahtua siirtyminen AggroState-tilaan, mikäli pelaaja tulee tarpeeksi lähelle. RangedState on tarkoitettu pitkän, ja MeleeState lyhyen välimatkan hyökkäyksille. Ne molemmat periytyvät kantaluokasta AttackState. Myös näihin tiloihin siirrytään tiettyjen etäisyyksien perusteella. Jos ei olla hyökkäysetäisyydellä, siirrytään ChaseState-tilaan jahtaamaan pelaajaa. Siirtyminen AnyState-tilasta

DeadState-tilaan tarkoittaa, että vihollinen voi kuolla missä tahansa tilassa. Tällöin kuolemiseen johtavia ehtoja ei tarvitse määritellä jokaisessa tilassa erikseen.



Kuva 19. Luokkakaavio

Kuva 19 havainnollistaa eri NPC-luokkia ja periytymistä niiden välillä. Pelissä voi olla siis useita erilaisia vihollisia, jotka kaikki periytyvät samasta kantaluokasta NPCBase. Kantaluokassa määritellään tietyt yhteiset ominaisuudet, ja jokainen periytyvä luokka saa ne käyttöönsä. Lisäksi yksittäisillä NPC-luokilla voi olla ominaisuuksia tai toimintoja, jotka eroavat täysin muista luokista. Kaikki eri tilaluokat periytyvät myös yhdestä kantaluokasta.

Periytyminen toteutettiin tekemällä ensin skriptit NPCStateMachine, NPCState ja NPCBase. NPCStateMachine pitää kirjaa siitä, missä tilassa NPC on milloinkin. NPCState-luokassa on kaikille tiloille yhteiset muuttujat ja funktiot. Funktiot on määritelty virtuaalisiksi, jotta niitä on mahdollista ylikirjoittaa ja siten toteuttaa yksilöllinen toiminnallisuus aliluokissa. NPCBase-luokka on kantaluokka kaikille NPC-hahmoille, ja myös se sisältää kaikki yhteiset muuttujat ja funktiot, jotka tästä luokasta periytyvät eri NPC-luokat saavat käyttöönsä. Kaikki eri tilaluokat taas periytyvät NPCState-luokasta.

```
public class NPCState
{
    protected NPCStateMachine fsm;
    protected NPCBase npcBase;
```

```
    public float startingTime { get; protected set; }
    protected string animBool;

    public NPCState(NPCBase enemyBase, NPCStateMachine fsm, string animBool) {
        this.npcBase = enemyBase;
        this.fsm = fsm;
        this.animBool = animBool;
    }

    public virtual void EnterState() {
        startingTime = Time.time;
        npcBase.anim.SetBool(animBool, true);
    }
    public virtual void ExitState() {
        npcBase.anim.SetBool(animBool, false);
    }
    public virtual void LogicUpdate() {
    }
    public virtual void PhysicsUpdate() {
    }
}
```

Esimerkkikoodi 4. NPCState-luokka

Esimerkkikoodissa 4 näkyy NPCState-luokan rakennetta. Ensimmäisenä halutaan pitää kirjaa tilakoneesta, jolle NPCState kuuluu. Tässä on siis määritelty NPCStateMachine-tyyppinen muuttuja, joka on muotoa protected. Käytännössä se toimii kuten private, mutta periytyvillä luokilla on siihen kuitenkin vapaa pääsy. Myös NPCBase-luokkaan halutaan pääsy, joten sitä varten on määritelty muuttuja npcBase. AnimBool-muuttuja on tilaa vastaavan boolean-tyyppiä olevan muuttujan nimi, jotta jokainen tila pystyy automaattisesti asettamaan kulloinkin tarvittavat animaatioparametrit. Vihollisluokassa määritetyn animaation nimi on oltava täsmälleen sama kuin sitä vastaavan boolean-muuttujan nimi Unityn animaattorissa. Lisäksi NPCState-luokalle tarvitaan konstruktori, jossa määritellään kaikki parametrit, jotka uudelle tilaoliolle pitää antaa sitä luodessa.

Tämän jälkeen on määritelty EnterState(), ExitState(), LogicUpdate() ja PhysicsUpdate(), jotka ovat siis kaikille tiloille yhteisiä funktioita. NPCState-luokassa näihin funktioihin ei juurikaan laiteta toimintoja, koska ideana on toteuttaa yksilöllinen käyttäytyminen jokaisessa NPC-aliluokassa. Se mistä kuitenkin halutaan aina olla selvillä, on tilan alkamisaika. Sitä sitä varten on luotu muuttuja startingTime, jonka arvo asetetaan EnterState()-funktiossa. Näin ollen aina, kun kyseistä funktiota kutsutaan tilasta, jossa kulloinkin ollaan, se tallentaa kyseisen tilan alkamisajankohdan. Mistä tahansa tilasta voidaan siis tarvittaessa viitata tähän muuttujaan ilman, että alkamisaikaa tarvitsee asettaa jokaisessa tilassa erikseen.

```
public class NPCStateMachine
{
    public NPCState currentState { get; private set; }

    public void OnEnable(NPCState defaultState) {
        currentState = defaultState;
        currentState.EnterState();
    }

    public void ChangeState(NPCState newState) {
        currentState.ExitState();
        currentState = newState;
        currentState.EnterState();
    }
}
```

Esimerkkikoodi 5. NPCStateMachine-luokka

Esimerkkikoodissa 5 näkyy NPCStateMachine-luokka, jossa on ensin määritelty NPCState-tyyppinen muuttuja currentState, joka pitää kirjaa kulloinkin käynnissä olevasta tilasta. OnEnable-funktio saa parametrinaan tilan, joka asetetaan samalla NPC:n oletustilaksi. Tämän jälkeen kutsutaan NPCState-luokan EnterState()-funktiota. ChangeState()-funktio hoitaa tilasta toiseen siirtymisen, eli se saa parametrinaan uuden tilan. Funktiossa kutsutaan ensin parhaillaan aktiivisena olevan tilan ExitState()-funktiota. Sitten currentState-muuttuja saa arvokseen tilan, johon ollaan vaihtamassa. Lopuksi kutsutaan uuden tilan EnterState()-funktiota.

NPCBase-luokassa pitää määritellä, mitä kaikkea yhteistä halutaan kaikilla NPC-hahmoilla olevan. Yhteistä NPC-hahmoilla voivat olla esimerkiksi erilaiset Unityn komponentit, kuten animaattori ja Rigidbody. Myös mahdollinen erillinen grafiikkaobjekti voi löytyä kaikilta NPC-hahmoilta. Muita yhteisiä muuttujia voisivat olla esimerkiksi jäljellä olevien elämäpisteiden määrä tai erilaiset resistanssit. Lisäksi takaa hyökättäessä tai kulkusuunnan vaihtuessa halutaan pyöräyttää NPC ympäri x-akselilla. Silloin tarvitaan katsomissuuntamuuttujaa, joka myös voidaan tarvittaessa asettaa yhteiseksi NPC-hahmojen välille NPCBase-luokassa.

NPCBase-luokkaan on myös tehty funktiot kaikkien hahmojen yhteisille toiminnoille. Nekin on määritelty virtuaalisiksi, jotta niihin on mahdollista tehdä muutoksia aliluokista. Start()- ja Update()-funktioiden lisäksi oli tarpeellista tehdä funktiot myös esimerkiksi liikkumisnopeuden asettamista varten, monenlaisille tarkistuksille ympäristön havainnointia varten, DealDamage()- ja TakeDamage()-funktiot pelaajahahmon vahingoittamista ja vihollisen vahingoittumista varten, Flip()-funktio peliobjektin kääntämistä varten sekä OnDrawGizmos()-funktio apuviivojen piirtämistä varten, jotta Unityssa voidaan

havainnoida visuaalisesti objektien sijaintia ja etäisyyksiä toisistaan sekä asettaa niille helpommin halutut arvot.

```
public NPCStateMachine fsm;

public virtual void Start() {
        currentHealth = baseData.maxHealth;
        facingDirection = 1;

        rb = GetComponent<Rigidbody2D>();
        anim = GetComponent<Animator>();

        fsm = new NPCStateMachine();
    }
```

Esimerkkikoodi 6. NPCBase-luokka

Esimerkkikoodissa 6 nähdään oman tilakoneen luonti jokaiselle NPC:lle. Ensin on luotu NPCStateMachine-tyyppinen instanssimuuttuja ja Start()-funktiossa luodaan viimeisenä uusi tilakone jokaiselle NPC:lle.

Tämän jälkeen tehtiin tilaluokat. Niille luotiin lisäksi dataluokat, jotka ovat skriptattavia objekteja. Skriptattavat objektit ovat käytännössä datasäiliöitä, jotka voivat pitää sisällään suuria määriä dataa ja ovat itsenäisiä luokkien instansseihin nähden. Esimerkiksi tiloilla on erilaisia muuttujia muun muassa paikallaanoloajan, liikkumisnopeuden ja etäisyyksien määrittämistä varten. Itse tilat ovat vain olioita, joten niiden muuttujien arvoja ei voi asettaa Unityn puolella. Dataluokat siis pitävät sisällään nämä muuttujat eri luokille, ja ne kaikki ovat julkisia. Varsinaiset dataobjektit luodaan Unityn puolella.

```
public class ChaseState : NPCState
{
    protected ChaseStateData npcStateData;

    public ChaseState(NPCBase npc, NPCStateMachine stateMachine, string anim-
Bool, ChaseStateData npcStateData) : base(npc, stateMachine, animBool)
    {
        this.npcStateData = npcStateData;
    }

    public override void CheckSurroundings() {
        base.CheckSurroundings();
    }

    public override void EnterState() {
        base.EnterState();
    }

    public override void ExitState() {
        base.ExitState();
    }

    public override void LogicUpdate() {
```

```
        base.LogicUpdate();
    }

    public override void PhysicsUpdate() {
        base.PhysicsUpdate();
    }
}
```

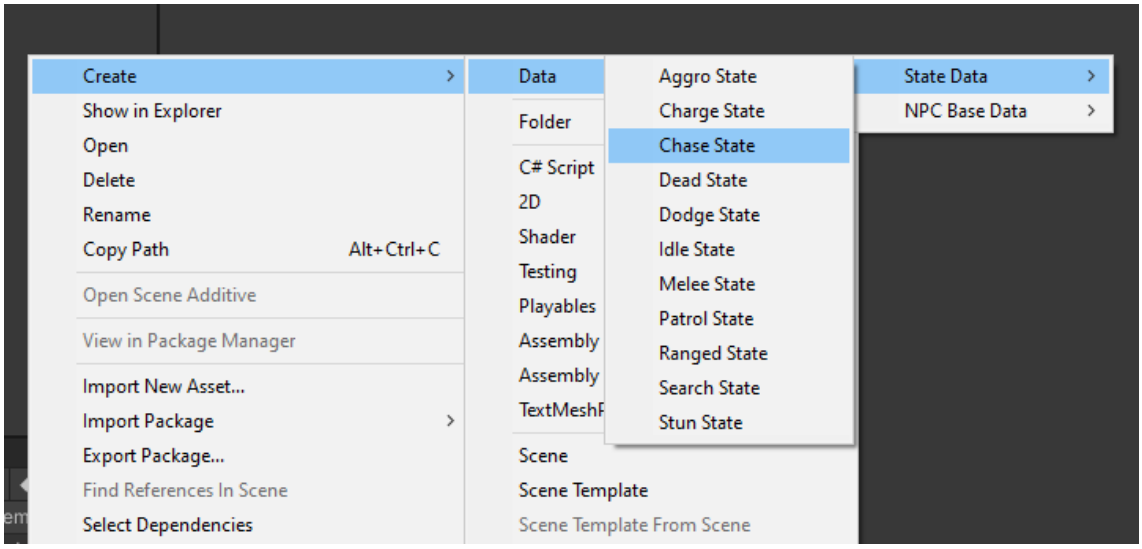
Esimerkkikoodi 7. Esimerkki tilaluokasta ChaseState.

Esimerkkikoodissa 7 näkyy ChaseState-luokka. Se on yksi tilaluokista ja periytyy NPCState-luokasta. Tilaan pitää määritellä dataluokan muuttuja npcStateData, jotta kaikki data saadaan käyttöön. Lisäksi tarvitaan konstruktori sekä ylikirjoitusfunktiot ainakin niihin kantaluokan funktioihin, joihin aiotaan tehdä muutoksia.

```
[CreateAssetMenu(fileName = "newChaseStateData", menuName = "Data/State Da-
ta/Chase State")]
public class ChaseStateData : ScriptableObject
{
    public float chaseSpeed;
    public GameObject target;
}
```

Esimerkkikoodi 8. ChaseStaten dataluokka.

Esimerkkikoodissa 8 on ChaseState-luokan dataluokka. Se periytyy ScriptableObjectista. Luokan yläpuolelle on kirjoitettu määrittely CreateAssetMenu, jotta dataobjektin luominen onnistuu Unityssa.



Kuva 20. Dataobjektien luominen

Kuvassa 20 nähdään, miten dataobjektin saa luotua Unityssa.

Tilaluokista ensimmäisenä tehtiin pelaajan jahtaamista varten ChaseState-luokka, liikkumista varten PatrolState-luokka ja paikallaanoloa varten IdleState-luokka sekä näille kaikille omat dataluokat. Myös NPCBase-luokalle tehtiin oma dataluokka, jossa on määritelty muuttujat muun muassa elämän määrälle, pelaajan, maan, seinien ja reunojen havaitsemisetäisyyksille sekä NPC:n vahingoittamisessa näkyvälle efektille. Näitä arvoja voi Unityn puolella säätää jokaisen NPC:n kohdalla yksilöllisesti.

Myös hierarkia astuu mukaan AttackState-luokassa eli hyökkäystilassa. Pelissä on kaksi tapaa hyökätä: pitkän (Ranged) ja lyhyen (Melee) etäisyyden hyökkäykset. Sekä RangedAttackState että MeleeAttackState periytyvät AttackState-luokasta, joka taas periytyy NPCState-luokasta. Myös kaikkiin tilaluokkiin tehtiin konstruktori tarvittavilla parametreilla, jotka viedään NPCState-luokalle.

Tilaluokkiin on kirjoitettu ylikirjoitusfunktiot, jotta voidaan muuttaa tai lisätä niiden toimintoja. Esimerkiksi PatrolState-luokan EnterState()-funktiossa asetetaan NPC:n nopeus ja tilan kesto, joka on satunnainen aika tilan dataluokassa määritetyn minimi- ja maksimiajan välillä. LogicUpdate()-funktiossa verrataan tätä aikaa tilan alkamisaikaan ja päätetään sen perusteella, onko aika päättynyt. Itse tilan vaihtamisen ehdot määritellään yksittäisten NPC-hahmojen omissa tilaluokissa. Siten useilla eri NPC-hahmoilla voi olla erilaisia tiloja ilman, että niillä kaikilla täytyy olla kaikki samat tilat. Kaikille mahdollisille tiloille siis tehdään omat tilaluokat, mutta näistä periytyvät yksittäisen NPC-hahmon tilaluokat tehdään vain sen tarvittaville tiloille.

```
public override void LogicUpdate()
    {
        base.LogicUpdate();

        if (minAggroDistance) {
            fsm.ChangeState(wolf.chaseState);
        }
        else {
            fsm.ChangeState(wolf.searchState);
        }
    }
```

Esimerkkikoodi 9. NPC:n AggroState-luokka

Esimerkkikoodissa 9 näkyy yksittäisen NPC:n AggroState-luokka eli pelaajan havaitsemistila. Tilanvaihto tapahtuu LogicUpdate()-funktiossa. Jos ollaan havaitsemisetäi-

syydellä, vaihdetaan jahtaustilaan (chaseState), muussa tapauksessa etsimistilaan (searchState).

```
public class Wolf : NPCBase
{
    [Header("State Data")]
    [SerializeField]
    private IdleStateData idleStateData;
    [SerializeField]
    private PatrolStateData patrolStateData;
    [SerializeField]
    private AggroStateData aggroStateData;
    [SerializeField]
    private ChaseStateData chaseStateData;
    [SerializeField]
    private MeleeStateData attackStateData;
    [SerializeField]
    private DeadStateData deadStateData;

    [Header("Other")]
    [SerializeField]
    private Transform meleeAttackPosition;

    public Wolf_IdleState idleState { get; private set; }
    public Wolf_PatrolState patrolState { get; private set; }
    public Wolf_AggroState aggroState { get; private set; }
    public Wolf_ChaseState chaseState { get; private set; }
    public Wolf_AttackState attackState { get; private set; }
    public Wolf_DeadState deadState { get; private set; }


    public override void Start()
    {
        base.Start();

        idleState = new Wolf_IdleState(this, fsm, "idle", idleStateData, this);
        patrolState = new Wolf_PatrolState(this, fsm, "patrol", patrolStateData,
this);
        aggroState = new Wolf_AggroState(this, fsm, "aggro", aggroStateData,
this);
        chaseState = new Wolf_ChaseState(this, fsm, "chase", chaseStateData,
this);
        attackState = new Wolf_AttackState(this, fsm, "attack", meleeAttackPosi-
tion, attackStateData, this);
        deadState = new Wolf_DeadState(this, fsm, "dead", deadStateData, this);

        fsm.OnEnable(patrolState);
    }
 }
```

Esimerkkikoodi 10. Yksittäisen NPC:n luokka

Esimerkkikoodissa 10 näkyy jokaista yksittäistä NPC:tä varten tehty NPCBase-luokasta periytyvä NPC-luokka, joka on nimetty kyseessä olevan vihollistyypin mukaan. Tämä nimenomainen skripti liitetään Unityssa sitä vastaavaan NPC-peliobjektiin. Luokassa

on luotu muuttujat dataobjekteille ja kyseisen NPC:n käyttämille tiloille. Start()-funktiossa on luotu instanssimuuttujat eri tiloille ja asetettu tilakoneeseen oletustila (ks. esimerkkikoodi 7).

Kuva 21. Vihollinen jahtaustilassa (ChaseState).

Kuva 21 havainnollistaa vihollisen toimintaa. Kun pelaaja tulee tietyn etäisyyden päähän viholliseesta, se lähtee ensin jahtaamaan pelaajaa ja tarpeeksi lähelle päästyään hyökkää tätä kohti. Tässä vihollinen on kohta siirtymässä jahtaustilasta hyökkäystilaan. Siltä on myös mahdollista päästä karkuun, koska se lopettaa jahtaamisen, kun pelaaja on tarpeeksi kaukana.

4.6 Reitinhaun toteutus A*-algoritmilla

2D-tasohyppelypelissä A*-algoritmin käyttö saattaa osoittautua haasteelliseksi, koska solmujen arvoissa pitäisi ottaa huomioon myös painovoiman vaikutus. Reitinhakua päätettiin kokeilla lentävään NPC-hahmoon, jolloin painovoiman olemassaolon voi käytännössä unohtaa.

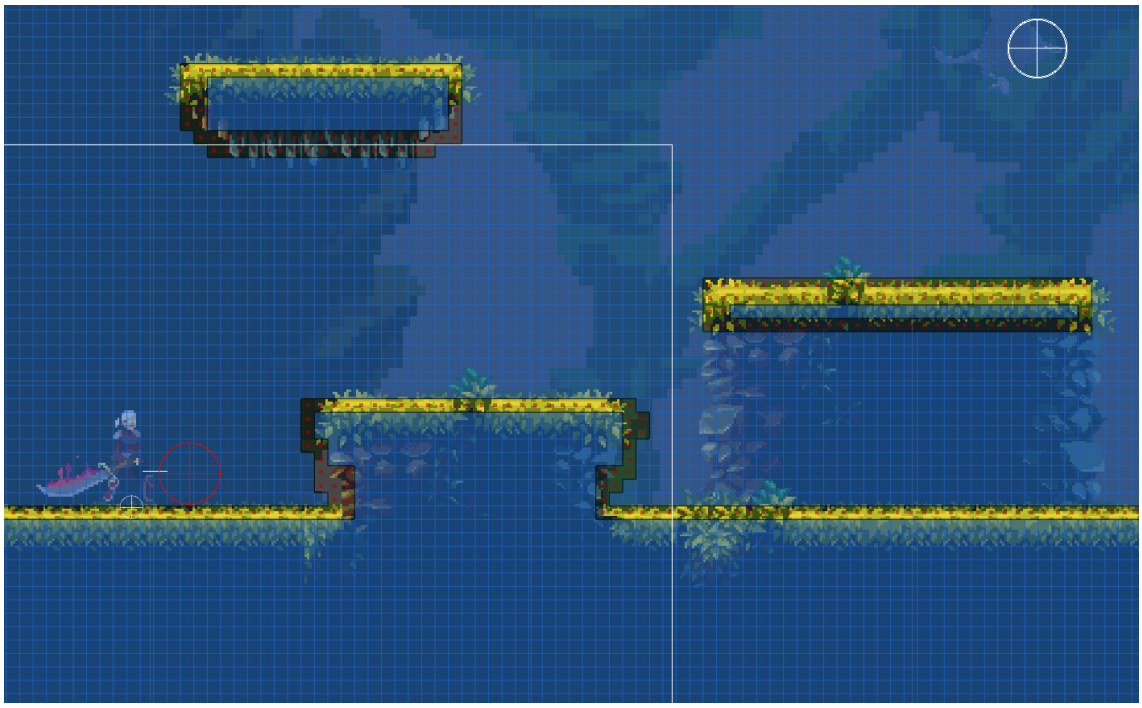
A*-algoritmin toiminnan voi toki ohjelmoida alusta asti käsin, mutta siinä on mahdollista hyödyntää myös Unityyn ladattavaa lisäosaa A* Pathfinding Project. [35.] Tässä projektissa kokeiltiin molempia tapoja, mutta varsinaiseen NPC-hahmoon käytettiin lisäosan

avulla toteutettua reitinhakua, jonka toimintaa säädeltiin kustomoidulla skriptillä halutunlaiseksi.

Manuaalisesti toteutettavassa versiossa pitää tehdä skripti, jossa pelikenttä jaetaan 2D-pelin ollessa kyseessä ruudukkoon (engl. grid). Jokainen ruutu edustaa sijaintia maailmassa ja solmua itse reitinhaussa. Varsinainen reitinhakuskripti käyttää tätä ruudukkoa reitin etsimiseen. Myös yksittäistä solmua varten kannattaa tehdä oma skripti, jossa voidaan esimerkiksi laskea solmun F-arvo. Suoraan ja vinottain liikkumisen hinnat lasketaan reitinhakuskriptissä.

A* Pathfinding Project on erittäin monipuolinen työkalu reitinhakuun, ja tukee sekä 2D- että 3D-pelejä. Mukana tulee tarkka dokumentaatio ja useita malliskenaarioita, joissa reitinhaun toimintaa pystyy testaamaan erilaisissa ympäristöissä ja tilanteissa.

Kun projekti on ladattu ja tuotu Unityyn, luodaan uusi peliobjekti, joka on reitinhaun tärkein elementti ja määrittää kuljettavan alueen. Tähän lisätään uusi komponentti Pathfinder, joka on reitinhaun pääskripti. Ensimmäisenä luodaan uusi graafi, joka tässä tapauksessa on Grid Graph eli ruudukkograafi. Se muodostaa pelikentän päälle ruudukon, jonka voi määritellä juuri oikean kokoiseksi. Skriptissä on runsaasti parametreja kustomointia varten. Myös ruudut eli solmut voidaan säätää juuri halutun kokoisiksi. Mitä pienempiä ne ovat, sitä tarkemmin reitinhaku toimii, mutta laskenta vie tällöin pidemmän aikaa. Jokainen solmu myös tutkitaan törmäyksiä tunnistamalla, onko se kuljettavissa vai onko tiellä esteitä. Kaikki ei-kuljettavat olevat alueet voidaan määritellä LayerMaskin avulla.



Kuva 22. Reitinhakugraafi

Kuvassa 22 näkyy pelikenttää sekä ruudukkograafi, jota reitinhakualgoritmi käyttää reittiä etsiessään. Solmujen koko on tässä säädetty erittäin pieneksi (0,3 yksikköä). Sininen alue edustaa kuljettavissa olevia solmuja.

Kun AIPath-skripti liitetään peliobjektiin, se lisää automaattisesti myös välttämättömän Seeker-skriptin. Seeker-skripti generoi reitin ja prosessoi sitä jälkeenpäin. AIPath-skripti seuraa reittiä, jonka Seeker on laskenut. Se muun muassa määrittelee, miten usein reitinlaskenta suoritetaan ja hoitaa tarvittaessa myös liikkumisen. AIPath-skriptin voi halutessaan korvata omalla skriptillä, kuten tässä projektissa tehtiin.

Peliobjektiin on mahdollista liittää myös AI Destination Setter -skripti, johon saadaan suoraan asetettua seurattava kohde, tässä tapauksessa pelaaja. Kohde oli kuitenkin jo valmiiksi määriteltynä NPC:tä kontrolloivassa skriptissä, joten samaa muuttujaa hyödynnettiin myös reitinhaussa.

```
public class RavenController : MonoBehaviour
{
    [SerializeField]
    private Transform
        target,
        damageCheck;
    [SerializeField]
    private float
```

```
        moveSpeed,
        pickNextWpDistance,
        attackDistance,
        updatePath;
    private Path path;
    private Seeker seeker;
    private int currentWp;
    private bool pathEnded = false;

    void Start()
    {
        seeker = GetComponent<Seeker>();
        rb = GetComponent<Rigidbody2D>();
        anim = GetComponent<Animator>();
        currentWp = 0;
        InvokeRepeating("UpdatePath", 0f, updatePath);
    }

    void UpdatePath()
    {
        if (seeker.IsDone())
        {
            seeker.StartPath(rb.position, target.position, OnPathComplete);
        }
    }

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

Esimerkkikoodi 11. Osa NPC-skriptiä

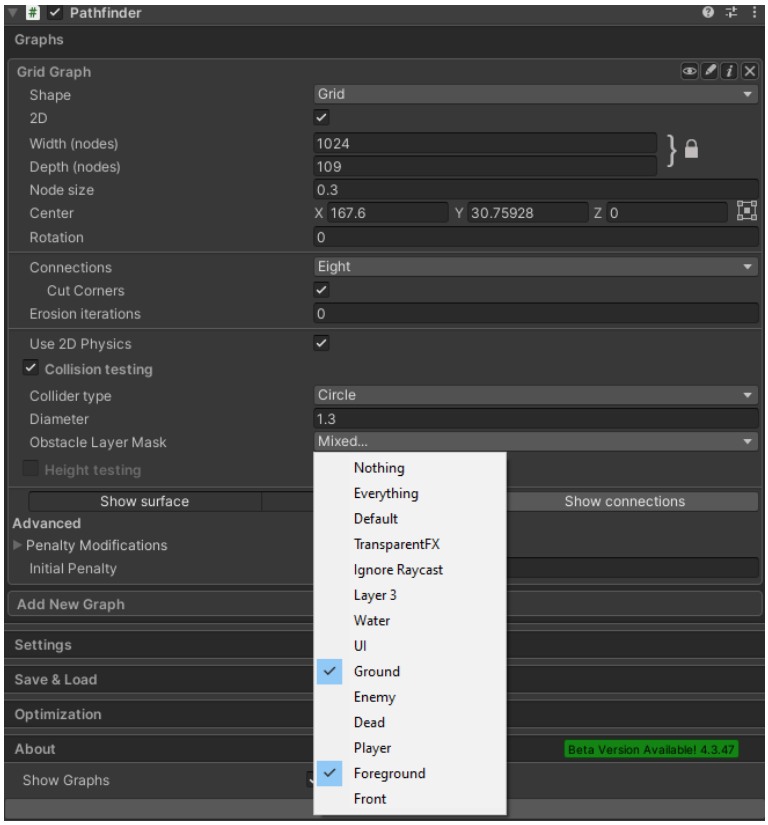
Esimerkkikoodissa 11 näkyy skripti, joka kontrolloi reitinhakua käyttävää NPC:tä. Siinä on määritelty tarvittavat muuttujat reitinhakua varten. Muuttujaan pickNextWpDistance tallennetaan ensinnäkin etäisyys seuraavaan pisteeseen, eli kuinka lähellä liikkumispistettä pitää olla ennen kuin siirrytään seuraavaan. Path on senhetkinen reitti, mitä liikutaan ja currentWp liikkumispiste kyseisen reitin varrella. Lisäksi halutaan pitää kirjaa siitä, onko saavuttu reitin loppuun.

Funktiot UpdatePath() ja OnPathComplete() ovat kohteen löytymistä ja reitin generointia varten. InvokeRepeating()-funktio toistaa samaa skriptiä jatkuvasti, eli reittiä päivitetään koko ajan. Päivitysväliä kannattaa säätää sopivaksi, koska jatkuva päivittäminen vie melko paljon laskentatehoa. Seeker-skripti ottaa vain yhden reitinhakukutsun kerrallaan. Seeker.StartPath()-funktion kutsu laittaa reitinhaun vain jonoon, eli reitti ei ole laskettu heti tämän jälkeen. Tämä johtuu siitä, että jos reitinhaun laskentaa suoritetaan samanaikaisesti monen eri yksikön toimesta, ruudunpäivitysnopeus saattaa pudota, jollei laskentaa hajauteta usealle framelle.

```
if (path == null)
    return;

if (currentWp >= path.vectorPath.Count) {
    pathEnded = true;
    return;
    } else {
        pathEnded = false;
    }


Vector2 direction = ((Vector2)path.vectorPath[currentWp] -
rb.position).normalized;
Vector2 movement = direction * moveSpeed * Time.deltaTime;
rb.AddForce(movement);

float distance = Vector2.Distance(rb.position, path.vectorPath[currentWp]);

     if (distance < nxtWpDistance) {
         currentWp++;
     }
```

Esimerkkikoodi 12. Kulkeminen reittiä pitkin

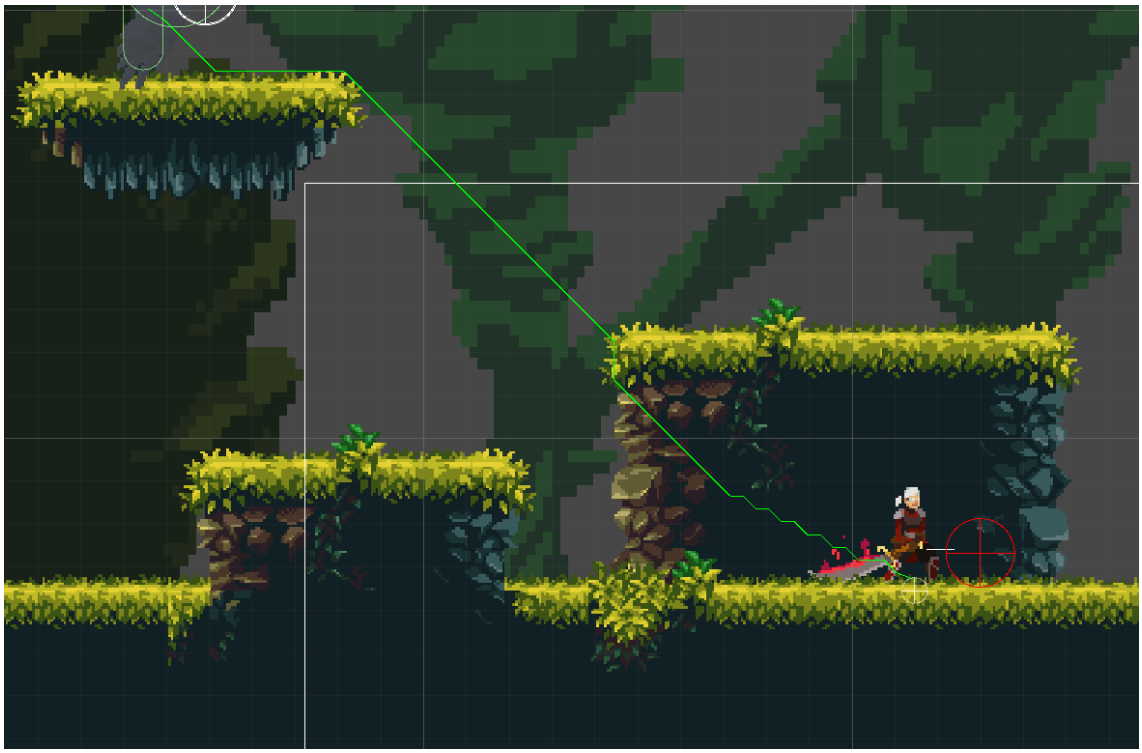
Esimerkkikoodi 12 näyttää, mitä tapahtuu luokan FixedUpdate()-funktiossa. Ensin pitää varmistaa, että ylipäätään on reitti, mitä seurata, ja toiseksi, onko reitillä vielä kulkemispisteitä vai onko jo saavuttu kohteeseen. Liikkumista varten pitää tietää suunta reitin seuraavalle pisteelle. Suunta selviää vähentämällä NPC:n sijainti currentWp:n tarkasta sijainnista, jolloin saadaan vektori NPC:n sijainnista seuraavaan liikkumispisteeseen. Normalized-määre varmistaa, että tämän vektorin pituus on 1.



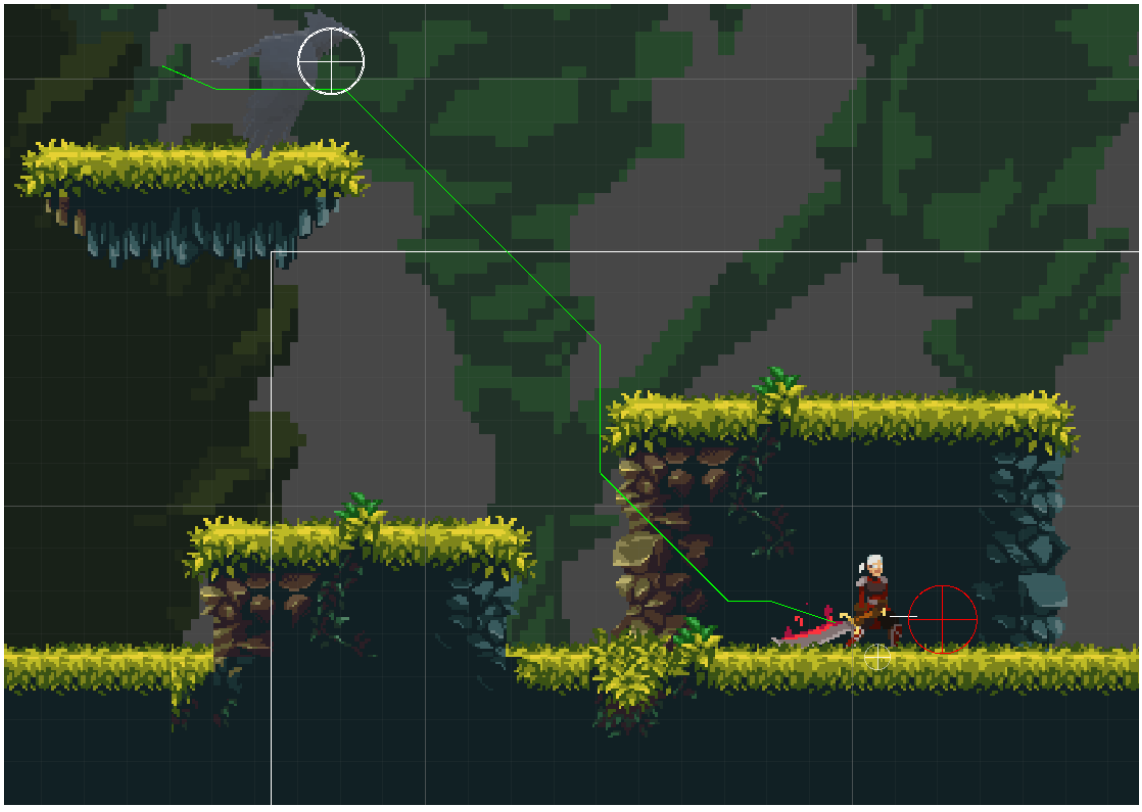
Kuva 23. Pääskriptin asetukset

Kuvassa 23 näkyy pääskriptin eli Pathfinderin asetukset Unityssa. Ruudukon ja solmujen koko voidaan asettaa sopiviksi ja väistettävät alueet määritellä Obstacle Layer Mask -kohdassa. Myös esimerkiksi colliderin tyypin ja halkaisijan voi säätää mieleisekseen.

Kuvat 24 ja 25 (alla) havainnollistavat, miten reitin generointi näkyy reaaliaikaisesti gizmojen avulla. Kuvassa 25 solmujen kokoa on kasvatettu, jolloin myös reitinhaun tarkkuus pienenee.



Kuva 24. Reitin generointia



Kuva 25. Reitin generointia

5 Tekoälyn toiminnan arviointi

Peliin toteutettiin onnistuneesti kaksi erilaista tekoälyä: äärellinen tilakone ja reitinhaku. Lisäksi luotiin yksinkertainen tekoäly NPC-hahmoon ilman varsinaisia tekoälyalgoritmeja. Pelin toiminnan kannalta ei ole merkitystä, että siitä löytyy erilaisia tekniikoita käyttäviä NPC-hahmoja. Osa NPC-hahmoista käyttää vain yksinkertaista, hahmoa kontrolloivaan pääskriptiin upotettua ”tekoälyä”, joka havainnoi ympäristöä ja liikkuu itsenäisesti vaihdellen kahden tai kolmen eri tilan välillä. Osa taas käyttää joko yksinkertaista tai monimutkaisempaa tilakonetta ja osa reitinhakua. Tällä tavalla saman pelin sisällä oli mahdollista kokeilla eri tekniikoita ja myös vertailla niitä toisiinsa.

NPC-hahmon yksinkertainen tekoäly

Yksinkertaisen tekoälyn toteutus NPC-hahmoon liikkumispisteitä ja Raycastia hyödyntäen toimii hyvin tämänkaltaisessa pelissä. Periaatteessa pelistä saisi täysin toimivan kokonaisuuden ilman mitään tämän monimutkaisempaa tekoälyä. Etenkin tällaista toteutusta voisi käyttää niin sanotuissa ystävällisissä NPC-hahmoissa, jotka eivät hyökkää pelaajan kimppuun, mutta joita kuitenkin on mahdollista eliminoida esimerkiksi keräiltävien objektien toivossa.

Jos kyseistä toteutustapaa käytetään vihamielisessä NPC-hahmossa, jonka pitäisi havaita pelaaja myös esimerkiksi takaapäin, Raycasteja pitäisi tehdä useampi. Tässä mielessä toimivampi ratkaisu voisi olla toteutusosiossakin mainittu collider, joka toimii trigger-alueena. Se on mahdollista määrittää juuri sen kokoiseksi kuin tarve vaatii, ja sen avulla lähestyminen voitaisiin havaita mistä tahansa suunnasta tultaessa.

NPC:n liikkumista toteuttaessa kävi melko aikaisessa vaiheessa ilmi, että se saattoi välillä jäädä jumiin platformille. Ongelman sai kuitenkin korjatuksi tekemällä pelikentän liikuttavissa olevan alueen omaksi peliobjektikseen ja asettamalla nurmikon tai muu epätasaisen maaston etualalle ilman collideria. Jumiin jääminen on mahdollista myös estää lisäämällä kitkaton materiaali pelihahmon Rigidbody 2D -komponenttiin.

Toteutuksessa ei ilmennyt muita mainittavia haasteita, ja se saatiin toimimaan juuri toivotulla tavalla. Itse koodi oli hyvin yksinkertainen, eikä ”tekoälyä” varten ollut tarvetta tehdä edes erillistä skriptiä. Halutut toiminnot saatiin järkevästi upotettua NPC:tä kontrolloivaan skriptiin.

Tilakoneen toteutus enum-luokalla

Myös tilakoneen tekeminen enumia ja switch-lauseita käyttäen oli melko mutkaton kokonaisuus. Tässäkin tapauksessa koodin saattoi kirjoittaa suoraan NPC-skriptiin ilman, että kokonaisuudesta olisi tullut mitenkään sekava. Kyseinen toteutus toimii hyvin yksinkertaisilla NPC-hahmoilla, joilla ei ole useita erilaisia tiloja ja toimintoja. Tilakoneen kasvaessa koodista tulee kuitenkin helposti vaikeaselkoinen, etenkin jos myös tilanvaihto if-lauseineen tapahtuu switch-rakenteen sisällä.

Tilakone toimi kuten pitikin ja tuntuu soveltuvan hyvin käytettäväksi tällaisissa tapauksissa. Tätä vaihtoehtoa voisi hyvin käyttää laajentamaan ensimmäisenä tehtyä liikkumispisteitä käyttävää ja ympäristöä havainnoivaa NPC-hahmoa.

Tilakoneen toteutus periytymisellä

Tilakoneen toteutus periytymistä hyödyntäen oli kaikkein työläin kokonaisuus. Erilaisia luokkia kertyy nopeasti paljon. Skriptien ja koodin määrän kasvaessa oli välillä hankala hahmottaa, mitä tapahtuu missäkin. Yksityiskohtaisen luokkakaavion tekeminen olisi varmasti selkeyttänyt tätä vaihetta. Aikaa vievän pohjatyön jälkeen periytyminen kuitenkin helpottaa paljon, ja uusien vihollisten luominen peliin on vaivatonta.

Oikeastaan ainoat haasteet tilakoneen toteutuksessa ilmenivät tilamuutosten kohdalla. Siirtyminen ei aina tapahtunut niin kuin oli tarkoitus. Debug.Log()-toiminnon lisääminen koodin eri kohtiin oli suureksi avuksi vianmäärityksessä, kun haluttiin nähdä, mihin asti koodin suoritus eteni tavoitteiden mukaisesti. Debug-luokan Log-funktiolla Unityn konsoli-ikkunaan voidaan tulostaa haluttu teksti. Tämän avulla oli esimerkiksi helppo testata, oliko toivottua tilamuutosta ylipäätään tapahtunut, kirjoittamalla kyseinen koodirivi tilaluokan EnterState()-funktioon. Poikkeuksetta kyseessä oli aina jokin inhimillinen virhe, kuten muuttujan arvon päivitys tai jokin muu koodirivin unohtuminen jostain kohtaa skriptiä.

Toteutunutta tilakonetta voisi helposti ruveta laajentamaan myös lisäämällä erilaisia tiloja, kuten pakeneminen, piiloutuminen tai suojautuminen. Myös tilojen hierarkiaa voisi jalostaa vaikkapa jakamalla taistelutilanteessa käytettävät tilat eri kategorioihin, kuten hyökkäys, puolustus ja suojaus. Todella monimutkaisen järjestelmän ylläpito on

kuitenkin hankala toteuttaa pelkällä tilakoneella. Toisaalta tällaista retrotasohyppelyä ei ehkä kannatakaan tehdä kovin monimutkaiseksi.

Molemmat tavat tilakoneen toteutuksesta ovat täysin riittäviä yksinkertaisessa tasohyppelypelissä käytettynä. Etenkin periytymistä käyttävää tilakonetta olisi kuitenkin mahdollista jatkokehittää muuttamalla sen toimintaa enemmänkin käytöspuun tapaiseksi tai hyödyntämällä jonkin asteista tarvejärjestelmää, sumeaa logiikkaa tai jopa kaikkia näitä.

Reitinhaku A*-algoritmilla

Kokonaan manuaalisesti ohjelmoitavan reitinhaun toteuttaminen oli projektin monimutkaisin vaihe. Pelkästään reitinhaun käyttämän ruudukon koodaaminen oli työlästä. Siinä vaadittiin myös paljon teoreettista ymmärrystä eri osa-alueista aiheen ympärillä, vaikka varsinainen reitinhaun toimintalogiikka onkin suhteellisen yksinkertainen. Tästä syystä lopullinen, peliin implementoitu reitinhaku päätettiin toteuttaa A* Pathfinding Project -lisäosaa hyödyntäen. Myös aloittelija onnistuu melko varmasti sen avulla toteuttamaan toimivan reitinhaun käyttämällä valmiita skriptejä ilman syvällisempää ymmärrystä algoritmin toiminnasta. Lisäosassa tuli mukana paljon pieniä demopelejä, joissa reitinhaun toimintaa pystyi tutkimaan erilaisissa tilanteissa ja tällä tavalla tutustumaan paremmin sen toimintaan.

Reitinhaku saatiin toimimaan toivotulla tavalla. Jossain vaiheessa vihollisen osumatarkkuus hieman kärsi sen saavuttaessa pelaajan ja ryhtyessä hyökkäämään, mutta tämäkin korjaantui melko hyvin pelkästään solmujen kokoa pienentämällä sekä säätämällä hyökkäysalueen sijaintia sopivaksi jokaisen hyökkäysanimaation framen kohdalla. Välillä hahmo jäi jumiin ahtaisiin paikkoihin, eikä osannut kääntyä pelaajan kierrettyä sen toiselle puolelle. Myös tällaisen jumiin jäämisen mahdollisuus väheni pienentämällä solmujen kokoa. Tällä ei myöskään havaittu olevan erityistä vaikutusta suorituskykyyn.

Kokeilussa todettiin, että reitinhaku toimii hyvin myös 2D-tasohyppelyssä. Lentävässä hahmossa sen toiminnan huomaa erityisesti silloin, jos tiellä on paljon esteitä, kuten ilmassa leijuvia platformeja. Reitinhaun saisi varmasti toimimaan moitteitta myös maassa liikkuviin hahmoihin esimerkiksi säätämällä Rigidbodyn asetuksia, kuten massaa ja painovoiman vaikutusta.

6 Yhteenveto

Opinnäytetyön tavoitteena oli tutustua tekoälyyn yleisellä tasolla sekä hieman syvällisemmin sen soveltamiseen videopeleissä. Työssä toteutettiin 2D-tasohyppelypeli, jossa tekoälyn hyödyntämistä oli mahdollista tutkia käytännössä. Pelin rakentaminen olikin projektin aikaa vievin vaihe, koska aiempaa peliohjelmointikokemusta ei ollut kertynyt vielä kovin paljoa. Alusta asti oli kuitenkin selvää, että projektissa käytettävä peli haluttiin tehdä alusta alkaen itse. Itse tekeminen ei ainoastaan tarjoa mielenkiintoista oppimiskokemusta; tekijä myös tietää tarkalleen, mitä missäkin skriptissä tapahtuu ja miten peli on kokonaisuutena rakennettu. Tällaisesta pelin kokonaisvaltaisesta ja yksityiskohtaisesta tuntemisesta on huomattavaa apua tekoälyn toteuttamisessa.

Yksinkertainen tekoäly oli mahdollista toteuttaa NPC-hahmoille ilman varsinaisten tekoälyalgoritmien käyttöäkin. Tilakone ja reitinhaku toivat kuitenkin huomattavasti lisää eloa ja aitouden tuntua peliin.

Projektissa toteutetusta pelistä puuttuu vielä jonkin verran siihen suunniteltuja ominaisuuksia, kuten keräiltävistä objekteista saatavat voimat, joitakin käyttöliittymän elementtejä sekä äänimaailma. Peliä olisi mahdollista laajentaa paljonkin lisäämällä esimerkiksi enemmän kenttiä, jotka muuttuvat haastavammiksi pelin edetessä. Myös vihollisten vaikeustaso voisi kasvaa eri tavoin, ja peliä pystyisi kehittämään lisäämällä siihen nykyistä monimutkaisempia, useita erilaisia tekoälymenetelmiä hyödyntäviä hahmoja. Peliin voisi lisätä myös erilaisia maastoja ja liikkumistapoja sekä muita toiminnallisuuksia kuten avattavia ovia, teleportteja, ansoja ja esteitä. Pelaajan valittavana voisi myös olla useampi kuin yksi pelaajahahmo, jolloin pelaajaluokkia olisi useita erilaisia. Lisäksi peliin voisi kehittää jonkinlaisen kevyen esine- ja varastojärjestelmän keräiltäviä esineitä ja niiden säilyttämistä varten. Pelaajahahmon käytettävissä voisi myös olla muutama erilainen ase. Tämänkaltaisessa pelissä olisi mahdollista jopa kokeilla proseduraalista generointia kenttien luomiseen.

Kaiken kaikkiaan projektissa onnistuttiin rakentamaan toimiva tekoälykokonaisuus 2D-tasohyppelypeliin, jonka pelaaminen kaikessa yksinkertaisuudessaan on jopa yllättävän miellyttävää ja mukaansatempaavaa.

Lähteet

1 Elements of AI. Verkkoaineisto. Helsingin Yliopisto / Reaktor. <https://course.elementsofai.com> Luettu 23.8.2021.

2 Artificial Intelligence (AI). Verkkoaineisto. IBM. <https://www.ibm.com/cloud/learn/what-is-artificial-intelligence> Luettu 6.3.2020.

3 The Turing Test is Obsolete. It's Time to Build a New Barometer for AI. Verkkoaineisto. FastCompany. <https://www.fastcompany.com/90590042/turing-test-obsolete-ai-benchmark-amazon-alexa> Luettu 14.10.2021.

4 AI in Video Games: Toward a More Intelligent Game. Verkkoaineisto. Harvard. <https://sitn.hms.harvard.edu/flash/2017/ai-video-games-toward-intelligent-game> Luettu 28.7.2019.

5 Alan Turing. Verkkoaineisto. The Stanford Encyclopedia of Philosophy. <https://plato.stanford.edu/archives/win2019/entries/turing> Luettu 20.4.2020.

6 In 1950, Alan Turing Created a Chess Computer Program That Prefigured A.I. Verkkoaineisto. History.com. <https://www.history.com/news/in-1950-alan-turing-created-a-chess-computer-program-that-prefigured-a-i> Luettu 20.4.2020.

7 John McCarthy. Verkkoaineisto. Stanford.edu. <http://jmc.stanford.edu/general/index.html> Luettu 16.6.2021.

8 The Turbulent Past and Uncertain Future of Artificial Intelligence. Verkkoaineisto. IEEE Spectrum. <https://spectrum.ieee.org/history-of-ai> Luettu 14.10.2021.

9 What is Alpha-Beta Pruning? Verkkoaineisto. Educative. <https://www.educative.io/edpresso/what-is-alpha-beta-pruning> Luettu 20.4.2020.

10 20 Years after Deep Blue: How AI Has Advanced Since Conquering Chess. Verkkoaineisto. Scientific American. <https://www.scientificamerican.com/article/20-years-after-deep-blue-how-ai-has-advanced-since-conquering-chess> Luettu 20.4.2020.

11 Which AI has come closest to passing the Turing test? Verkkoaineisto. Dataconomy. <https://dataconomy.com/2021/03/which-ai-closest-passing-turing-test> Luettu 23.8.2021.

12 The Chinese Room Argument. Verkkoaineisto. The Stanford Encyclopedia of Philosophy. <https://plato.stanford.edu/entries/chinese-room> Luettu 20.4.2020.

13 How to benchmark the performance of machine learning platforms. Verkkoaineisto. Neuraldesigner. <https://www.neuraldesigner.com/blog/how-to-benchmark-the-performance-of-machine-learning-platforms> Luettu 6.11.2021.

14 Distinguishing between Narrow AI, General AI and Super AI. Verkkoaineisto. Medium. <https://medium.com/mapping-out-2050/distinguishing-between-narrow-ai-general-ai-and-super-ai-a4bc44172e22> Luettu 6.8.2021.

15 Strong AI. Verkkoaineisto. IBM. <https://www.ibm.com/cloud/learn/strong-ai> Luettu 6.3.2020.

16 AI vs. Machine Learning vs. Deep Learning vs. Neural Networks: What's the Difference? Verkkoaineisto. IBM. <https://www.ibm.com/cloud/blog/ai-vs-machine-learning-vs-deep-learning-vs-neural-networks> Luettu 6.3.2020.

17 Why neural networks aren't fit for natural language understanding. Verkkoaineisto. TechTalks. <https://bdtechtalks.com/2021/07/12/linguistics-for-the-age-of-ai> Luettu 4.8.2021.

18 How artificial intelligence will revolutionize the way video games are developed and played. Verkkoaineisto. The Verge. <https://www.theverge.com/2019/3/6/18222203/video-game-ai-future-procedural-generation-deep-learning> Luettu 18.10.2020.

19 The Total Beginner's Guide to Game AI. Verkkoaineisto. Gamedev.net. <https://www.gamedev.net/articles/programming/artificial-intelligence/the-total-beginners-guide-to-game-ai-r4942> Luettu 14.9.2020.

20 6 Ways Machine Learning will be used in Game Development. Verkkoaineisto. Logikk. <https://www.logikk.com/articles/machine-learning-in-game-development> Luettu 24.3.2020.

21 AlphaStar: Grandmaster level in StarCraft II using multi-agent reinforcement learning. Verkkoaineisto. DeepMind. <https://deepmind.com/blog/article/AlphaStar-Grandmaster-level-in-StarCraft-II-using-multi-agent-reinforcement-learning> Luettu 19.8.2021.

22 Pathfinding Algorithms in Game Development. Verkkoaineisto. IOP Publishing. <https://iopscience.iop.org/article/10.1088/1757-899X/769/1/012021/pdf> Luettu 18.5.2021.

23 Dijkstra's Algorithm in C++ | Shortest Path Algorithm. Verkkoaineisto. FavTutor. <https://favtutor.com/blogs/dijkstras-algorithm-cpp> Luettu 13.8.2021.

24 An Introduction to Utility Theory. Verkkoaineisto. Game AI Pro. <http://www.gameaipro.com/GameAIPro/GameAIPro_Chapter09_An_Introduction_to_Utility_Theory.pdf> Luettu 6.11.2021.

25 Behavior Decision System - Dragon Age Inquisition's Utility Scoring Architecture. Verkkoaineisto. Game AI Pro. <http://www.gameaipro.com/GameAIPro3/GameAIPro3_Chapter31_Behavior_Decision_System_Dragon_Age_Inquisition%E2%80%99s_Utility_Scoring_Architecture.pdf> Luettu 6.11.2021.

26 The Perfect Organism: The AI of Alien: Isolation. Verkkoaineisto. Game Developer. <https://www.gamedeveloper.com/design/the-perfect-organism-the-ai-of-alien-isolation> Luettu 14.9.2020.

27 What is Fuzzy Logic in AI and What are its Applications? Verkkoaineisto. Edureka. <https://www.edureka.co/blog/fuzzy-logic-ai> Luettu 6.8.2021.

28 Nvidia's AI recreates Pac-Man from scratch just by watching it being played. The Verge. <https://www.theverge.com/2020/5/22/21266251/nvidia-ai-gamegan-recreate-pac-man-virutal-environment> Luettu 4.9.2020.

29 Can AI Create Video Games. Verkkoaineisto. Analytics India Magazine. <https://analyticsindiamag.com/can-ai-create-video-games> Luettu 9.1.2021.

30 Nvidia has created the first video game demo using AI-generated graphics. Verkkoaineisto. The Verge. <https://www.theverge.com/2018/12/3/18121198/ai-generated-video-game-graphics-nvidia-driving-demo-neurips> Luettu 4.9.2020.

31 Look at These Incredibly Realistic Faces Generated By A Neural Network. Verkkoaineisto. Futurism. <https://futurism.com/incredibly-realistic-faces-generated-neural-network> Luettu 6.11.2020.

32 Creating Worlds of Endless Variety: An Evaluation of Procedural Content Generation in Gaming Content Generation in Gaming. Verkkoaineisto. The Digital Showcase (Knight-Capron Library & University of Lynchburg). <https://digitalshowcase.lynchburg.edu/cgi/viewcontent.cgi?article=1165&context=utcp> Luettu 23.8.2021.

33 This neural network could make animations in games a little less awkward. Verkkoaineisto. TechCrunch. <https://techcrunch.com/2017/05/01/this-neural-network-could-make-animations-in-games-a-little-less-awkward> Luettu 24.3.2020.

34 C# documentation. Verkkoaineisto. Microsoft. <https://docs.microsoft.com/en-us/dotnet/csharp> Luettu 7.3.2021.

35 A* Pathfinding Project. Verkkoaineisto. Arongranberg. <https://arongranberg.com/astar> Luettu 5.8.