KARELIA-AMMATTIKORKEAKOULU
Tietojenkäsittelyn koulutusohjelma

Arttu Nevalainen

NPC-TEKOÄLY PELEISSÄ

Opinnäytetyö
Marraskuu 2014

OPINNÄYTETYÖ
Marraskuu 2014
Tietojenkäsittelyn koulutusohjelma
Karjalankatu 3
80200 JOENSUU
p. (013) 260 600

Tekijä
Arttu Nevalainen

Nimike
NPC-tekoäly peleissä

Toimeksiantaja
-

Tiivistelmä

Opinnäytetyössä käsitellään NPC-hahmojen toimintaa ja rakennetta peleissä. NPC-hahmot, eli ei pelaajahahmot (non-player characters), ovat tekoälyn kautta käyttäytyviä hahmoja. Työssä selvitetään, mitä tekoäly merkitsee ja miten sitä sovelletaan NPC-hahmoille. Toiminnallisessa osuudessa käsitellään Unity-pelimoottorilla toteutettua reitinhaku menetelmää. Toiminnallisen osuuden tavoitteena oli luoda kaikki tarvittavat vaiheet toimivan reitinhaun toteuttamiseksi.

NPC-hahmon toiminta koostuu monista eri osista, kuten tilanteen luvusta, oppimisesta, reitinhausta sekä pelaajan ja NPC-hahmon välisestä vuorovaikutuksesta. Opinnäytetyössä tutkittiin uusien pelien tekoälyn toimintaa ja sovellettiin niiden eri osia ja menetelmiä.

Opinnäytetyön tuloksena syntyi A-tähti-nimisellä algoritmilla toimiva reitinhaku menetelmä, joka etsii paras ensin -menetelmällä lyhimmän reitin liikuttavaan kohteeseen. Reitinhaun toteutuksen yhteydessä luotiin myös menetelmät liikuttavan alueen laskentaan ja visualisointiin.

Kieli suomi	Sivut 47 Liitteet 4 Liitesivumäärä 8

Asiasanat
NPC, tekoäly

THESIS
November 2014
Degree Programme in Business
Information Technology
Karjalankatu 3
80200 JOENSUU
p. (013) 260 600

Author
Arttu Nevalainen

Title
Artificial Intelligence in Games

Commissionred by
-

Abstract

The thesis goes through NPC behavior and its structure in games. NPC shortens from the words non-player characters and they are used by an artificial intelligence. Study examines, what artificial intelligence means and how it is used with the NPC. The functional part covers the path finding system which is created by using Unity game engine. The goal for the functional part was to create all the necessary phases for a working path finding.

NPC structure consists of many different parts such as state behavior, learning, path finding and a player's and NPC's interaction. The thesis researched an artificial intelligence in the new games and applied their different parts and methods.

The result for the thesis was a path finding system, which utilized A-star named algorithm. It finds the shortest path for the target by using a best-first search algorithm. During this process the methods for the moveable area and its visualization were created.

Language
Finnish

Pages 47
Appendices 4
Pages of Appendices 8

Keywords

NPC, artificial intelligence

Sisältö

Sisältö 4
Sanasto 5
1 Johdanto 6
2 Tekoälyn luokittelu 8
2.1 Vahva tekoäly 8
2.2 Heikko tekoäly 9
3 NPC-toiminta 11
3.1 Yleistä 11
3.2 Tilanteen luku 13
3.2.1 Äärellinen automaatti 13
3.2.2 Käytöspuu 16
3.2.3 Tavoitteeseen suuntaava tehtävän suunnittelija 19
3.3 Oppiminen 20
3.4 Vuorovaikutus 22
3.5 Reitinhaku 26
3.6 NPC tekoälypelaajalla 30
4 NPC-ohjelmointi 31
4.1 Tieni tekoälyn parissa 31
4.2 Reitinhaku A* Unity-pelimoottorilla 34
4.2.1 A*-algoritmin pohjustaminen 34
4.2.2 A*-toteutus 35
4.2.3 Visuaalisen graafin luonti 36
4.2.4 Datagraafin luonti 37
4.2.5 A*-algoritmin luonti 39
5 Yhteenveto 44
Lähteet 46

Liitteet

Liite 1 Visuaalinen graafi
Liite 2 Datagraafi
Liite 3 A*-algoritmi
Liite 4 A*-psoudokoodi

Sanasto

AI	Tulee sanoista Artificial Intelligence ja tarkoittaa tekoälyä (Beal 2014).
Epic Games	Yhdysvaltalainen pelitalo (Epic Games 2014a).
Navigaatioverkko	Reitinhakuun rakennettu liikkumisalue (Epic Games 2014b).
NPC	”Non-player character”, eli ei pelaajan ohjaama hahmo (Webopedia 2014a).
RTS	”Real-time strategy”, eli strategiapeli, jossa ei ole erillisiä vuoroja (Webopedia 2014b).
Unity	Pelimoottori, jolla voidaan toteuttaa 2D- ja 3D-pelejä monille alustoille (Unity 2014).
Unreal Engine	Epic Games -nimisen yrityksen luoma pelimoottori (Epic Games 2014a).
Warcraft	Blizzard-nimisen pelitalon tekemä peli (Blizzard 2014).
World Editor	Warcraft 3:n kenttäeditori (TheHelper 2014).
XML	Merkintäkieli, jolla tiedon merkitys voidaan lukea tiedosta (Webopedia 2014c).

1 Johdanto

Tekoäly on tietojenkäsittelytieteen yksi haara. Siinä keskitytään luomaan tietokoneita, jotka käyttäytyvät kuten ihmiset (McCarthy 1956). Tekoälystä käytetään yleisesti lyhennettä AI, joka tulee sanoista Artificial Intelligence. Tekoäly tuo ihmisille monesti mielikuvan ihmismäisestä sovelluksesta, jolla on oma persoonansa, mutta sillä on myös monia muita sovellutuksia. Yksi näistä on asiantuntijajärjestelmät, jotka auttavat ihmisiä tekemään päätöksiä vaikeissa tilanteissa.

Ensimmäinen asiantuntijajärjestelmä luotiin jo vuonna 1965 Standfordin yliopistossa Edward Feigenbaumin ja Joshua Lederbergin toimesta. Sen tehtävänä oli analysoida kemikaalisia yhdistelmiä. Nykyään vastaavanlaisia järjestelmiä käytetään esimerkiksi lääketieteessä potilaiden diagnosointiin, öljyn jalostukseen ja taloudellisiin investointeihin. (Zwass 2014.)

Toinen tunnettu tekoälysovellus on luonnollisen puheen prosessointi NLP, joka tulee sanoista ”natural language processing”. Siinä tavoitteena on luoda ohjelma, joka ymmärtää ihmisen puhetta ja pystyy vastaamaan siihen mahdollisimman ihmismäisesti. Margaret Rousen (2011) mukaan, toimivan NLP:n luonti ei ole helppoa, koska tietokoneet ymmärtävät yleisesti ohjelmointikieltä, joka on tarkkaa, yksiselitteistä ja huolella rakennettua, tai mahdollisesti valmiiksi lausuttuja äänikomentoja. Kuitenkin ihmisaivot kykenevät ymmärtämään puhetta huomattavasti monipuolisemmin kuin NLP, koska ihminen on kykenevä päättelemään jopa virheellisten lauseiden merkitystä.

Uudet käyttöjärjestelmät tarjoavat jo nykyään melko toimivia äänentunnistusohjelmia, mutta ne vaativat lähes täydellistä englannin kielen lausuntaa toimiakseen kunnolla. Windowsin Cortana ja Applen Siri ovat tunnettuja tekoälysovelluksia, jotka ovat kykeneviä tunnistamaan puhetta.

Jos tekoäly viedään vielä edistyneemmälle tasolle, päästään tasolle, missä tietokoneen laskemat nollat ja ykköset muuttuvat digitaalisesti simuloiduiksi neuronien toiminnaksi. Tämä kutsutaan keinotekoiseksi hermoverkostoksi, joka tulee englannin kielisistä sanoista ”artificial neural networks” lyhennettynä ANNs. Jo vuonna 1943 Warren McCulloch ja Walter Pitts loivat laskennallisen mallin, kuinka kyseinen toiminnollisuus voitaisiin toteuttaa käyttäen matemaattisia algoritmeja (McCulloch & Pitts 1943). Vaikkakin keinotekoinen hermoverkosto viittaa ei-digitaaliseen tietokoneeseen, sitä voidaan simuloida digitaalisella tietokoneella. Tällä hetkellä keinotekoisia hermostoverkostoja käytetään esimerkiksi äänen ja kuvan tunnistuslaitteissa.

Tekoälyn muodostamiseen ei välttämättä tarvita kovinkaan monimutkaista toiminnollisuutta, jos sitä lähdetään tarkastelemaan tietokonepelien tasolla. Yksinkertaisinkin tietokoneen laskema laskutoimitus, joka pyrkii toimimaan esimerkiksi vastustajan pelaajana, voidaan märittää tekoälyksi. Hyvä esimerkki tästä on vuonna 1972 tehty peli ”PONG”. Se on pöytätenniksen tapainen peli, jossa pallo kimpoaa aina vuorotellen toisen sivun kautta. Taas toinen ääripää tekoälyissä on shakkitekoäly, koska siinä on arvioitu olevan 10e43–10e47 erilaista sääntöjen sallimaa pelinappuloiden paikkaa. Nykypäivänä tietokonepelien tekoälyt ovat kehittyneet erittäin korkealle tasolle, koska prosessorien laskentatehot ovat todella suuria. Tämä näkyy esimerkiksi ihmismäisten tekoälyhahmojen realistisesta käyttäytymisestä.

Tekoäly on kiehtonut minua jo nuoresta lähtien, joten se on myös motivoinut minua ohjelmoimaan monenlaisia tekoälyyn liittyviä ohjelmia. Työn tavoitteena on tutkia tietokonepelien tekoälyn ohjaamia hahmoja monella eri tasolla, sekä tarjota lukijalle monipuolinen katsaus niiden ominaisuuksista.

Työssä tutkitaan modernin peliteollisuuden tekoälyä ja selvitetään, kuinka NPC-hahmot (non-player character) toimivat. Lähes jokaisessa uudessa kolmiulotteisessa pelissä NPC-hahmot ovat suuressa roolissa ja niiden toiminnollisuudet ovat laajat ja monimutkaiset. Työssä myös tarkastellaan, kuinka tekoälypelaajat käsittelevät NPC-hahmoja eri peli genreissä, kuten strategia-, tai räiskintäpeleissä, jossa tekoälyllä on samat rajoitteet ja resurssit kuin pelaajallakin. Opinnäytetyöni toiminnallisessa osuudessa selvitetään, kuinka A* reitinhaku toteutetaan Unity-pelimoottorilla alusta loppuun ja sen sovelluksia edistyneimpiin reitinhakumenetelmiin.

2 Tekoälyn luokittelu

Tekoäly voidaan luokitella kahteen eri luokkaan: heikkoon ja vahvaan tekoälyyn.

2.1 Vahva tekoäly

Vahvaa tekoälyä ei ole vielä tietoisesti toteutettu, mutta se voidaan määritellä teoriatasolla. Vahvan tekoälyn tulee täyttää tiettyjä kriteerejä, jotta se ylittäisi heikon tekoälyn. Näistä ehkä tärkein on ihmismäinen ajattelu ja päättelykyky, toisin sanoen vahvan tekoälyn tulisi toimia kuten ihmisaivot tai jopa paremmin. Kuitenkaan tunteet ja persoonallisuudet eivät ole pakollisia ominaisuuksia vahvalla tekoälyllä, mutta niitä voi kuitenkin olla.

Vahvan tekoälyn voisi määritellä ohjelmaksi jolla on oma näkemys tai mielipide asiaan, mikä taas vaatii ominaisuuden ymmärtää ja aistia asioita. Tällöin tekoäly ei tulisi noudattamaan sille itselleen valmiiksi määriteltyjä ehtoja, vain se osaisi itse päätellä tilanteen tai jopa tunteen mukaan, kuinka toimia. (Mooney 2006.)

Realistisimmatkaan NPC-tekoälyt eivät ole päässet lähelle vahvan tekoälyn tasoa, koska laskentatehot tietokoneissa ovat vielä vahvan tekoälyn kriteereille liian pienet. Vaikkakin ihmismäistä toimintaa on helppo simuloida, pelkkä valmiista rakenteista ja toiminnoista luotu NPC-hahmo ei vastaa oikeaa ihmisen ajattelukykyä. Ja jos siihen pisteeseen ikinä peliteollisuudessa päästään, olisiko eettisesti oikein murhata virtuaalista ihmistä?

Vahvan tekoälyn vaatimukset ovat hyvin lähellä ihmisaivojen tapaista toimintaa ja siksi sen toteuttaminen on vielä nykyteknologialla hyvin kaukaista. Hubert Dreyfus (1972) on kuvaillut näkemyksensä näin: ”Jos hermosto noudattaa fysiikan ja kemian lakeja, jota meidän on erittäin hyvä syy uskoa, meidän tulisi silloin pystyä uudelleen luomaan samantapainen käyttäytyminen käyttäen fyysisiä laitteita.” Vuonna 2005 kyseinen käyttäytyminen, joka vastasi yhden sekunnin ihmisaivojen toimintaa, toteutettiin. Prosessi vaati kuitenkin huikeat 50 päivää 27 prosessorilla. (Izhikevich & Edelman 2007.)

On vaikeaa kuvailla jotain mitä ei ole vielä olemassa, mutta ihmiset ovat kuitenkin luoneet monia vahvoja tekoälyjä elokuviin. Kaikki varmastikin tuntevat elokuvasarjassa ”Terminator” esiintyvän tekoälyn nimeltä Skynet. Se kehittyi shakkitekoälyn pohjalta ja päätyi Amerikan digitaaliseksi puolustusjärjestelmäksi. Kuitenkin inhimillisen virheen tuloksena se karkasi ihmisten kontrollista ja päätyi tuhoamaan koko ihmiskunnan.

2.2 Heikko tekoäly

Heikko tekoäly suorittaa sille annettuja komentoja ja laskutoimituksia, jotta se lopulta pääsee haluttuun lopputulokseen. Suurin ero vahvan ja heikon tekoälyn välillä onkin siinä, että heikko tekoäly ei osaa itse päättää, kuinka se ratkaisisi kyseisen ongelman, koska se joutuu noudattamaan sille annettuja komentoja. Tämä taas vaikuttaa huomattavasti heikon tekoälyn laatuun, koska mitä monipuolisemmin eri tilanteisiin avautuu uusia toimintoja, joista voidaan valita optimaalisin vaihtoehto, sitä tehokkaammalta tekoäly vaikuttaa. Kuitenkin mitä monipuolisemmaksi tekoäly muuttuu, sitä enemmän muistia ja laskentatehoa se vaatii tietokoneelta. (Indika 2011.)

Heikko tekoäly ei enää nykyteknologialla ole kovin heikkoa. Ainakin George Dvorsky (2013) on vahvasti sitä mieltä: ”Mutta älä anna sen nimen häikäistä, siinä ei ole mitään heikkoa siihen nähden mitä vahinkoa se voisi aiheuttaa”, viitaten siihen, että nykyään monet laitteet ja järjestelmät toimivat käytännössä heikolla tekoälyllä. Vastaavissa tapauksissa olisi parasta, että tekoälyllä ei olisi tunteita tai omia mielipiteitä, koska ristiriitatilanteista voisi syntyä vakavia seurauksia.

NPC-hahmot yleisesti käyttävät heikkoa tekoälyä riippuen kuinka monipuolisiksi ne on haluttu ohjelmoida. Uusimmissa hiiviskelypeleissä NPC-hahmojen ihmismäisyys on viety jo erittäin korkealle tasolle. Vaikkakin tekoäly edelleen noudattaa tiettyjä kaavoja ja komentoja, sillä pystytään jo simuloimaan lähes ihmismäinen käyttäytyminen, koska nykytietokoneiden laskentatehot ovat heikon tekoälyn kriteereille erittäin suuret.

Kuvassa 1 on esimerkki yhdestä uusimmista hiiviskelypeleistä, nimeltä Hitman Absolution. Pelaaja on selvästikin astunut alueelle, mihin vain asianomaiset henkilöt ovat tervetulleita ja kuvan kaksi hohtavaa henkilöä ovat alkaneet epäillä pelaajan identiteettiä. Pelaaja peittää kasvojaan vetämällä hattuaan alaspäin. Tällöin NPC-hahmojen kiinnostus pelaajaa kohtaan vähenee ja pelaaja voi edetä aiheuttamatta minkäänlaista välikohtausta. On kuitenkin mahdollista, että NPC-hahmojoen kiinnostus on noussut tasolle, jossa hahmoja alkaa kiinnostamaan pelaajan identiteetti. Jos pelaaja ei poistu paikalta tai edes NPC-hahmojen lähipiiristä ajoissa, voivat hahmot kysyä pelaajalta, kuka tämä oikein on.

Kuva 1. Kuva pelistä Hitman Absolution (kuvankaappaus).

3 NPC-toiminta

3.1 Yleistä

NPC:t, eli ”non-player characters” ovat hahmoja, joka on ohjelmoitu kommunikoimaan pelaajan kanssa. Niillä voi olla monta eri roolia pelissä ja yleensä jokainen rooli käyttäytyy eri tavalla kohdatessaan itse pelaajahahmon. Yksi oleellisimmista lähtökohdista tulee vastaan, kun tarkastellaan, onko kyseinen NPC-hahmo pelaajan kanssa samalla puolella vai luokitellaanko se vihollishahmoksi. Tämä vaihtelee todella paljon riippuen pelin genrestä, esimerkiksi sotapeleissä usein kaksi maata taistelee toisiaan vastaan ja silloin vihollis-NPC-hahmoiksi luokiteltaisiin tekoälyn ohjaamat vastakkaisen maan joukot. Taas pelaajan kanssa samalla puolella olisivat luonnollisesti saman maalaiset taistelutoverit. Vaihtoehtoja on toki vielä kolmas, pelaajan kanssa neutraalissa tilassa olevat hahmot. Hyvä esimerkki neutraalista hahmosta olisi siviilihahmot.

Toinen tärkeä ominaisuus on aggressiivisuus. Sillä voitaisi esimerkiksi viitata eläinten ja taruolentojen käyttäytymiseen pelaaja kohtaan, koska niitä ei yleensä luokitella suoraan pelaajan vihollisiksi. Jos kyseessä olisi susi, se sopisi hyvin aggressiiviseksi NPC-hahmoksi, mutta taas jos pelaaja törmäisi metsässä peuraan, se olisi passiivinen. Jos pelaaja ärsyttää passiivista hahmoa esimerkiksi lyömällä sitä, passiivinen NPC voi muuttua aggressiiviseksi ja kääntyisi pelaajaa vastaan. Kuvassa 2 on susi-NPC monien tuntemasta pelistä, Minecraftista. Se ei kuitenkaan hyökkää pelaajan kimppuun, ellei pelaaja vahingoita sitä ensin.

Kuva 2. Susi pelistä MineCraft (kuvankaappaus).

Mutta kuinka kaikki tämä toimii? Jokaisella hahmolla on jonkinlainen ohjelmoitu toiminta, jota se noudattaa. Tämä toiminta taas saattaa vaihdella satunnaisesti ja muuttua pelin tilanteen edetessä. Yleensä edistyneimmillä NPC-hahmoilla on jonkin muotoinen sisäänkirjoitettu toiminnollisuus tai useiden toiminnollisuuksien yhdistelmä, joka tilanteiden mukaan laukaisee NPC-hahmon senhetkisen toiminnan. Tällaisia toiminnollisuuksia olisivat esimerkiksi äärellinen automaatti (Finite State Machine), käytöspuut (Behavior Tree) ja tavoitteeseen suuntaava tehtävän suunnittelija (Goal-Oriented Action Planning). (Simpson 2014, Owens 2014.)

Hyvin usein pelien kategoriat ovat joko toimintaa, ensimmäisen persoonan ampumista tai seikkailua. Lähes jokaisessa näistä esiintyy jonkin muotoista taistelua pelaaja vastaan. Jotta taas taistelusta tulisi pelaajalle mielenkiintoista, tulee tekoälyn toimia tehokkaasti. Siksipä nykypäivänä on yritetty luoda ihmispelaajan tasoisia NPC-hahmoja tai tekoälypelaajia tai ainakin simuloida vastaavaa toimintaa (Schreiner, 2009).

Yksi tärkeimmäksi NPC-hahmojen ominaisuudeksi on muodostunut kyky metsästää (Schreiner, 2009). NPC-hahmolle on annettu selvä tehtävä löytää pelaaja tai jokin muu vastapuolen NPC-hahmo. Tällöin NPC-hahmo voi käyttää kaikkia samoja resursseja pelaajan etsinnässä, kuin joita pelaajahahmokin voisi käyttää pelin sisäisessä maailmassa. NPC-hahmo voi olla esimerkiksi sotilas, joka hyökkää pelaajan tiimin tukikohtaan etsimällä mahdollisia vastustajia, tai vartija, joka on havainnut epäilyttäviä jälkiä maastossa lähtiessään etsimään mahdollisia tunkeilijoita.

Toinen tärkeä ominaisuus on selviytymisvaisto (Schreiner, 2009). NPC-hahmolle voi aktivoitua eri tiloja haavoittumisen tai muuten hankalaan paikkaan joutumisen yhteydessä. Uuden tilan aktivoiduttua, hahmon päätavoite muuttuisi vastustajan eliminoinnista oman henkensä varjelukseksi. Edelleen NPC-hahmo voi käyttää hyödykseen samoja ominaisuuksia kuin pelaajatkin, kuten maastoa ja kommunikointia. NPC-hahmo voi esimerkiksi piiloutua kiven taakse, ryömiä pusikossa tai huutaa tovereiltaan tukea.

Tällaisia NPC-hahmoja käytetään esimerkiksi sotapeleissä, joissa jokainen vihollispelaaja on erillinen NPC-hahmo. Nämä tekoälypelaajat eivät välttämättä ole kaikki pelaajan puolella ja voivat taistella myös toisiaan vastaan. Se yleensä myös estää NPC-hahmojen keskinäisen tiedonjaon, joka tekee itse pelistä reilumpaa ja realistisempaa. Muulloin ihmispelaajat joutuisivat nopeasti alakynteen.

3.2 Tilanteen luku

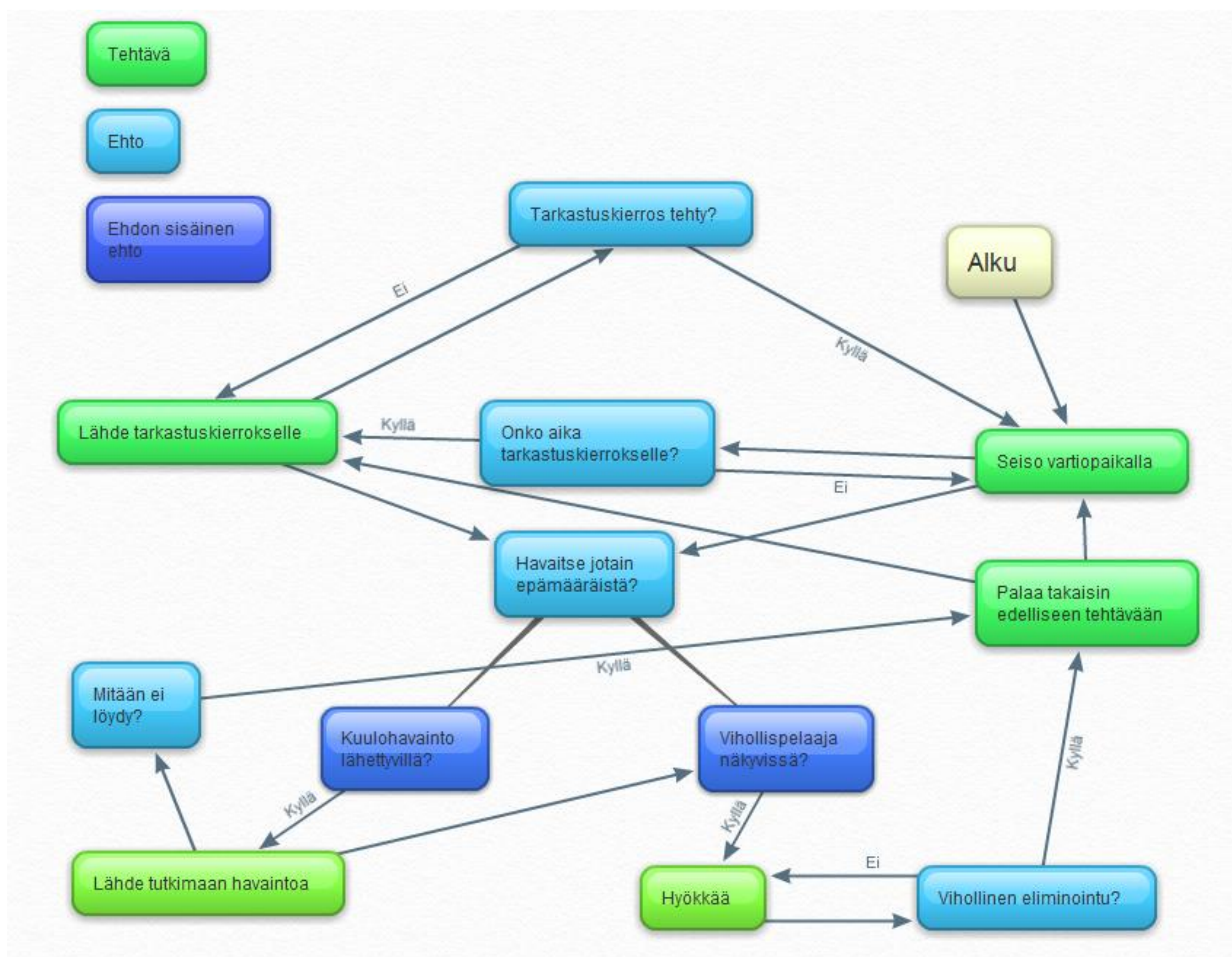
3.2.1 Äärellinen automaatti

Älykkäällä NPC-hahmolla voi hyvin useasti olla monen tyyppistä toimintaa, joka riippuu ympärillä muuttuvista tilanteista. Tietyt tilanteet voivat aktivoida NPC-hahmolle uusia tiloja, jotka taas aukaisevat mahdollisuuden aktivoida kyseisestä tilasta erikoistuvia tiloja. Jos taas mitään ei tapahdu ja tilanne niin sanotusti rauhoittuu, voi NPC:n tila palautua takaisin sen oletustilaan. Vastaavaa toimintaa voidaan kuvata äärellisellä automaatilla (Brownlee 2002).

Jos otamme tutkinnan kohteeksi mahdollisimman yksinkertaisen ja monellekin tutun NPC-hahmotyypin: vartijan, voimme verrata sitä kätevästi oikeiden vartijoiden toimenkuvaan. Jotta vartija osaisi toimia jokaisessa tilanteessa oikealla tavalla, hänen tulee noudattaa tiettyjä sääntöjä, joita hänelle on ennalta määritelty. Tämä periaate toimii hyvin samalla tavalla myös tekoälyn omaavalle vartijalle.

Ajatellaan että NPC-vartijalla on tietty tila johon hän aina palaa takaisin, jos mitään muuta tilaa ei ole aktivoitu. Tämä voisi olla tila jossa vartija lähtee kiertämään määritettyä vartiointireittiä tarkastellen ympäristöä mahdollisten vihollisten varalta. Jos taas vartija havaitsee katsekontaktilla toisen hahmon, joka vartijan ohjeistuksen mukaan on laskettava vihollishahmoksi, vartijalle aktivoituu uusi tila. Kuviossa 1 on hyvin yksinkertainen esimerkki siitä, kuinka NPC-vartija voisi toimia. Kuvion vihreät ruudut ovat vartijan tehtäviä, jotka tarkastavat sinisten ehtoruutujen kautta mihin toimintaa seuraavaksi tulisi siirtyä.

Uuden tilan aktivoiduttua vartijan ajattelu muuttuu erilaiseksi ja tuo uusia toimintoja hänen käyttäytymiseen. Esimerkiksi havaittuaan vihollisen vartija voi pohtia, tulisiko hänen tehdä hälytys vai hyökätä pelaajan kimppuun. Tällöin tekoälyn tulee käsitellä ehtoja, joista lopuksi voidaan laskea paras mahdollinen lopputulos. Kyseisessä tilanteessa ehtoja voisivat olla seuraavat: onko lähistöllä muita vartijoita, onko vihollinen aseistettu, tarvitaanko hälytystä?



Kuvio 1. Esimerkki äärellisestä automaatista.

Uusissa peleissä NPC-hahmot saattavat reagoida kuulohavainnoinnin kautta tilanteeseen. Tämä on yleistä hiiviskelytyyppisissä peleissä, jossa pieninkin pelaajan virhe voi laukaista NPC-hahmojen huomion. Välttämättä pieni kuulohavainto ei laukaise vartijatyyppisellä NPC-hahmolla suoraa hälytystä, vain ihmismäisen valmiustilan, jossa vartija alkaa tarkastelemaan ympäristöään tarkemmin tai jopa lähtee tutkimaan äänen lähdettä.

NPC-hahmot voivat myös kommunikoida keskenään tai jopa käskeä toisia hahmoja toimimaan eri tavalla. Vastaavissa tilanteissa NPC-hahmojen omakohtainen ajattelu tulee ohittaa ja aktivoida suoraan toiminnollisuus, jonka käskyttävä NPC-hahmo on määrännyt. Esimerkkinä tallaisesta tilanteesta olisi vartijan kuullessa esimiehensä käskyn liikkua uuteen vartiointipaikkaan.

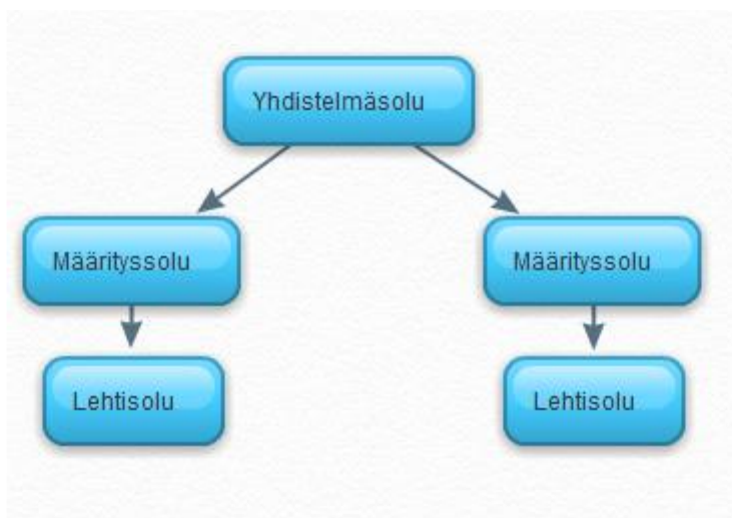
3.2.2 Käytöspuu

Äärellinen automaatti on vain yksi tapa kuvata NPC-hahmon toimintaa. Toinen tunnettu systeemi on nimeltään käytöspuu, jossa puu haarautuu hierarkkisiin haaroihin, jotka ohjaavat tekoälyn päätöksentekoa. Jokainen haara päättyy lehteen, missä varsinaiset tekoälyä ohjaavat komennot sijaitsevat. Uuden tapahtuman aktivoituessa tekoäly lähtee laskemaan juuresta ylöspäin kohti sopivinta lehteä kyseisen tilanteen ratkaisemiseksi. (Simpson 2014.)

Käytöspuuta voi soveltaa hyvin monella tapaa, koska sen rakennetta voidaan käyttää tehokkaasti monissa toiminnoissa. Sen voi esimerkiksi jakaa kahteen eri käyttömekaniikkaan, tietojakoiseen ja koodijakoiseen toimintaan. Kuitenkaan mikään ei estä käyttämään molempia mekaniikoita. Tällainen ratkaisu toimisi hyvin grafiikkapohjaisen pelimoottorin tekoälyeditorissa, jossa tekijän on helppo rakentaa omia käytöspuita käsin. Jokaisesta tietoa sisältävästä solmusta aukeaa aina tietokannan antamat vaihtoehdot, joista tekijä voi valintansa mukaan rakentaa omanlaisensa puun. (Simpson, 2014.)

Kun puu on rakennettu ja koodin omaava NPC-hahmo alkaa toimimaan pelin käynnistyttyä, muuttuu puun käsittely koodijakoiseksi. Toisin sanoen solmut, jotka sisälsivät tietoa, ovat NPC-hahmolle koodillisia toimintoja, jotka ohjaavat tekoälyn toimintaa. Mutta mitä kaikkea ne voisivat sisältää ja miten solmujen välinen virtaus toimii? (Simpson, 2014.)

Käytännössä mitään virallista määritelmää solmujen toiminnalle ja niiden väliselle virtaukselle ei ole, mutta kovin usein käytetty yksinkertainen menetelmä on onnistumis-, epäonnistumis- ja suoritus menetelmät. Solmut voidaan myös jakaa omiin luokkiinsa, kuten esimerkiksi yhdistelmä-, määritys- ja lehtisolmuihin. Yleisesti ottaen yhdistelmäsolulla on useampi lapsisolu, joka taas voi olla puolestaan mikä tahansa yllämainitusta soluluokasta. Kuvio 2 kertoo lyhyesti, kuinka yhdistelmäsolu ja sen lapsisolut voisivat muodostua. (Simpson 2014.)



Kuvio 2. Käytöspuun rakenne.

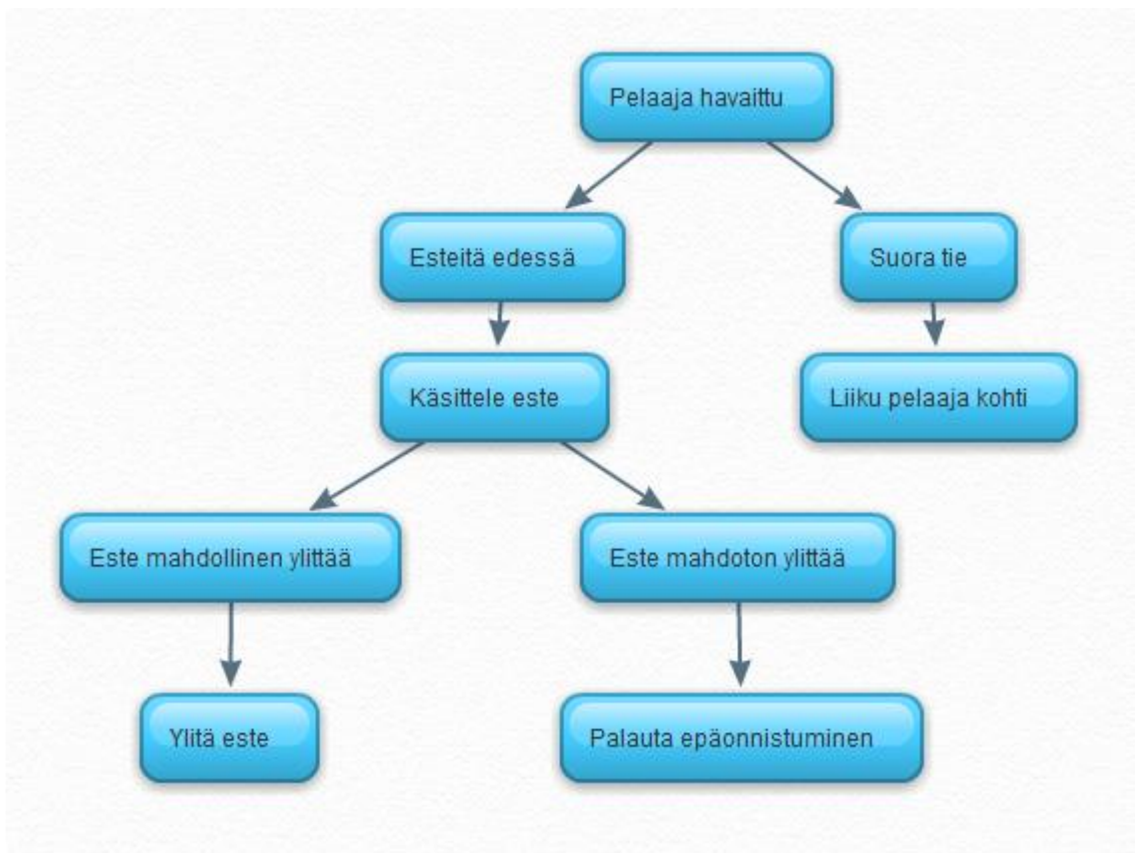
Yhdistelmäsolu voi suorittaa lapsisolunsa joko järjestelmällisesti tai satunnaisessa järjestyksessä sen mukaan millainen yhdistemäsolu on kyseessä. Yhdistelmäsolu saa tehtävänsä aikana käsiteltävältä oksalta tiedon joko onnistumisesta tai epäonnistumisesta. Jos oksan tehtävä on kesken, se palauttaa suoritus tilan.

Kuten kuviossa 2 nähdään, määrityssolulla on vain yksi lapsisolu. Sen tehtävänä on tehdä jotakin sen omaavalle lapsisolulle, joka tässä tilanteessa on lehtisolu. Määrityssolu voi olla esimerkiksi silmukka, joka toistaisi lehitisolun toiminnan annetuilla arvoilla useampaan kertaan. Toinen vastaava taupaus voisi olla yksinkertainen tiedonmuuttotoiminto lehtisolun parametriarvolle. (Simpson, 2014).

Lehtisolu voidaan ajatella kaikista tehokkaimaksi soluksi, koska se yleensä sisältää varsinaisen toiminnon, johon ehtohen kautta ollaan päädytty. Lehtisolmu voi olla myös linkki aivan uudelle käytöspuulle tai muulle vastaavalle toiminnollisuudelle. Yleensä, jos toiminnolisuudet on jaoteltu järkeviin kokonaisuuksiin, se helpottaa ulkopuolisia henkilöitä ymmärtämään toimintojen kulkua.

Esimerkiksi NPC-hahmon tekoälyllä voisi olla kahdenlaista toimintaa, eteenpäin liikkuminen ja hyppiminen. Kyseinen toiminto voisi soveltua hyvin NPC-hahmoihin tasohyppelypeleissä, joissa tekoälyn tarkoitus olisi seurata pelaajan hahmoa. Pelaaja toki osaa hyppiä esteiden ja rotkojen yli luonnollisesti, mutta NPC-hahmot vaativat tietynlaisen toiminnalisuuden pystyäkseen samaan. Käytetään siis hyvin yksinkertaista käytöspuuta, joka kertoo NPC-hahmolle mitä tehdä missäkin tilanteessa.

Lähdetään tilanteesta, jossa NPC-hahmo on havainnut pelaajahahmon, mutta ei tiedä vielä mitä tekisi asialle. Käydään siis läpi kuvion 3 mukaisen kaavio, joka voi päätyä joko uuden käytöspuun lukuun tai kyseisen käytöspuun epäonnistumiseen. Kohdissa ”Ylitä este” ja ”Liiku pelaaja kohti” NPC-hahmo on päässyt lehteen joka kutsuu uutta puuta ja ”Palauta epäonnistuminen” lopettaa puun toiminnon kunnes sitä kutsutaan taas uudelleen.



Kuvio 3. Esimerkki käytöspuusta.

3.2.3 Tavoitteeseen suuntaava tehtävän suunnittelija

Erittäin monipuolisissa ja edistyneissä NPC-hahmojen tekoälyissä on käytetty tavoitteeseen suuntaavaa tehtävän suunnittelijaa, joka lyhennetään sanaksi GOAP (Goal-Oriented Action Planning). Siinä NPC-hahmon tekoäly pyrkii etsimään mahdollisimman hyvän ratkaisun päästäkseen tavoitteeseen. Menetelmiä, jotka päätyvät samaan tavoitteeseen, voi olla useita, mutta GOAP laskee niistä parhaimman ja lisää sen NPC-hahmon toimintalistaan. (Owens 2014.)

Äärellinen automaatti muistuttaa läheisesti GOAP:a, mutta GOAP:ssa ei ole tehtävien välistä linkitystä. GOAP rakentaa suoritettavien tehtävien kulun ennen kuin NPC-hahmo alkaa suorittaa mitään, toisin kuin äärellinen automaatti vaihtelee tehtäviä ehtojen aktivoiduttua dynaamisesti. GOAP on äärellistä automaattia hitaampi, mutta se on paljon selkeämpi ja parempi ratkaisu monimutkaisten ja moniosaisten tehtäväjonojen suorittamiseen. (Owens 2014.)

GOAP jaottelee tehtävät niiden kustannuksen mukaan. Samaan tavoitteeseen päätyvät tehtävät voivat olla kustannukseltaan erisuuruisia. Joskus jokaista tehtävää ei ole mahdollista suorittaa, koska tehtävään tarvittavat ehdot eivät täyty ja GOAP joutuu valitsemaan kustannukseltaan suurempia tehtäviä päästäkseen tavoitteeseen. Tallaisia ehtoja voisivat olla esimerkiksi pelin maailmassa olevat asiat ja tapahtumat. (Owens 2014.)

Ajatellaan tilanne, jossa NPC-hahmon tavoitteena on rakentaa talo. Talon rakentamiseen tarvitaan aikaa kuusi tuntia. NPC-hahmolla on todella monta tehtävää ja toteutustapoja on monia. Ensin GOAP kirjaa ylös senhetkiset maailman muuttujat, kuten käytössä olevat työkalut, rakennustarvikkeet, kellon ajan ja sään. Työkaluista puuttuu saha, rakennustarvikkeita on vain puolet, kello on puoli yhdeksän aamulla ja sää on kirkas. Ensimmäinen näistä ehdoista hidastaa tavoitteeseen pääsyä puolella ja toinen estää tavoitteeseen pääsyn kokonaan. Tässä tilanteessa GOAP joutuu käyttämään vaihtoehtoisia tehtäviä päästäkseen tavoitteeseen.

Kello kaksitoista päivällä alkaa sade, joka kestää kaksi tuntia. Tuolloin NPC-hahmo ei voi rakentaa taloa, mutta hän voi ajaa autolla läheiseen kylään ostamaan puuttuvia rakennustarvikkeita ja sahan. Matka kestää yhteensä kolme tuntia. GOAP laskee optimaalisimmaksi suoritusjonoksi rakentaa taloa puolella nopeudella kello yhteentoista asti ja lähteä ostoksille tuntia ennen sateen alkamista. Tavoitteeseen kuluva kustannus on kymmenen tuntia. Jos NPC-hahmo olisi lähtenyt suoraan kauppaan ja palannut sateen alkaessa, kustannus olisi ollut yksitoista tuntia. Toimintatapoja voisi olla myös useita muita. Esimerkiksi NPC-hahmon olisi voinut hakea tarvikkeet kävellen, mutta olisi joutunut tekemään viisikymmentä hakua, joista jokainen olisi kustantanut kolmetoista tuntia.

Tavoitteeseen suuntaavaa tehtävän suunnittelija voi alkaa äärellisen automaatin tehtävästä, kuten NPC-hahmon tullessa toimettomaan tilaan. Kuitenkaan äärellisen automaatin, joka suorittaa GOAP:a, ei tarvitse olla laaja, koska GOAP hoitaa tarvittavat yksityiskohdat. Esimerkkinä äärellinen automaatti voisi suorittaa NPC-hahmon toimettomuus-, liikkumis- ja tehtävänsuoritustilan. GOAP joutuu kuitenkin itse tarkistamaan maailmassa tapahtuvia muuttujia, jos ne esimerkiksi vaihtuvat. (Owens 2014.)

GOAP:n tyylinen tekoäly NPC-hahmoilla löytyy esimerkiksi ”F.E.A.R”-nimisestä kauhupelisarjasta.

3.3 Oppiminen

Tekoälyn oppiminen voidaan käsittää monella eri tavalla. Varsinainen virheistä oppiminen tai muuten ihmismäinen oppiminen on erittäin harvinaista heikolla tekoälyllä, koska se vaatisi valtavasti muistia sekä laskentatehoa. Kuitenkin oppimista voidaan simuloida valmiiksi ohjelmoiduilla muistipalikoilla, joita tekoäly voi saada tiettyjen ehtojen täyttyessä. Näin pelaajalle luodaan kuva, että tekoäly olisi oppinut jotain, vaikka kyseessä onkin vain illuusio älyllisestä ajattelusta. NPC-hahmoilla voi olla myös omat muistikirjastonsa, joissa ne voivat pitää tietoa esimerkiksi juonen eri vaiheista tai omista NPC-kohtaisista tiedoista. (Indika 2011.)

Yksi esimerkki oppimisesta olisi, jos pelaajahahmo olisi tavannut tietyn NPC-hahmon aiemmin pelissä ja auttanut häntä suorittamalla hänelle jonkin tehtävän. Uudelleen tapaamisessa NPC-hahmo tuntisi pelaajan jo ennestään ja reagoisi tapaamiseen eri tavalla kuin ensimmäisellä kerralla. Jos pelaajalla olisi vielä mahdollisuus vaikuttaa pelaajan ja NPC-hahmon väliseen suhteeseen positiivisella, negatiivisella tai neutraalilla tavalla, se vaikuttaisi myös uudelleen tapaamisessa NPC-hahmon reagointiin. Monipuolisten uusia toiminnollisuuksia avaavien vaihtoehtojen toteuttaminen ei kuitenkaan ole kovin yksinkertaista. Esimerkiksi jo muutamien vaihtoehtojen takana, saattaa olla koodillisesti hyvinkin työläs toteutus.

Yksi helpoimmista tavoista muodostaa muistikirjastoja on käyttää XML-datapankkia, koska tiedon luku ja tallennus onnistuu erittäin joutuisasti käyttäen XML:ää. Kuitenkin XML rajoittuu vain teksti- ja numeraalisiin tiedostoelementteihin, eikä voi sisältää esimerkiksi koodillista toimintaa. Siksipä sitä käytetään useimmiten dialogien tai muuttuvien arvojen tallentamiseen. Kuviossa 5 on esimerkki, NPC-hahmon mahdollisista toiminnasta pelaaja kohtaan käyttäen XML-kieltä.

Pelaajan tavatessa NPC-hahmo ensimmäistä kertaa, tämä reagoi pelaajaa kohtaa vain positiivisella tavalla. Kun taas he tapaavat seuraavan kerran, lähtökohdat voivat olla useanlaisia. Vaikka dialogia on valmiiksi kirjoitettua, pelaajalle luodaan illuusio, että NPC-hahmo on oikeasti muistanut hänen tekonsa. Kooditasolla NPC-hahmo saattaa vain tietää kokonaislukumuuttujan avulla oliko ensimmäisen tapaamisen tulos positiivinen, neutraali vai negatiivinen ja sen perusteella hakee XML tiedostosta tarvittavat tiedot.

```
<Vaihtoehdot>
    <Tapaaminen1>
        <Positiivinen>
            <Tervehdys>Hyvää päivää</Tervehdys>
            <Kysymys>Auttaisitteko minua kantamaan nämä laatikot?</Kysymys>
        </Positiivinen>
    </Tapaaminen1>
    <Tapaaminen2>
        <Positiivinen>
            <Tervehdys>No hei taas ystäväni!</Tervehdys>
            <Palkinto>Kiitoksia kun autoit minua silloin, tässä 10 kultarahaa</Palkinto>
        </Positiivinen>
        <Neutraali>
            <Tervehdys>Terve taas</Tervehdys>
            <Kysymys>Ehditteko auttamaan minua nyt?</Kysymys>
        </Neutraali>
        <Negatiivinen>
            <Tervehdys>Ai te taas</Tervehdys>
            <Lause>Rikoin selkäni, kun yritin kantaa laatikoita yksin</Lause>
        </Negatiivinen>
    </Tapaaminen2>
</Vaihtoehdot>
```

Kuvio 5. XML-datapankki.

Toinen tapa käsittää oppiminen varsinkin NPC-hahmoilla voisi esimerkiksi viitata taitoihin, joita hahmot oppisivat saavutettuaan tiettyjä tasoja. Tämä kuitenkaan ei ole varsinaista oppimista, vain hyvin yksinkertaisesti valmiiksi tehtyjen ohjelmointipalikoiden lisäämistä NPC-hahmon taitolistaan. Nämä ohjelmistopalikat saattavat sisältää erittäinkin monipuolista toimintaa. Esimerkiksi jos NPC-hahmo oppisi lentämään, tulisi hänen periä kokonainen lentoluokka, joka taas hakisi NPC:n omasta muistikirjastosta tarvittavia arvoja, kuten maksimilentokorkeuden ja nopeuden. Tämä lentoluokka voisi sisältää ennalta määritettyjä metodeja, joita NPC sitten voisi vapaasti käyttää. Nämä voisivat olla esimerkiksi lentoonlähtö, lentäminen ja laskeutuminen.

3.4 Vuorovaikutus

Jos NPC-hahmo on ihminen tai edes olio, joka kykenee kommunikoimaan pelaajan kanssa, on hyvin todennäköistä, että tämä hahmo sisältään jonkin muotoisen keskustelukirjaston. Yleensä puhutaan sanasta dialogipuu, jossa keskustelu voi edetä moneen eri suuntaan ja NPC hakee tarvittavan vastauksen omasta keskustelukirjastostaan riippuen pelaajan valitsemasta lauseesta.

Tämä on kuitenkin hyvin suljettua keskustelua ja se ei jätä yleensä pelaajalle paljoa valinnan varaa. Tällöin oikeiden vaihtoehtojen valitseminen on jopa turhankin helppoa ja se saattaa turhauttaa pelaajaa. Kuitenkin huolella tehdyissä peleissä, joissa keskusteluihin on paneuduttu tarkasti, pelaaja ei välttämättä osaa arvata oikeiden vaihtoehtojen kulkua. Joskus jopa itsestään selvät oikeat vastaukset saattavat aiheuttaa täysin päinvastaisen efektin, mutta ne taas tuovat peliin huomattavasti lisää pohdittavaa. Kuviossa 6 on esimerkki dialogipuun ensimmäisestä juuresta:

```
Vartija: Hei! Minne te luutte menevänne?
Sisäänpääsy on vain valtuutetuille henkilöille!

A.      "Aivan, tässä on kulkulupani."

B.      "öhh.. tosiaan taisin unohtaa kulkulupani kotiin."

C.      "(Lähde juoksemaan karkuun minkä kintuistasi pääset)"
```

Kuvio 6. Dialogipuun juuri.

Toinen tapa luoda mielenkiintoisia keskusteluja on käyttää tekstin järjestelymoottoria. Nykypäivänä näitä ei enää usein näe, koska niiden käyttö on sijoittunut suurimmaksi osaksi 1980-luvun tekstipohjaisiin rooliseikkailupeleihin. Teoriassa ne ovat todella toimivia ja monipuolisia, mutta toimiakseen ne vaativat erittäin monipuolisen kirjaston. Yleisesti tekstinjärjestelymoottorit tunnistavat pelaajan syöttämästä lauseista verbin ja substantiivin, joista se vertaa niitä oikeisiin vastauksiin. Haittapuolena on toistuva väärin vastaaminen liian vaikeisiin kysymyksiin, koska se äkkiä turhauttaa pelaajaa. Tekstinkäsittelymoottorit kärsivät samasta ongelmasta kuin äänentunnistus ohjelmatkin, ne eivät osaa käsitellä virheellisiä lauseita kuten ihmisaivot. (Stanford 2014.)

Toisaalta hyvin suunnitellut ja monipuolisesti rakennettu tekstinjärjestelymoottori saattaa antaa pelaajalle vinkkiä mahdollisesta vastauksesta. Toinen auttava ominaisuus voi olla tunnistettujen sanojen pohjalta rakentuva vihje, joka auttaa pelaajan arvaamaan loputkin tarvittavat sanat. Mitä edistyneempi moottori, sitä enemmän mahdollisia vaihtoehtoja jokaiseen eri tilanteeseen löytyy. (Stanford 2014.)

Tekstinjärjestelymoottoria on yleisemmin käytetty pelaajan ja itse pelin väliseen kommunikointiin, mutta se soveltuu myös NPC-hahmon ja pelaajan väliseen dialogiin. Kuviossa 7 on esimerkki tekstinjärjestelymoottorilla toteutetusta pelaajan ja NPC-hahmon välisestä tapahtumasta:

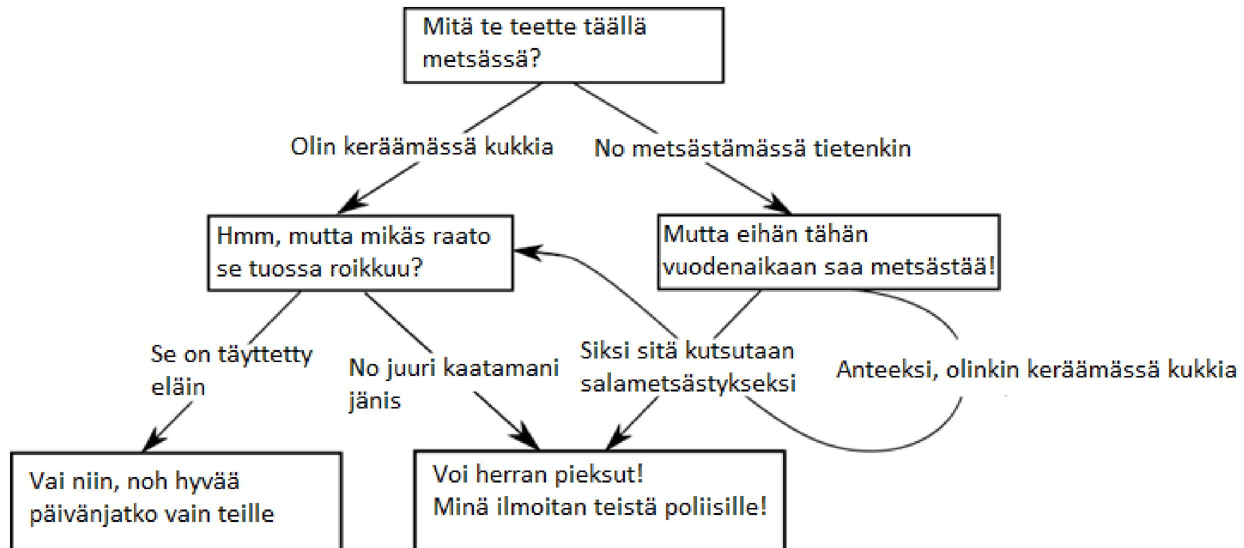
```
Kauppias: "Kappas, vanha ystäväni Masahan se siinä."

TERVEHDI KAUPPIASTA

Kauppias: "En aivan ymmärrä, mitä tarkoitat."
Hän varmaan odottaa sinulta jonkin muotoista tervehdystä.

TERVEHDI HÄNTÄ

Masa: "No, heipä hei. Emme ole nähneetkään toviin."

Kauppias: "Olisitko vailla jotain valikoimastani?"

(ja dialogi jatkuu)
```

Kuvio 7. Tekstinjärjestelymoottori.

Monissa uusissa peleissä pelaajan ja tekoälyn väliset keskustelut on synkronoitu luomalla juoneen sekä NPC-hahmojen persoonaan vaikuttavia muuttujia. Toisin sanoen pelaajan teot muokkaavat sitä, miten NPC-hahmot suhtautuvat pelaajaan. Useimmiten keskustelut jaetaan kolmeen kategoriaan: positiiviseen, neutraaliin ja negatiiviseen. Tällöin yksi helpommista tavoista muodostaa keskustelut on käyttää jotakin skriptauskieltä luomalla valmiiksi muodostettuja keskustelu dialogiapuita. Näistä kielistä tunnetuimpia on Lua-skriptaus sekä XML, jota käytettiin myös NPC-hahmojen oppimisessa.

Keskustelun toiminnollisuus toimii periaatteella, jossa jokaisen valinnan jälkeen avautuu uusia valintoja, kunnes lopulta keskustelu päätyy. Liian monipuolisen keskustelun tapahtuessa jokainen valinta ei välttämättä luo uusia valintoja, vaan saattaa johdattaa keskustelun esimerkiksi takaisin aikaisempaan kohtaan. On myös hyvin tavallista, että useampi valinta päätyy lopulta samaan lopputulokseen, koska pitkät keskustelut saattavat viedä valtavasti muistia varsinkin, jos ne ovat valmiiksi ääninäyteltyjä.

Kuviossa 8 on keskustelu, jossa keskustelu päättyy joko neutraalilla tai negatiivisella tavalla huolimatta siitä, mitä pelaaja päättää NPC-hahmolle sanoa. Jokainen kohta ei avaa sille omaa seurausta, vaan rakenteessa on osattu kekseliäästi kierrättää samoja kohtia luoden yksinkertainen ja monipuolinen dialogi.



Kuvio 8. Esimerkki pelaajan ja NPC-hahmon keskustelusta.

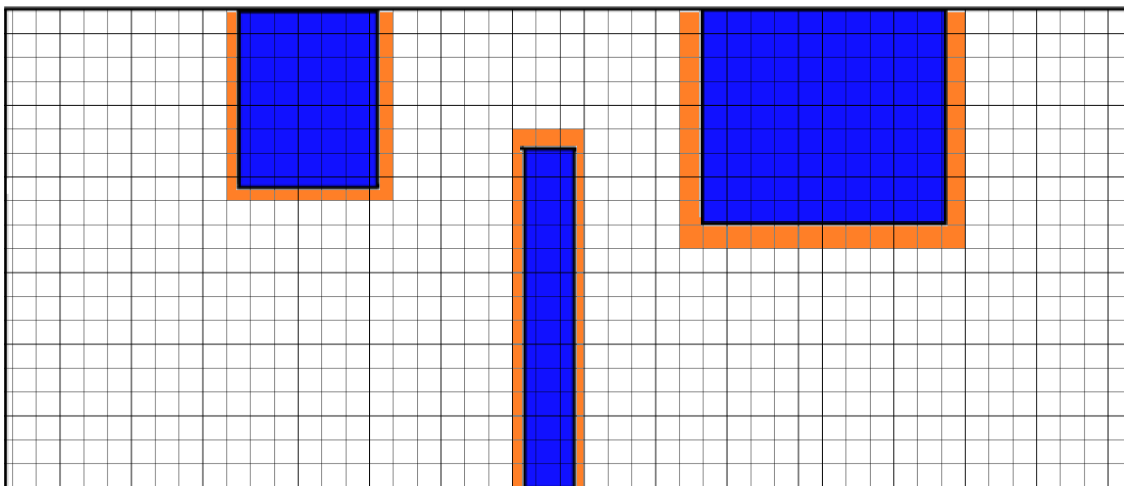
Joskus on mahdollista, että keskustelujen loppuessa päädytään johonkin tapahtumaan. Tallaisia tilanteita voisi olla esimerkiksi pelaajan ärsyttäessä NPC-hahmoa pisteeseen jossa ystävälliset NPC-hahmot muuttuvat pelaajan vihollisiksi. Jos taas keskustelu ei aiheuta mitään konkreettista tapahtumaa, voi se useasti kirjata tekoälylle ylös muistiin tietoa, joka vaikuttaa tuleviin keskusteluihin. Tällä tavalla pelin juoni voi muuttua erilaiseksi, ja se tuo pelaajalle paljon enemmän valinnanvaraa pelata peli monella tapaa läpi.

3.5 Reitinhaku

Yleisesti reitinhaun ideana on löytää mahdollisimman lyhyt reitti kahden pisteen väliltä käyttäen tiettyä laskenta-algoritmia (Dijkstra 1959). Tämä on yksi tärkeimmistä ominaisuuksista, joita NPC-hahmot tarvitsevat, sekä se on myös yksi vanhimmista tekoälyn toiminnoista peleissä. Kenties tunnetuin ruudukkotyylinen reitinhakumenetelmäalgoritmi on nimeltä A* (A star), joka luotiin jo vuonna 1968. Tätä asiaa käsitellään tarkemmin opinnäytetyön toiminnallisessa osuudessa.

Yleisesti ottaen reitinhaulle on lukemattomia toteutustapoja, koska kaikki voivat muokata jo luoduista algoritmeista omansa tai yksikertaisesti keksiä aivan uuden toteutustavan. Mutta mitä varsinaisesti tarvitaan toimivan reitinhaun toteuttamiseen? Pelkkä toimivan algoritmin luonti ei aina riitä, vaan tarvitaan myös elementtejä, jotka kommunikoivat sen kanssa. Esimerkiksi maasto tai edes jonkin tapainen alue, missä liikkuminen on mahdollista, ovat yleisiä elementtejä kommunikointiin reitinhakualgoritmin kanssa.

Alueiden määrittäminen on yksi tapa aloittaa reitinhakumekaniikan luonti. Lähtökohtana yleensä on jonkin tapainen maasto tai alue, kuten lattia. Alueella saattaa olla kohtia, jossa liikkuminen on mahdotonta, kuten seinät, kivet, puut tai liikkuvat objektit. Nämä alueet tulee siis joko jättää laskematta tai merkitä alueiksi, joissa NPC-hahmot eivät voi liikkua. Yksi helpoimmista tavoista on lähteä luomaan ruudukkopohjaista aluetta, missä jokaisen lasketun pisteen tai laatikon kohdalla tarkistetaan kyseinen alue mahdollisilta törmäyksiltä. Kuviossa 9 on esimerkki ruudukkopohjaisesta alueesta, jossa siniset laatikot ovat esteitä ja oranssit alueet suljettuja solmuja. Kun ruudukko on luotu, sitä voidaan halutessaan lähteä soveltamaan eteenpäin. (Epic Games 2014b.)



Kuvio 9. Laskettu ruudukko.

Dynaamisten objektien liikkuminen ruudukkopohjaisella alueella voidaan toteuttaa monilla eri tavoilla. Itse olen käyttänyt menetelmää jossa dynaamiset objektit tarkistavat peittämänsä alueen liikuttuaan ruudukossa yhden pykälän verran. Näin ruudukon luoneen koodin ei tarvitse määritysvaiheen jälkeen päivittää mitään, vaan se hoidetaan ulkopuolelta käsin.

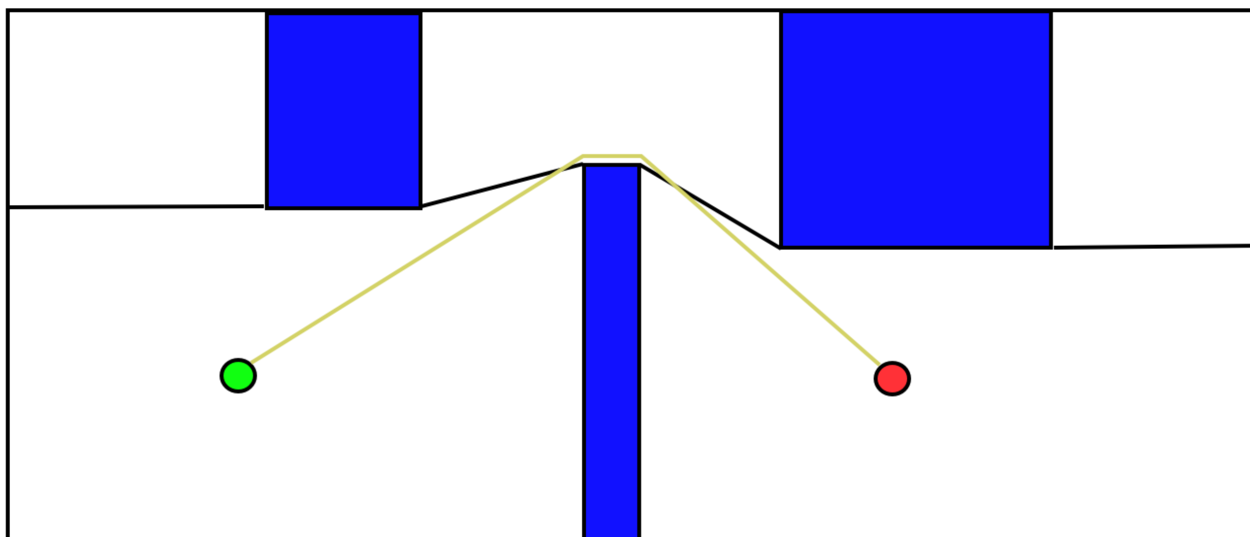
Uusimmat pelimoottorit käyttävät reitinhakunaan navigaatioverkkoa (navigationmesh). Sen toiminta on paljon tarkempaa kuin ruudukkopohjaisten reitinhakumenetelmien, mutta sen toteuttaminen vaatii puolestaan hiukan enemmän laskentatehoa. Navigaatioverkkoa käyttämällä päästään myös paljon realistisimpiin reitteihin, mikä nähdään tekoälyn liikkuminen sulavuudessa. Kuitenkin navigaatioverkon laskentatehon raskauden vuoksi dynaamista navigaatioverkkoa ei löydy kuin uusimmista pelimoottoreista, kuten Unreal Engine 4:stä, mikä on Epic Games nimisen yrityksen oma pelimoottori. Mutta kuinka Unreal Enginessä navigaatioverkko on toteutettu?

Unreal Enginen navigaatioverkko on toteutettu käyttäen ruudukkopohjaista systeemiä sen pohjana (Epic Games 2014b). Kun ruudukko on laskettu, tulee seuraavaksi yhdistää pisteet luoden mahdollisimman suuria neliöitä, jotta alue muuttuisi mahdollisimman yksinkertaiseksi. Lopuksi neliöistä poistetaan reunat, jotka osuvat toisiin neliöihin, ja neliöihin jätetään vain ääriviivat suljettujen alueiden reunoista. On toki mahdollista, että alueella on esteitä, jotka ovat ruudukon mukaan vinossa. Tällöin voi muodostua sahalaitareunoja reunoja, jotka hankaloittaisivat reitinhakua. Jotta niistä päästäisiin eroon, tulee samansuuntaiset sahalaitareunat yhdistää yhtenäiseksi viivaksi.

Oheisessa kuviossa 10, siniset laatikot ovat esteitä ja valkoiset alueet laskettua navigaatioverkkoa. Keskimmäinen este on aiheuttanut sahalaitaisia kulmia, mutta koska ne ovat samansuuntaisia, niistä on voitu muodostaa suora viiva.

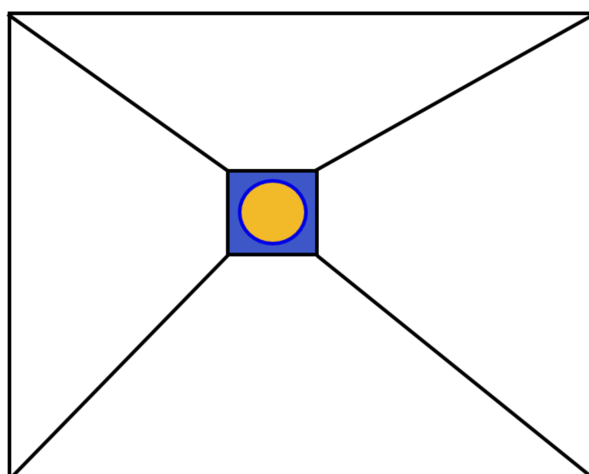
Kuvio 10. Laskettu navigaatioverkko.

Seuraavaksi NPC-hahmo laitetaan keskimmäisen esteen vasemmalle puolelle ja kohteen, johon NPC-hahmon tulee liikkua, esteen oikealle puolelle. NPC-hahmo tarvitsee algoritmin lyhimmän reitin laskemiseen ja tässä tilanteessa se on laskenut keskimmäisen esteen kaksi ylintä pistettä reittiinsä. Kuten kuviossa 11 näkyy, vihreää pistettä esittävä NPC-hahmo on luonut reitin, joka näkyy keltaisena viivana, kohti punaista päätepistettä.



Kuvio 11. Laskettu reitti navigaatioverkossa.

Kun ruudukkopohjaisen reitinhakumenetelmän dynaaminen objekti kirjasi kaikki peittämänsä pisteet suljetuiksi, ei tällaista toimenpidettä enää tarvita navigaatioverkkoa käytettäessä. Objektin tarvitsee vain tietää, mihin alueisiin se vaikuttaa ja niiden avulla rajata itselleen alue, mihin tekoäly ei voi liikkua. Kuviossa 12 on esimerkki kyseisestä tilanteesta.



Kuvio 12. Dynaaminen objekti.

Oranssi piste viittaa dynaamisen objektiin, valkoinen liikuttavaan alueeseen ja sininen alueeseen missä ei voi liikkua. Toki kuvan kyseinen dynaaminen objekti ei itse huomioi tekemäänsä estettä, koska muuten se olisi jumissa itsessään.

3.6 NPC tekoälypelaajalla

Käsite tekoälypelaajana on huomattavasti laajempi kuin yksittäisten NPC-hahmojen toiminta. Siinä tekoälyä verrataan ihmispelaajaan, jolla on samat säännöt ja toiminnollisuudet pelin kannalta. Tekoäly voi kontrolloida useampaa NPC-hahmoa ja on kykenevä havaitsemaan jokaisen muuttujan, jonka se havaitsee omien hahmojensa kautta. Tekoälypelaaja voi olla myös yksittäinen NPC-hahmo joka yrittää simuloida ihmispelaajan tapaisia toimintoja tekemällä tahallisesti tietyn verran virheitä riippuen esimerkiksi tekoälyn vaikeustasosta.

Tekoälypelaaja vaatii runsaasti tietokoneen laskentatehoa, varsinkin jos kyseessä on useitten kymmenien tai jopa satojen NPC-hahmojen kontrollointi. On myös hyvin tärkeää, että tekoälyn täytyy ottaa huomioon toisten pelaajien toiminnat ja laskea mahdollisimman toimiva ratkaisu jokaiseen tilanteeseen. Tämän tyyppistä tekoälyä voidaan kutsua strategiatekoälyksi. (Ertürk 2009.)

Useimmiten strategiapelien toimintaperiaate koostuu tukikohdan ja oman armeijan rakentamisesta sekä ohjailusta. Kuitenkaan pelkkä yksinkertainen rakentelu tekoälyllä ei yleensä riitä, vaan useimmiten johtaa hyvin nopeaan häviöön. Siksipä tekoäly tulee ohjelmoida ajattelemaan kuten ihmispelaaja tai jopa paremmin. Ei ole kuitenkaan helppoa luoda korkeatasoista tekoälypelaajaa, jos itse ohjelmoija ei tiedä miten peliä pelataan. Tällöin pelintekijät saattavat hiukan huijata ja antaa tekoälylle ominaisuuksia, joita ihmispelaaja ei voi saada.

Hyvin suunnitellulla tekoälyllä voi olla äärellisen automaatin tai käytöspuun tapainen toimintamalli, joka toimii hyvin samalla periaatteella kuin yksittäisen NPC-hahmon käyttämä toimintamalli. Koska ihmispelaajankin täytyy osata ottaa huomioon pelin eri vaiheet, niin miksipä ei tekoälynkin. Peli voidaan jakaa yleensä kolmeen perus vaiheeseen: alku-, keski-, ja loppupeliin, joilla taas kaikilla on omat aliluokkansa, jotka vaikuttavat tekoälyn valintoihin. Esimerkiksi jos ihmispelaaja päättää tehdä äkillisen alkupelin rynnäkön, täytyy tekoälyn osata vaihtaa tilakseen alkupelin puolustava asema. Tällöin tekoäly ei esimerkiksi lähde kehittämään rakennuksiaan suuremmille tasoille, vaan kasvattaa puolustustaan parhaansa mukaan.

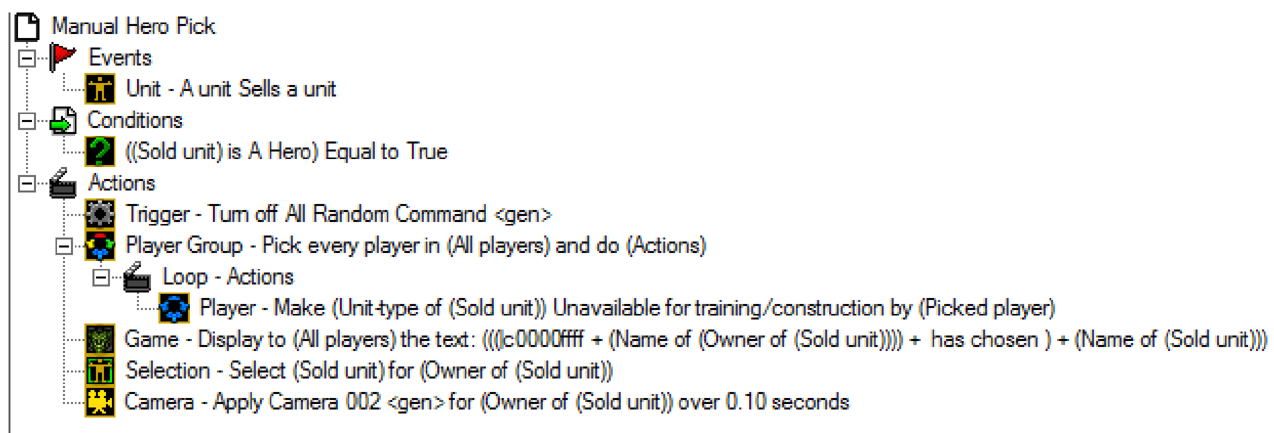
Strategiapeleissä NPC-hahmot voivat edelleen noudattaa niille sisäänkirjoitettua omakohtaista tekoälyä, mutta tekoälypelaajan ottaessa ne omaan kontrolliinsa ne yleensä ohitetaan tekoälypelaajan omiin komentoihin. Sama periaate toimii myös ihmispelaajan hahmoille, koska ilman sisäänkirjoitettua tekoälyä hahmot, joita ei ole komennettu tekemään mitään, eivät myös tekisi mitään vaikka vihollinen hyökkäisi niiden kimppuun.

4 NPC-ohjelmointi

4.1 Tieni tekoälyn parissa

Olin jo nuorena kiinnostunut siitä, kuinka tekoäly toimi osana tietokonepelejä. Kesti kuitenkin useita vuosia, kunnes törmäsin peliin jossa oli mahdollista rakentaa oma tekoäly käyttäen sen omaa pelimoottoria. Pelin nimi oli Warcraft 3 ja sen käyttämä kenttäeditori oli nimeltään World Editor.

Kuitenkaan suora C-kielen skriptaus ei toiminut tässä editorissa, vaan jouduin rakentamaan omat skriptini valmiita luokkia käyttäen. Kuviossa 13 on lyhyt esimerkki siitä, kuinka kenttäeditorin skriptaus tehtiin. Kohta ”Events” oli tapahtuma joka aktivoi skriptin ”Actions”-osuuden jos ”Conditions” eli ehdot täyttyivät.



Kuvio 13. Osio World Editorilla tehdystä koodistani.

Koska en itse ollut kovinkaan taitava pelaaja, päätin tehdä oman strategiatekoälyn ja yrittää voittaa sillä muita ihmispelaajia. Vastaan tuli kuitenkin ongelmia pelimoottorin sisäänkirjoitettujen skriptien kautta, koska ne ohittivat minun omani. Tekoälyni onnistui kuitenkin voittamaan pelin omat tekoälyt vaikeimmalla vaikeustasolla, mutta olin kuitenkin syvästi pettynyt kun tekoälyni ei päihittänyt ihmispelaajia. Rakensin myös lukiossa ollessani toisen tekoälyn käyttäen World Editoria. Tällä kerralla en tehnytkään enää strategiatekoälyä vaan yksittäisten NPC-hahmojen toimintaa. Kyseessä oli viiden ihmisen pelattava peli, jossa vihollistekoäly yritti vallata kaupunkia, jota taas pelaajat sekä kolme NPC-hahmoa yrittivät puolustaa.

Tavoitteenani oli saada näistä kolmesta pelaajien puolella olevista NPC-hahmoista mahdollisimman realistisia ohjelmoimalla niille käyttäytymisiä, joita ihmispelaajatkin suorittivat. Haastavinta oli saada NPC-hahmot reagoimaan pelin eri vaiheissa samoin kuin ihmispelaajat. Oli kuitenkin lähes mahdotonta keksiä toiminnollisuutta, jolla NPC-hahmot toimivat täydellisessä synkroniassa ihmispelaajien kanssa. Yritin myös luoda peliin tietynlaista tasapainoa luomalla tekoälyn pelistä lähteneille pelaajille. Ongelmana kuitenkin oli NPC-hahmojen taitojen käyttäminen, koska pelissäni oli yli kaksikymmentä pelattavaa hahmoa, joilla jokaisella oli keskimäärin kolme aktiivista taitoa. Onneksi kuitenkin Warcraftin sisäänkirjoitettu tekoäly osasi käyttää osaa niistä automaattisesti.

Seuraavaksi siirryin Karelia-ammattikorkeakouluun, jossa päätin erikoistua peliohjelmointiin. Annoin itselleni tavoitteeksi ohjelmoida shakkitekoälyn. Ensimmäisen vuoden lopulla päästyäni olio-ohjelmoinnin kurssilta päätin aloittaa shakkitekoälyni C#-ohjelmointikielellä. Tein projektia noin kolme kuukautta ja sain tekoälyni laskemaan kolme siirtoa eteenpäin. Valitettavasti siirtojen laskemiseen meni keskimäärin 30 sekuntia ja päätin, että aion toteuttaa tekoälyn alusta loppuun uudestaan, kun olen oppinut lisää ohjelmoinnista.

Opin kuitenkin shakkitekoälyn peruslogiikkaa ja shakkinappuloiden monia eri arvoja. Itse näistä aluksi käytin Yevgeny Gikin arvoja, mutta myöhemmin vaihdoin kuitenkin yleiseen standardiin. Tällä hetkellä se koostuu arvoista, joissa sotilas vastaa yhtä, ratsu ja lähetti kolmea, torni viittä, kuningatar yhdeksää ja lopuksi kuningas neljää. Mutta miksi kuninkaan arvo on tornia ja kuningatarta pienempi?

Se johtuu puhtaasti kuninkaan taistelukyvystä, jossa on otettu huomioon sen liikkuvuus ja mahdollisuus hyökätä viholliseen. Kuningas on korvaamaton, koska shakkimatti-tilanteeseen joutuvan pelaajan kuningas häviää pelin. Tästä muodostuikin erittäin hankala kysymys: kuinka tekoäly käsittelee äärettömän arvokasta nappulaa, jonka todellinen arvo on vain neljä?

Kesäkuussa 2013 aloitin työharjoitteluni, ja silloin pääsin heti työskentelemään tekoälyn parissa. Ensimmäinen projektimme oli 3D ”Hack&Slash”-tyylinen peli, jossa pelaaja seikkaili tyrmissä taistellen epäkuolleita vastaan. Minun tehtäväni oli ohjelmoida vihollis-NPC-hahmojen tekoäly, mutta olin vielä täysin kokematon Unity-pelimoottorin käytössä ja oppimiseen meni aikaa.

Kovan työn tuloksena sain ohjelmoitua kolme vihollis-NPC-hahmoa, joilla oli suhteellisen monipuolinen toiminnollisuus. Tekoäly osasi muodostaa vihollisista joukkoja jotka toimivat ja liikkuivat ryhmänä. Laitoin myös viholliset pakenemaan ja etsimään uutta ryhmää, jos toverien lukumäärä vierellä alkoi tippua alhaiseksi.

Pohdin pitkään tekoälyni peruslogiikkaa, jossa minulla oli kaksi erilaista vaihtoehtoa. Toinen näistä oli tyhjä objekti, joka hoitaisi kaiken vihollislogiikan yksin välittämättä yksilöllisten vihollisten havainnoista. Tämä koitui kuitenkin erittäin raskaaksi logiikaksi, joka hidasti itse peliä sekä tiputti vihollisten reagointikykyä huomattavasti. Päädyin siis luomaan jokaiselle viholliselle oman komponentin, joka hoiti niiden yksilölliset toiminnat sekä kommunikoi toisten vihollisten komponenttien kanssa. Tämä kevensi peliä huomattavasti ja toi siihen paljon monipuolisempaa tekoälyn toimintaa.

Tammikuussa 2014 osallistuin Game Jam nimiseen tapahtumaan, missä pelinkehittäjät ympäri maailmaa loivat pelejä kaikille yhteisestä aiheesta. Tapahtuma kesti vain kaksi vuorokautta ja aiheeksi oli annettu ”We don't see things as they are, we see them as we are”. Ryhmämme päätti luoda pelin, jossa alkoholiin riippuvainen henkilö yritti päästä läheiseen viinakauppaan ja pelaajan tehtävänä oli viivästyttää NPC-hahmon pääsyä mahdollisimman pitkään.

Käytössämme oli Unity-pelimoottorin ilmaisversio, jossa dynaamisen navigaatioverkon käyttö oli estetty. Tuolloin sain idean luoda oman dynaamisen reitinhakumenetelmän. Valitettavasti en ehtinyt luoda kahdessa vuorokaudessa toimivaa menetelmää ja pelimme jäi hyvin keskeneräiseksi. Päätin kuitenkin jatkaa reitinhakumenetelmäni työstämistä ja kahden kuukauden päästä olin luonut toimivan dynaamisen reitinhakumenetelmän Unity-pelimoottorilla.

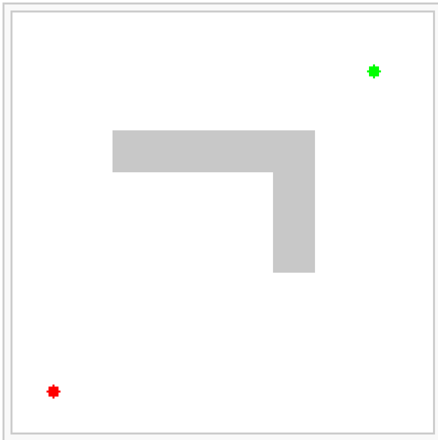
4.2 Reitinhaku A* Unity-pelimoottorilla

4.2.1 A*-algoritmin pohjustaminen

A-tähti-algoritmi perustuu vuonna 1959 julkaistuun Dijkstran algoritmiin, joka selvittää mahdollisimman lyhyen reitin kahden pisteen väliltä. Toisin kuin A-tähti, Dijkstran algoritmi ei käytä heuristista etsintä logiikkaa, jossa reitinetsintä pyritään tekemään mahdollisimman nopeasti. Vaikkakin Dijkstran algoritmi on erittäin tarkka, se tarvitsee huomattavasti enemmän aikaa laskemiseen kuin A-tähti-algoritmi. (Dijkstra 1959.)

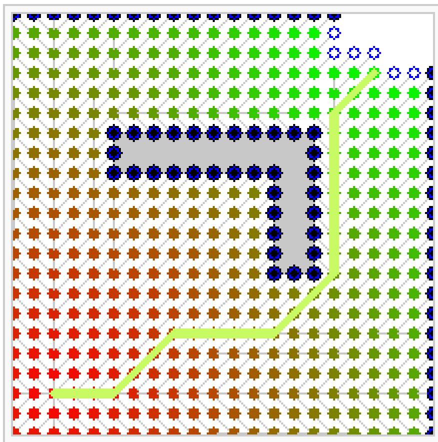
A-tähti-reitinhakumenetelmässä reittiä lähdetään laskemaan lähtöpisteestä ”paras ensin”-menetelmällä, jossa kahdeksasta lähimmästä pisteestä lasketaan paras mahdollinen piste. Paras piste ei välttämättä ole aina piste josta on lyhin matka kohteeseen, vaan pisteen arvoon lasketaan myös, monellako siirrolla siihen päästään. Jos taas tätä verrataan Dijkstran algoritmiin, jossa alkupisteestä lähdetään laskemaan järjestelmällisesti lähintä aukinaista solmua, huomataan A-tähden olevan selvästi nopeampi laskentamenetelmä.

Molempien algoritmien lopputulokset on esitelty kuvioissa 15 ja 16. Kuviossa 14 punainen piste viittaa laskennan alkupisteeseen ja vihreä piste maaliin, johon algoritmit pyrkivät pääsemään. Harmaa este on alue joka estää liikkumisen, joten molempien algoritmien tulee väistää se parhaalla mahdollisella tavalla. Kuviossa 15 on lopputulos Dijkstran algoritmista. Reunoilla sekä harmaan esteen ympärillä olevat siniset pisteet viittaavat solmuihin, joihin tekoäly ei voi liikkua. Tästä voidaan selvästi nähdä Dijkstran algoritmin kattavan huomattavasti suuremman alueen kuin kuviossa 16 esiintyvän A-tähti-algoritmin.

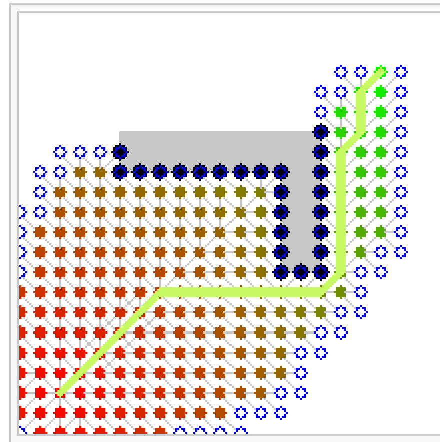


Kuvio 14. Alku tilanne (Wikipedia 2014a).

Kuvioissa 15 ja 16 näkyvä keltainen viiva viittaa reittiin, jonka algoritmi on laskenut. Reitti ei kuitenkaan ole kovin realistinen kummallakaan algoritmilla, joten jos sitä haluttaisiin käyttää NPC-hahmon reittinä, tulisi sitä käyttää vain suuntaa antavina pisteinä.



Kuvio 15.



Kuvio 16.

Kuvio 15. Dijkstran algoritmi (Wikipedia 2014b).

Kuvio 16. A-tähti-algoritmi (Wikipedia 2014a).

4.2.2 A*-toteutus

Tässä luvussa käsitellään sitä, kuinka A* rakennetaan alusta loppuun käyttäen Unity-pelimoottoria. Perusrakenne voidaan jakaa kahteen pääosaan: graafin ja A*-algoritmin luontiin. Tavoitteena on aluksi rakentaa kolmiulotteiseen maailmaan graafi, joka kulkee maata pitkin huomioiden korkeussuhteita sekä mahdollisia staattisia ja dynaamisia objekteja.

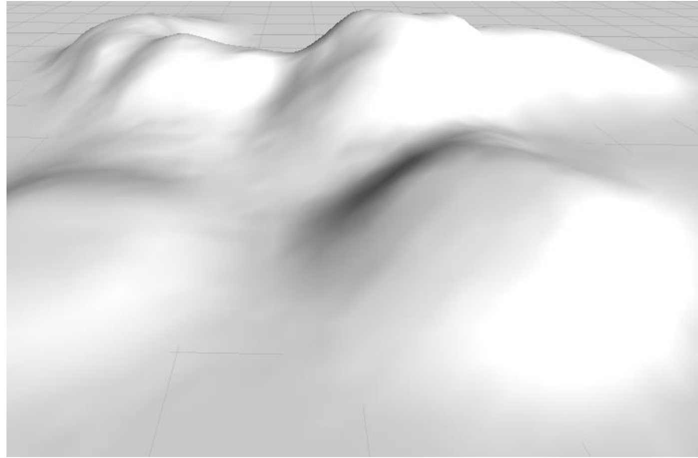
4.2.3 Visuaalisen graafin luonti

Jotta työskentely olisi sujuvampaa ja itse koodi kevyempää, tulee luoda kaksi graafia. Ensimmäinen näistä on visuaalinen graafi, joka ei varsinaisessa pelissä tule pelaajille olemaan näkyvissä. Sen tarkoituksena on vain helpottaa ohjelmoijien työtä. Visuaalinen graafi piirtää maaston päälle ruudukon ja lisää punaisia laatikoita kohtiin, joissa liikkuminen on kiellettyä. Graafi itsessään koostuu vertekseistä, joista luodaan kolmioita antamalla niille uv-kartta ja normaalit. Liitteessä 1 nähdään c#-koodin, jolla visuaalinen graafi muodostetaan.

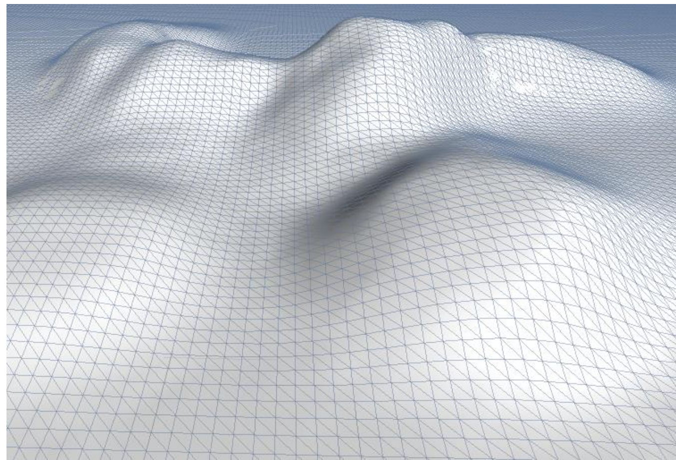
Koodin alkuvaiheessa määritetään tarvittavien muuttujien arvot hakemalle ne aktiivisen maaston tiedoista. Graafin osia täytyy myös alustaa tarvittava määrä, koska Unity-pelimoottori tukee objekteja, jotka sisältävät enintään 65000 verteksiä. Kun kaikki tarvittavat alustukset ovat valmiita, luodaan kaksi for-silmukkaa sisäkkäin, jotka käyvät maaston joka toisen verteksin pysty- ja vaakasuuntaan luomalla kahdesta kolmiosta neliön. Verteksimäärän täytyttyä otetaan käsittelyyn tyhjä graafi ja kutsutaan sen ”CreateMesh”-komponentin ”CreateGrid”-metodia. Tämä kyseinen metodi ottaa parametrikseen juuri lasketut tiedot ja rakentaa niistä osan visuaalista graafia.

Jokaisen graafin osan luonnin lopuksi alustetaan kaikki tarvittavat taulukot ja muuttujat nollaan, jotta uuden graafin luontia voidaan jatkaa puhtaalta pöydältä. Tätä toistetaan kunnes jokainen maaston kohta on käyty läpi ja rakennettu graafiin.

Kuviossa 17 nähdään tyhjä maasto, jonka pintaa on muokattu kumparemaiseksi. Kun projektiin lisätty graafikoodi ajetaan, tuloksena on kuvion 18 näköinen graafi, joka kulkee maaston pinnan mukaan. Tämä helpottaa huomattavasti selventämään tekoälyn liikkumisen vaiheita sekä mahdollisten esteiden esiintymistä graafissa.



Kuvio 17. Maasto ennen laskutoimitusta.



Kuvio 18. Laskettu graafi.

4.2.4 Datagraafin luonti

Toisen graafin tehtävänä on tallentaa kaikki tarvittava data muistiin, jota A*-reitinhaku algoritmi lukee. Toisin kuin visuaalinen graafi, tätä graafia ei voi jättää laskematta, mutta testaamista varten voidaan lisätä muutamia visuaalisia ominaisuuksia, jotka voi halutessaan jättää pois päältä varsinaisesta pelistä.

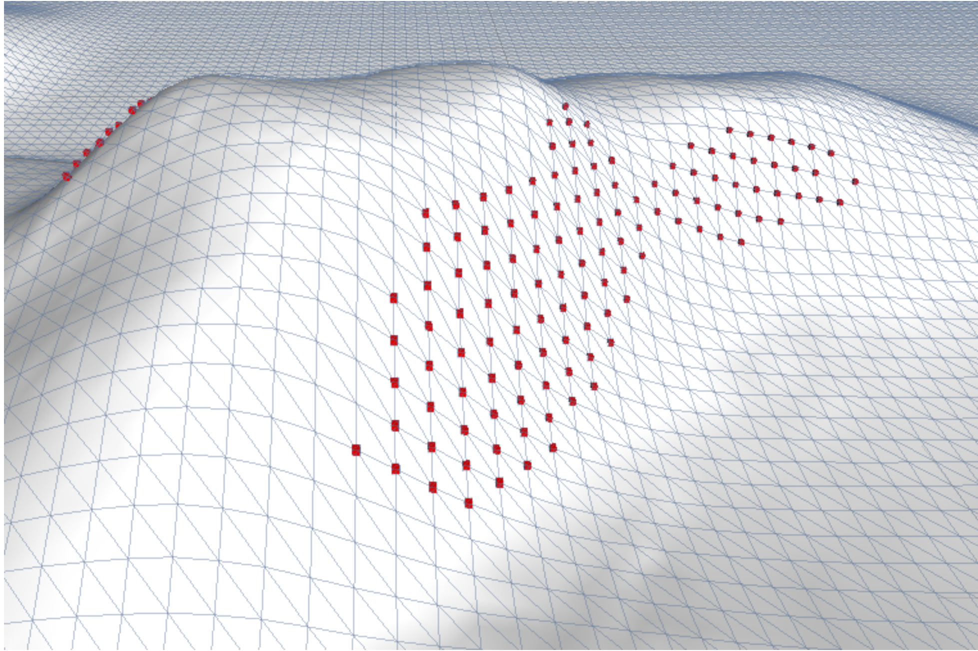
Koodillisesti datagraafi muistuttaa visuaalisen graafin pohjaa, mutta jos liitteen 2 koodia tarkastellaan syvemmin, voidaan huomata eroavaisuuksia. Yksi tärkeimmistä asioista on huomata, että visuaalisessa graafissa käsitellään verteksejä ja datagraafissa puhutaan solmuista ”Nodes”.

Toinen ero on se, että visuaalisessa graafissa tarvittiin vain laskea joka toinen verteksi for-silmukassa, kun taas datagraafissa tulee käydä jokainen solmu tarkasti läpi. Tämä on erittäin tärkeää, koska yksikin puutteellinen solmu graafissa voi aiheuttaa kriittisiä ongelmia itse pelissä.

Liitteen 2 alussa on erillinen luokka ”Node”. Tämä kyseinen luokka sisältää yhden solmun kaikki tarvittavat muuttujat. Näistä A*-reitinhauku algoritmiin tärkeimpiä muuttujia ovat ”distFromStart”, joka on lyhin mahdollinen matka alkupisteestä kyseiseen solmuun, sekä ”distToGoal”, eli suoraan laskettu lyhin matka päätössolmuun.

Koodin tärkein osuus on itse for-silmukassa, joka koostuu kahdesta osiosta. Ensimmäinen on korkeuden tarkastelu, jossa käsiteltävä solmu tarkistaa korkeuseron kaikkiin naapurisolmuihinsa nähden. Jos eroavaisuutta on ylitse maksimijyrkkyyden ”steepness”, merkataan kyseinen solmu suljetuksi. Tämän laskettuaan silmukka tarkistaa solmun ympäryksen mahdollisten esteiden varalta ja merkkaa sen suljetuksi tarvittaessa. On myös madollista, että solmu osuu dynaamiseen objektiin, joka merkataan myös solmuun ylös tarvittavien jatkotoiminpiteiden varalta.

Jos maastoa tarkastellaan hiukan lähempää, voidaan nähdä miten jyrkkiin kohtiin on muodostunut alueita, joiden solmut ovat punaisia (Kuvio 19). Tämä helpottaa havainnollistamaan kuinka tekoäly väistelee esteitä ja rakentaa itselleen reitin joka väistelee punaisia alueita.

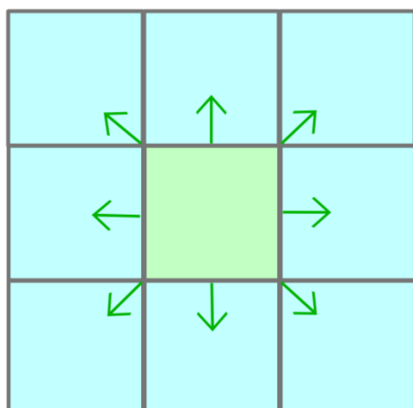


Kuvio 19. Laskettu jyrkänne.

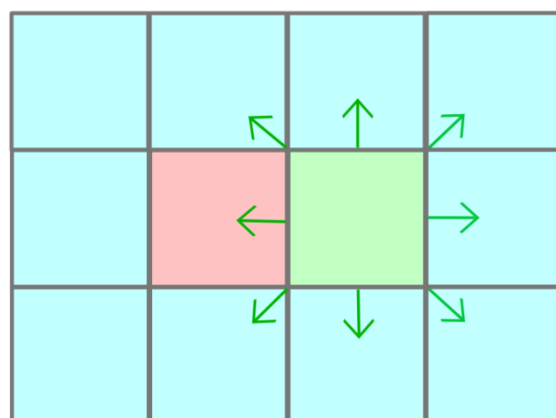
4.2.5 A*-algoritmin luonti

Ennen kuin siirrytään liitteen 3 koodiin, on hyvä tietää kuinka A*-reitinhaku toimii teoriassa (liite 4). Itse laskeminen aloitetaan lähtöpisteen solmusta, josta käydään tähtimuodostelman muotoisesti kahdeksan lähintä solmua läpi (Kuvio 20). Nämä kaikki solmut siirretään aukinaisten solmujen listaan ja niistä lähdetään tarkastamaan paras ensin -taktiikalla optimaalisimman solmun. Kun siirrytään käsiteltävään solmuun, se itsessään siirretään heti suljettujen solmujen listaan ja sen vanhemmaksi määritetään edeltävä solmu (Kuvio 21). Tämä on erittäin tärkeä vaihe, koska lasketun reitin muodostamisessa täytyy tietää, jokaisen solmun vanhempi.

Kuvioiden vaaleansiniset laatikot ovat aukinaisia solmuja, vaaleanpunainen suljettu ja vaaleanvihreä viittaa käsiteltävään solmuun, joka on myös suljettu.



Kuvio 20. Lähtöpiste.



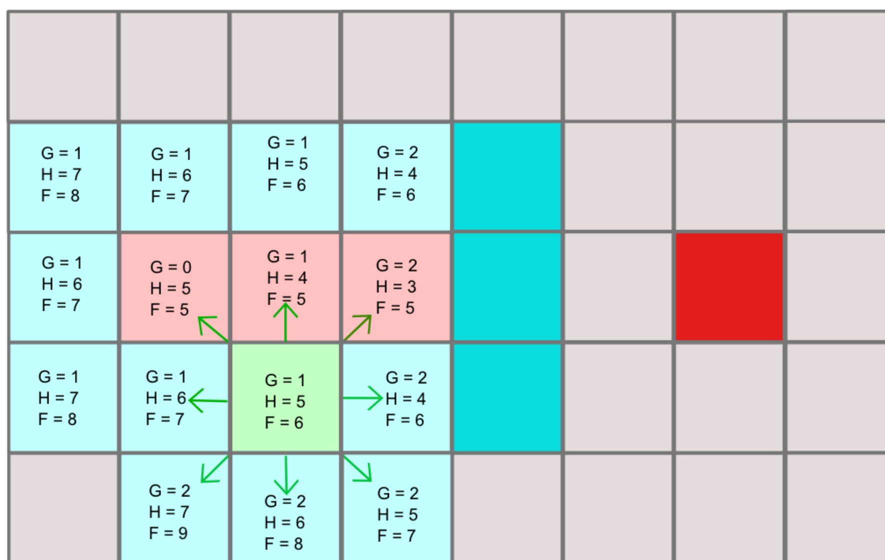
Kuvio 21. Ensimmäinen askel.

Kuten jo datagraafista kertovassa luvussa mainittiin, A*-reitinhauelle tärkeät muuttujat ovat käsiteltävien solmujen välimatkat sekä maaliin että aloituspisteeseen. Käytetään siis A*-algoritmin yleistä kaavaa $f(n) = g(n) + h(n)$, missä f on välimatkojen summa, g solmuun liikuttava matka aloituspisteestä ja h maaliin saapumiseen tarvittava matka. Kuvio 22 havainnollistaa kyseistä prosessia. Vaikkakin A* itsessään voi liikkua myös nurkissa oleviin solmuihin, laskiessa välimatkaa joko alkuun tai maaliin, joudutaan laskemaan solmut sekä pysty että vaakasuunnassa. Teoriassa nurkissa oleviin solmuihin matkaa on kaksi yksikköä ja muihin neljään vain yksi. Kuviosta 22 nähdään, että punainen solmu on maali, johon pyritään pääsemään.

Kuvio 22. Solmujen arvot.

Kuviossa 22 nähdään, että lähtösolmun G-arvo on nolla, koska sen matka itseensä on nolla. Taas jos tarkastellaan sen kahdeksaa naapurisolmua, huomataan niiden G-arvojen olevan yksi. Tämä johtuu siitä, että ne sijaitsevat yhden liikkumisyksikön verran aloitussolmusta. Kun kaikkien solmujen F-arvoja verrataan, nähdään että solmut jotka ovat lähempänä maalisolmua, omistivat pienemmän F-arvon. Tällöin on päästy tulokseen, jossa A* on liikkunut yhden askeleen parhaimpaan mahdolliseen solmuun.

Kun käsiteltävästä solmusta ei avaudu uusia solmuja tai niiden F-arvo on suurempi kuin pienimmän F-arvon omaavan solmun avoimessa listassa, on osuttu esteeseen. Kuviossa 23 A* on siirtynyt avoimen listan ensimmäiseen pienimpään solmuun, koska kuvan siniset solmut ovat suljettuja. A* tutkii jokaisen pienimmän F-arvon alkaen pienimmästä G-arvosta, kunnes jostakin solmusta aukeaa sitäkin pienempi F-arvo.



Kuvio 23. Askel 2.

Kyseistä algoritmia toistetaan kunnes käsiteltävästä solmusta maalisolmuun on matkaa yhden liikkumisyksikön verran. Kuviosta 24 nähdään kuinka A* on laskenut jokaisen tarvittavan solmun löytääkseen tiensä maalisolmulle asti. Tämä ei kuitenkaan vielä riitä tekoälylle, jotta se voisi liikkua alkupisteestä maaliin. A*:n tulee vielä rakentaa lyhin mahdollinen reitti käyttäen suljetun listan solmuja jota pitkin tekoäly tulee kulkemaan.

Reitin laskeminen toteutetaan päinvastaisessa järjestyksessä verrattuna edeltävään toimenpiteeseen, mutta muuten se toimii melko samalla periaatteella. Suljetusta listasta valitaan solmu joka päätyi maaliin, jolloin reittiä lähdetään muodostamaan lisäämällä tämän solmun vanhempi reittitaulukkoon. Kyseinen vanhempi on paras mahdollinen liikkumiskohde, koska sen kautta A* on löytänyt reitin maaliin ja siksi sitä ei tarvitse laskea enää uudelleen. Sitten kyseistä toimenpidettä toistetaan kunnes saavutaan takaisin lähtösolmuun. Oheisessa kuviossa keltaisella reunuksilla olevat suljetut solmut ovat reitti, jonka A* on laskenut.

	G = 2 H = 7 F = 9	G = 2 H = 6 F = 8	G = 2 H = 5 F = 7	G = 3 H = 4 F = 7			
G = 1 H = 7 F = 8	G = 1 H = 6 F = 7	G = 1 H = 5 F = 6	G = 2 H = 4 F = 6				
G = 1 H = 6 F = 7	G = 0 H = 5 F = 5	G = 1 H = 4 F = 5	G = 2 H = 3 F = 5		G = 5 H = 1 F = 6		
G = 1 H = 7 F = 8	G = 1 H = 6 F = 7	G = 1 H = 5 F = 6	G = 2 H = 4 F = 6		G = 4 H = 2 F = 6	G = 5 H = 1 F = 6	
	G = 2 H = 7 F = 9	G = 2 H = 6 F = 8	G = 2 H = 5 F = 7	G = 3 H = 4 F = 7	G = 4 H = 3 F = 7	G = 5 H = 2 F = 7	

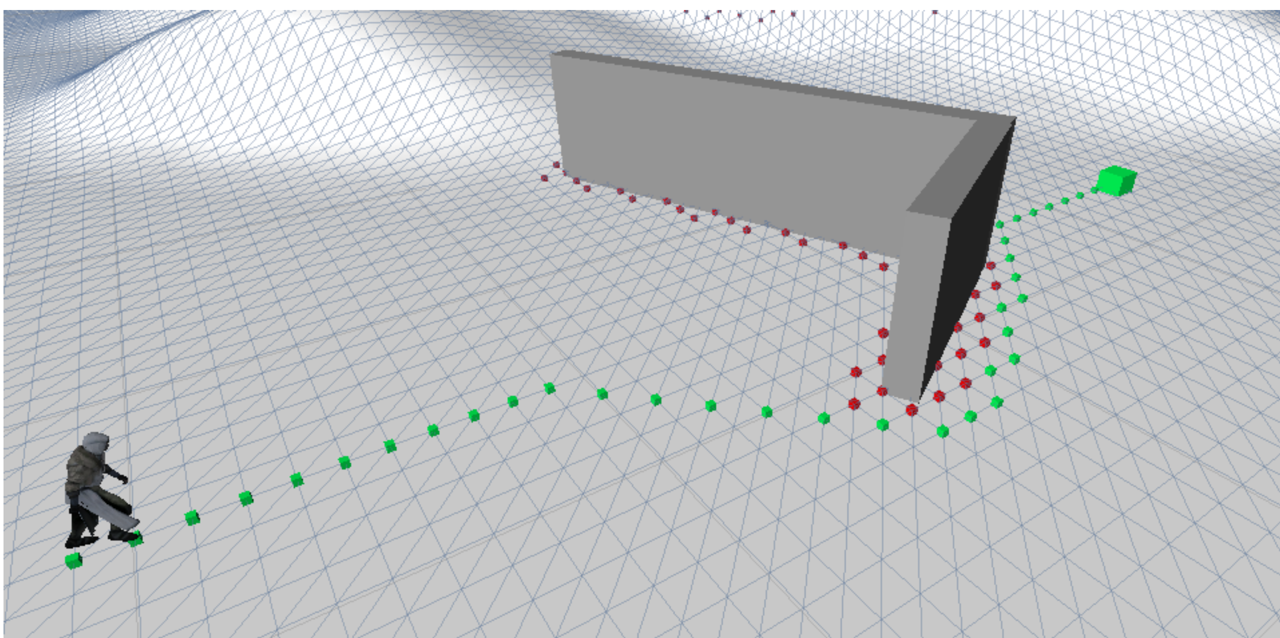
Kuvio 24. Laskettu reitti.

Liitteessä 3 nähdään koodi A*-algoritmista. Samoin kuin datagraafissa, A* tarvitsee erillisen solmuluokan josta se voi luoda itselleen listat avoimille ja suljetuille solmuille. Jos koodia verrataan ylhäällä oleviin kuviin, voidaan huomata kuinka yksinkertaisesti ja helposti A* luodaan käyttäen Unity-pelimoottoria ja C#-kieltä.

Itse A*-algoritmi on vain hyvä pohja lähteä soveltamaan omaa reitinhakumekaniikkaa, koska siitä saa jo pienillä lisäyksillä tehtyä erittäin realistisia reittejä. Esimerkiksi soveltaen visuaalista graafia navigaatioverkoksi olisi mahdollista saada huomattavasti tehokkaampi reitinhakumenetelmä. Tämä kuitenkin vaatisi suuria muutoksia sekä datagraafissa, sekä A*:n toimintalogiikassa.

Toisin kuin A*-algoritmissa käytettävä graafi, navigaatioverkko rakennetaan jo pelintekovaiheessa, koska sen generoimisessa voi kestää useita minuutteja sen moniosaisen luomisprosessin vuoksi. Useimmiten puhutaan ”agenteista”, kun viitataan NPC-hahmoihin jotka liikkuvat käyttäen navigaatioverkkoa.

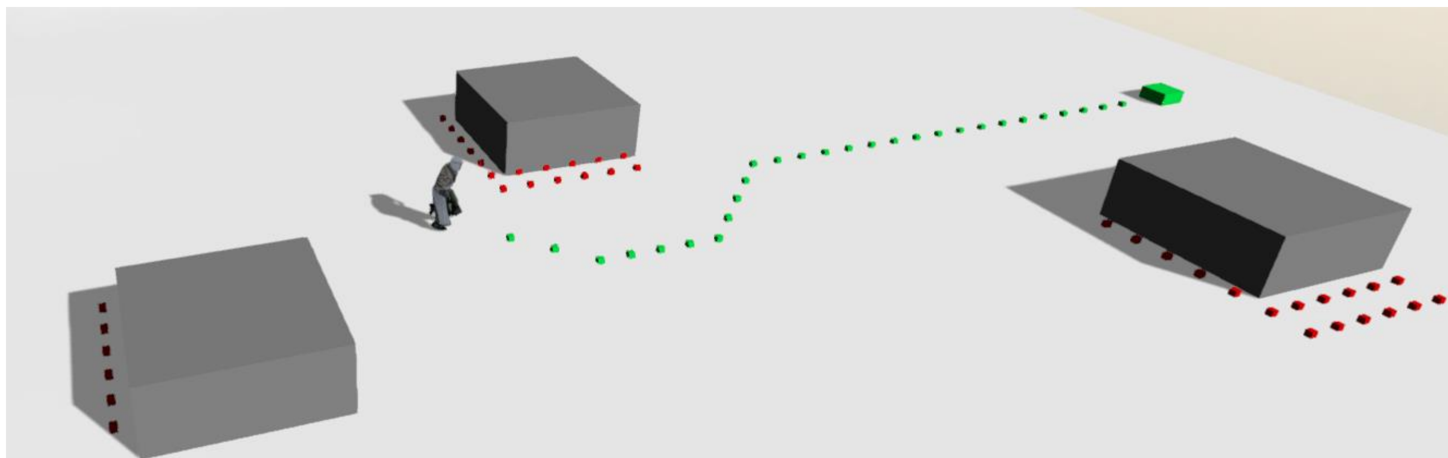
Jätin koodistani liikkumiseen liittyvät metodit pois, koska ne eivät olleet olennaisia aiheessamme ja ne eivät kuulu A*-algoritmin ytimeen. Reitin luonninkin voisi tehdä vaihtoehtoisilla tavoilla käyttäen Unityn omia ominaisuuksia, mutta se olisi rikkonut jälleen A*:n logiikkaa. Kuviossa 25 on simulaatio, jossa Unity-pelimoottorilla on toteutettu kaikki kolme koodia, ja lopputuloksena on lähes samanlainen reitti kuin kuvassa 16.



Kuvio 25. Laskettu reitti Unity-pelimoottorissa.

Dynaamisia objekteja on helppo lisätä valmiiseen graafiin, koska niiden tarvitsee vain muuttaa peittämiensä solmujen arvoja, jotta reitinhaku osaisi väistää niitä. Tämä on vain yksi tapa hoitaa dynaamisten objektien käyttäytymistä ja se ei välttämättä sovellu jokaiseen tilanteeseen. Esimerkiksi objektit, joiden etäisyys peittämistään solmuista olisi toiselle pelaajalle liian matala, mutta toiselle pelaajalle se olisi juuri sopiva. Tallaisesta tilanteesta syntyisi ristiriita ja dynaamisen objektin toiminnalle tulisi tehdä erilainen logiikka.

Kuviossa 26 harmaat laatikot liikkuvat ristiin ja NPC-hahmon tehtävänä on pujotella reitti kohti vihreää laatikkoa. Kuvion keskimmäinen laatikko on juuri estänyt NPC-hahmon reittiä ja A*-algoritmi on laskenut uudeksi reitiksi kuviossa esiintyvän mutkan. Muutaman sekunnin kuluttua kuvion viimeinen laatikko osuu NPC-hahmon eteen ja A*-algoritmi joutuu päivittämään reittiä. Harmaat laatikot ovat dynaamisia objekteja, joten ne sulkevat peittämänsä solmut. Suljetut solmut näkyvät kuviossa punaisilla laatikoilla.



Kuvio 26. Dynaamiset objektit

5 Yhteenveto

Opinnäytetyöni tavoitteena oli tutkia NPC-hahmojen toimintaa monesta eri näkökulmasta ja samalla oppia uusia menetelmiä ja käsitteitä. Toiminnollisen osuuden tarkoituksena oli oppia yksi NPC-hahmojen tärkeimmistä ominaisuuksista kooditasolla ja pyrkiä soveltamaan valmiista ideasta oma toimiva versio. Opinnäytetyön tarkoituksena oli myös parantaa töissä käyttämiäni menetelmiä tekoälyn ohjelmoinnissa. Aikaa opinnäytetyön tekoon meni noin yhdeksän kuukautta, josta kaksi kuukautta kului ohjelmointiin, toiset kaksi kuukautta harjoitusaineeseen ja loput viisi kuukautta raportin kirjoittamiseen.

Tärkein tutkimuskohde opinnäytetyössä oli NPC-hahmojen tilanteenluku. Tilanteenluvulle on todella monta eri toteutustapaa, mistä tavoitteeseen suuntaava tehtävän suunnittelija toimisi parhaiten omiin tekoälyprojekteihini. Tulevaisuudessa aion tutkia kyseistä menetelmää tarkemmin ja soveltaa sitä töissä tekemäämme uuteen peliprojektiin.

Dynaamisesta reitinhausta Unity-pelimoottorilla ei ole ollut minulle mitään hyötyä vielä toistaiseksi, mutta toivottavasti löydän sille jotakin käyttöä tulevaisuudessa. Voin lähteä myös kehittämään dynaamista reitinhakua navigaatioverkoksi, koska rakentamastani pohjasta on helppo lähteä jatkokehittämään uusia sovellutuksia. Reitinhaku on NPC-hahmoilla käytössä lähes jokaisessa kolmiulotteisessa pelissä, joten reitinhaun ymmärtäminen tulee varmasti auttamaan minua uusissa peliprojekteissa. Aion ottaa itselleni tavoitteeksi oppia jokaisen NPC-toimintaan liittyvän ominaisuuden huolella ja kehittää itseäni tekoälyohjelmoijana.

Lähteet

Beal, V. 2014. AI - artificial intelligence. http://www.webopedia.com/TERM/A/artificial_intelligence.html. 15.11.2014.

Blizzard. 2014. Warcraft III. http://eu.blizzard.com/en-gb/games/war3/. 11.11.2014.

Brownlee, J. 2002. Finite State Machines (FSM). http://ai-depot.com/FiniteStateMachines/FSM-Background.html. 3.11.2014.

Dijkstra, E. 1959. A Note on Two Problems in Connexion with Graphs. Numerische Mathematik 1, 269–271. http://www-m3.ma.tum.de/foswiki/pub/MN0506/WebHome/dijkstra.pdf. 3.11.2014.

Dreyfus, H. 1972. What Computers Can't Do. 106 s. ISBN 0-06-011082-1.

Dvorsky, G. 2013. How Much Longer Before Our First AI Catastrophe? http://io9.com/how-much-longer-before-our-first-ai-catastrophe-464043243. 1.5.2014.

Edelman, M. & Izhikevich, M. 2007. http://www.izhikevich.org/publications/large-scale_model_of_human_brain.pdf. 1.5.2014.

Epic Games. 2014a. Epic Games. http://epicgames.com/. 11.11.2014.

Epic Games. 2014b. Navigation Mesh Reference. http://udn.epicgames.com/Three/NavigationMeshReference.html. 1.5.2014.

Ertürk, U, R. 2009. Short Term Decision Making with Fuzzy Logic And Long Term Decision Making with Neural Networks In Real-Time Strategy Games. http://www.hevi.info/tag/artificial-intelligence-in-real-time-strategy-games/. 4.11.2014.

Indika. 2011. Difference Between Strong AI and Weak AI. http://www.difference-between.com/difference-between-strong-ai-and-vs-weak-ai/. 3.11.2014.

McCulloch, W. & Walter, P. 1943. A Logical Calculus of Ideas Immanent in Nervous Activity. Bulletin of Mathematical Biophysics 5, 115–133.

Mooney, J. 2006. Philosophical Arguments Against "Strong" AI. http://www.cs.utexas.edu/~mooney/cs343/slide-handouts/philosophy.4.pdf. 3.11.2014.

Owens, B. 2014. Goal Oriented Action Planning for a Smarter AI. http://gamedevelopment.tutsplus.com/tutorials/goal-oriented-action-planning-for-a-smarter-ai--cms-20793. 10.11.2014.

Rouse M. 2011. Natural language processing (NLP) http://searchcontentmanagement.techtarget.com/definition/natural-language-processing-NLP. 1.5.2014.

Schreiner, T. 2007. Artificial Intelligence in Game Design. http://ai-depot.com/GameAI/Design.html. 11.11.2014.

Simpson, C. 2014. Behavior trees for AI: How they work. http://www.gamasutra.com/blogs/ChrisSimpson/20140717/221339/Behavior_trees_for_AI_How_they_work.php. 3.11.2014.

Stanford. 2014. NPC Converstaion. http://www-cs-students.stanford.edu/~amitp/Articles/NPC-Conversation.html. 26.10.2014.

TheHelper. 2014. World Editor Tutorials. http://world-editor-tutorials.thehelper.net/. 11.11.2014.

Unity. 2014. Unity 3D. http://unity3d.com/. 15.11.2014.

Webopedia. 2014a. NPC. http://www.webopedia.com/TERM/N/NPC.html. 15.11.2014.
Webopedia. 2014b. RTS. http://www.webopedia.com/TERM/R/RTS.html. 15.11.2014.
Webopedia. 2014c. XML. http://www.webopedia.com/TERM/X/XML.html. 15.11.2014.
Wikipedia. 2014a. A* search algorithm. http://en.wikipedia.org/wiki/A*_search_algorithm. 1.5.2014.
Wikipedia. 2014b. Dijkstra’s algorithm. http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm. 1.5.2014.
Zwass, W. 2014. Expert Systems. http://global.britannica.com/EBchecked/topic/198506/expert-system. 1.5.2014.

```
public Vector3[] newVertices; //graafin verteksit
public Vector3[] newNormal; //graafin normaalit
public Vector2[] newUV; //graafin uv
public int[] newTriangles; //pisteet joista luodaan kolmioita
Terrain terrain; //maasto
int W; //maaston leveys
int H; //maaston korkeus (z koordinaatistossa)
int vCount = 0; //vektori määrä
int tCount = 0; //kolmiopisteiden määrä
int currentChild = 0; //tämänhetkinen graafin lapsiobjecti
float gcc; //graafin osien määrä
float vX; //käseiteltävän verteksin x koordinaatti
float vY; //käseiteltävän verteksin y koordinaatti
float vZ; //käseiteltävän verteksin z koordinaatti
Vector3 tempCoord; //käsiteltävä koordinaatti

void Start ()
{
        terrain = Terrain.activeTerrain; //haetaan aktiivinen maasto
        transform.position = terrain.gameObject.transform.position; //asetetaan graafin aloitus kohdaksi maaston koordinaatit
        tempCoord = terrain.gameObject.transform.position; //tedään sama käsiteltäville koordinaateille
        W = (int)terrain.terrainData.size.x * 2; //laketaan maaston leveys
        H = (int)terrain.terrainData.size.z * 2; //lasketaan maaston korkeus
        int c = (int)(((H - 1) * (W - 1))); //lasketaan graafin verteksien määrä
        gcc = c / 64992;
        if(c % 64992 != 0)
        {
                gcc = (int)gcc++; //määritetään graafin osien määrä
        }
        newVertices = new Vector3[64992];  //Graafi voi sisältää vain 64992 verteksiä, normaalia ja uv:ta per osa
        newNormal = new Vector3[64992];
        newUV = new Vector2[64992];
        newTriangles = new int[97488]; //Graafi voi sisältää vain 97488 kolmion pistettä per osa
        GameObject grid = Resources.Load("Grid") as GameObject; //ladataan tyhjä graafi
        for(int i = 0; i <= gcc; i++) //luodaa tarvittava määrä tyhjiä graafeja
        {
                GameObject gridClone = Instantiate(grid, transform.position, Quaternion.identity) as GameObject;
                gridClone.transform.parent = transform;
                gridClone.name = "Grid " + i;
        }
        for (int y = 0; y < H - 1; y = y + 2) //lasketaan graafin joka toinen verteksi
        {
            for (int x = 0; x < W - 1; x = x + 2)
            {
                if(x == 0) //ensimmäisessä pisteessä x koordinaatti ei liiku
                {
                        tempCoord = new Vector3(0.0f, tempCoord.y, tempCoord.z);
                }
                else //muulloin liikutaan yhdellä yksiköllä eteenpäin
                {
                        tempCoord = new Vector3(tempCoord.x + 1.0f, tempCoord.y, tempCoord.z);
                }
                //määritetään neliön 2 komiota jotka koostuvat neljästä verteksistä, kuudesta komion pisteestä,
```

```
            //nejästä uv:sta ja neljästä normaalista
            vX = tempCoord.x;
            vY = terrain.SampleHeight(tempCoord);
            vZ = tempCoord.z;
            newVertices[vCount] = new Vector3(vX, vY + 0.1f, vZ); //jotta graafi erottuisi helpommin maastosta,
            //luodaan se hiukan maaston yläpuolelle
            vX = (tempCoord.x + 1.0f);
            vY = terrain.SampleHeight(new Vector3(vX, vY, vZ));
            newVertices[vCount + 1] = new Vector3(vX, vY + 0.1f, vZ);
            vX = tempCoord.x;
            vZ = (tempCoord.z + 1.0f);
            vY = terrain.SampleHeight(new Vector3(vX, vY, vZ));
            newVertices[vCount + 2] = new Vector3(vX, vY + 0.1f, vZ);
            vX = (tempCoord.x + 1.0f);
            vY = terrain.SampleHeight(new Vector3(vX, vY, vZ));
            newVertices[vCount + 3] = new Vector3(vX, vY + 0.1f, vZ);
            newUV[vCount] = new Vector2(0, 0);
            newUV[vCount + 1] = new Vector2(1, 0);
            newUV[vCount + 2] = new Vector2(0, 1);
            newUV[vCount + 3] = new Vector2(1, 1);
            newNormal[vCount] = -Vector3.forward;
            newNormal[vCount + 1] = -Vector3.forward;
            newNormal[vCount + 2] = -Vector3.forward;
            newNormal[vCount + 3] = -Vector3.forward;
            newTriangles[tCount] = vCount; //alku piste
            newTriangles[tCount + 1] = vCount + 2; //alkupisteestä yksi ylös
            newTriangles[tCount + 2] = vCount + 1; //alkupisteestä yksi oikealle
            newTriangles[tCount + 3] = vCount + 2;
            newTriangles[tCount + 4] = vCount + 3; //alkupisteestä yksi ylös ja oikealle
            newTriangles[tCount + 5] = vCount + 1;

            vCount = vCount + 4;
            tCount = tCount + 6;
            if(vCount == 64992) //jos graafin osa täyttyy, se luodaa valmiiksi kutsumalla GreateGrid metodia
            {
                transform.GetChild(currentChild).GetComponent<CreateMesh>().CreateGrid(newVertices,
                newUV, newTriangles, newNormal);
                //alustetaan uusi graafin osa
                vCount = 0;
                tCount = 0;
                newVertices = new Vector3[64992];
                newNormal = new Vector3[64992];
                newUV = new Vector2[64992];
                newTriangles = new int[97488];
                currentChild++;
            }
        }
            //liikutaan z koordinaatistossa yks yksikkö eteenpäin
            tempCoord = new Vector3(tempCoord.x, tempCoord.y, tempCoord.z + 1.0f);
    }
    //kun kaikki on laskettu, luodaan tämänhetkisen osan graafi
    transform.GetChild(currentChild).GetComponent<CreateMesh>().CreateGrid(newVertices, newUV, newTriangles, newNormal);
}
```

```
public class Node
{
        //Yksittäisen solmun tiedot
        public bool dynamicCol; //onko solmu dynaamisessa törmäyksessä
        public bool closed; //onko solmu suljettu
        public int distFromStart; //matka alkupisteestä
        public int distToGoal; //matka päätepisteeseen
        public Vector2 pos; //sijainti graafissa
        public Vector3 realPos; //sijainti maailman koordinaatistossa
        public GameObject dynObj; //dynaaminen objekti joka aiheuttaa dynaamisen törmäyksen
}
[SerializeField]
bool TestMode; //tällä saamme punaiset laatikot näkymään
[SerializeField]
float steepness; //maksimi jyrkkyys
public Node[,] nodes; //solmu taulukko
GameObject redBox; //punainen laatikko
Terrain terrain; //maasto
int W; //maaston leveys
int H; //maaston korkeus
float vX; //käsiteltävän solmun X koordinaatti
float vY; //käsiteltävän solmun Y koordinaatti
float vZ; //käsiteltävän solmun Z koordinaatti
Vector3 tempCoord; //käsiteltävä koordinaatti
void Start ()
{
        redBox = Resources.Load("redBox") as GameObject; //ladataan punainen laatikko
        terrain = Terrain.activeTerrain; //etsitään aktiivinen maasto
        transform.position = terrain.gameObject.transform.position; //asetetaan graafin sijainti
        tempCoord = terrain.gameObject.transform.position;
        W = (int)terrain.terrainData.size.x; //määritetään leveys
        H = (int)terrain.terrainData.size.z; //määritetään korkeus
        nodes = new Node[W, H]; //luodaan solmu taulukko
        for (int y = 0; y < H; y++)
        {
            for (int x = 0; x < W; x++)
            {
                if(x == 1)
                {
                        tempCoord = new Vector3(0.0f, tempCoord.y, tempCoord.z);
                }
                else
                {
                        tempCoord = new Vector3(tempCoord.x + 1.0f, tempCoord.y, tempCoord.z);
                }
                vX = tempCoord.x;
                vY = terrain.SampleHeight(tempCoord) + 0.1f;
                vZ = tempCoord.z;
                Node node = new Node();
                nodes[x, y] = node; //määritetään uusi solmu
                nodes[x, y].pos = new Vector2(x, y); //annetaan sille graafin koordinaatti
                nodes[x, y].realPos = new Vector3(vX, vY, vZ); //sekä maailman koordinaatti
```

```
            if(x > 0 && y > 0) //jos naapuri solmu on jyrkkyyden verran liian korkealla tai alhallaa, solmu on punainen
            {
                    if(nodes[x - 1, y].realPos.y + steepness < nodes[x, y].realPos.y || nodes[x - 1, y].realPos.y - steepness > nodes[x, y].realPos.y
                    || nodes[x, y - 1].realPos.y + steepness < nodes[x, y].realPos.y || nodes[x, y - 1].realPos.y - steepness > nodes[x, y].realPos.y)
                    {
                            nodes[x, y].closed = true;
                            if(TestMode) //luodaan laatikko, jos testaus on totta
                            {
                                    GameObject redClone = Instantiate(redBox, new Vector3(vX, vY, vZ), Quaternion.identity) as GameObject;
                                    redClone.name = "redBox" + nodes[x, y].pos;
                                    redClone.transform.parent = transform;
                            }
                    }
            }//jos solmu osuu objectiin, se on punainen
            Collider[] hitColliders = Physics.OverlapSphere(new Vector3(vX, vY, vZ), 1.0f, (1 << 9));
            if(hitColliders.Length > 0)
            {
                    int i = 0;
                    bool boxDone = false;
                    while (i < hitColliders.Length)
                    {
                            if(hitColliders[i].tag != "terrain" && !boxDone)
                            {
                                boxDone = true;
                                nodes[x, y].closed = true;
                                if(TestMode)
                                {
                                    GameObject redClone = Instantiate(redBox, new Vector3(vX, vY, vZ), Quaternion.identity) as GameObject;
                                    redClone.name = "redBox" + nodes[x, y].pos;
                                    redClone.transform.parent = transform;
                                }
                            }

                            if(!hitColliders[i].gameObject.isStatic) //jos objecti ei ole staattinen
                            {
                                    nodes[x,y].dynamicCol = true;
                                    nodes[x,y].dynObj = hitColliders[i].gameObject;
                                    i = hitColliders.Length;
                            }
                            i++;
                    }
            }
        }
        tempCoord = new Vector3(tempCoord.x, tempCoord.y, tempCoord.z + 1.0f);
    }
}
```

```
public class Nodes
{
        //Yksittäisen solmun tiedot
        public int f; //g + h
        public int g; //matka lähtöpisteestä solmuun
        public int h; //matka solmusta maaliin
        public Vector2 pos; //sijainti graafissa
        public Vector2 parentPos; //vanhemman sijainti graafissa
        public Vector3 realPos; //sijainti maailmassa

}
List<Nodes> openSet = new List<Nodes>(); //avoin lista
List<Nodes> closedSet = new List<Nodes>(); //suljettu lista
List<Vector3> path = new List<Vector3>(); //reitti
GridRaw grid; //data graafi
Terrain terrain; //maasto
float W; //maaston leveys
float H; //maaston korkeus
public int tX; //objectin X sijainti graafissa
public int tY; //objectin Y sijainti graafissa
int taX; //kohteen X sijainti graafissa
int taY; //kohteen Y sijainti graafissa
int curPathId = 0; //solmu jota kohti liikutaan
public int bestf; //paras F arvo
public int currentG = 0; //käsiteltävän solmun G arvo
float timer = 0.0f; //laskin
float maxTime = 0.01f; //maksimiaika while-silmukalle
bool pathFound = false; //reitti on löydetty
bool pathCalced = false; //reitti on laskettu
bool calcing = false; //ovatko laskutoimitukset kesen
public GameObject targetObj; //kohde
Vector3 target; //kohteen koordinaatti
Vector3 terPos; //graafin koordinatti
Vector2 getPos; //käsiteltävän solmun koordinaatti

void Start ()
{
        grid = GameObject.Find("Grid").GetComponent<GridRaw>(); //haetaan data graafi
        terrain = Terrain.activeTerrain; //haetaan maasto
        W = terrain.terrainData.size.x; //määritetään maaston leveys
        H = terrain.terrainData.size.z; //määritetään maaston syvyys
        target = new Vector3((((targetObj.transform.position.x / terrain.terrainData.size.x) * W) + 1), 0,
         (((targetObj.transform.position.z / terrain.terrainData.size.z) * H)));
        taX = Mathf.RoundToInt(target.x); //lasketaan kohteen ja objektin sijainnit graafissa
        taY = Mathf.RoundToInt(target.z);
        terPos = new Vector3((((transform.position.x / terrain.terrainData.size.x) * W) + 1), 0,
         (((transform.position.z / terrain.terrainData.size.z) * H)));
        tX = Mathf.RoundToInt(terPos.x);
        tY = Mathf.RoundToInt(terPos.z);
        bestf = 1000000;
}
```

```
void FixedUpdate()
{
        if(!pathFound && !calcing && running)
        {
                StartCoroutine(Astar()); //aloitetaan A*
        }
        if(pathFound && !pathCalced)
        {
                SeekPath(); //luodaan reitti
        }
}
IEnumerator Astar()
{
        if(!pathFound)
        {
                calcing = true;
                timer = 0.0f;
                while(!pathFound || timer < maxTime)
                {
                        timer += Time.deltaTime;
                        //lisätään käsiteltävä solmu suljettuun listaan ja poistetaan se aukinaisten solmujen listasta
                        if(openSet.Exists(a => a.pos == grid.nodes[tX, tY].pos))
                        {
                                int dist = (int)(Mathf.Abs(tX - taX) + Mathf.Abs(tY - taY));
                                currentG = openSet.Find(a => a.pos == grid.nodes[tX, tY].pos).g;
                                closedSet.Add(new Nodes(){f = dist + currentG, h = dist, g = currentG, pos = grid.nodes[tX, tY].pos,
                                 realPos = grid.nodes[tX, tY].realPos, parentPos = openSet.Find(a => a.pos == grid.nodes[tX, tY].pos).parentPos});
                                openSet.Remove(openSet.Find(a => a.pos == grid.nodes[tX, tY].pos));
                                currentG += 1;
                                CheckAOE(tX, tY);
                        }
                        else
                        {
                                int dist = (int)(Mathf.Abs(tX - taX) + Mathf.Abs(tY - taY));
                                closedSet.Add(new Nodes(){f = dist, h = dist, g = 0, pos = grid.nodes[tX, tY].pos, realPos = grid.nodes[tX, tY].realPos,
                                 parentPos = Vector2.zero});
                                currentG = 10;
                                CheckAOE(tX, tY);
                        }
                }
                if(pathFound)
                {
                        calcing = false;
                }
        }
        yield return new WaitForSeconds(0.01f);
}
```

```
void CheckAOE(int x, int y)
{
        //tarkastetaan 8 naapurisolmua
        for(int i = -1; i <= 1; i++)
        {
                for(int j = -1; j <= 1; j++)
                {
                        if((!grid.nodes[x + i, y + j].closed || grid.nodes[x + i, y + j].dynObj == gameObject))
                        {
                                newMoveFound = true;
                                int dist = (int)(Mathf.Abs((x + i) - taX) + Mathf.Abs((y + j) - taY));
                                if(!openSet.Exists(a => a.pos == grid.nodes[x + i, y + j].pos) &&
                                         !closedSet.Exists(a => a.pos == grid.nodes[x + i, y + j].pos))
                                {
                                        openSet.Add(new Nodes(){f = dist + currentG, h = dist, g = currentG,
                                         pos = grid.nodes[x + i, y + j].pos,
                                        parentPos = new Vector2(x, y)});
                                }
                        }
                }
        }
        if(openSet.Count > 0) //jos avoin lista ei ole tyhjä
        {
                bestf = openSet.Min(a => a.f);
                getPos = openSet.Find(a => a.f == bestf).pos;
                if(openSet.Find(a => a.f == bestf).h <= 0)
                {
                        pathFound = true;
                }
                else
                {
                        tX = (int)getPos.x;
                        tY = (int)getPos.y;
                }
        }
        else //muulloin olemme jumissa
        {
                Debug.Log("stuck");
                StopCoroutine("Astar");
        }
}
void SeekPath() //luodaan reitti solmujen vanhemmista
{
        Vector2 pPos;
        float timer1 = 0.0f;
        float cd = 0.1f;
        pathCalced = false;
        while(!pathCalced || timer < cd)
        {
                timer += Time.deltaTime;

                if(closedSet.Find(a => a.pos == grid.nodes[tX, tY].pos).g == 0)
                {
                        curPathId = path.Count - 1;
                        target = path[curPathId];
                        pathCalced = true;
                }
                else
                {
                        path.Add(closedSet.Find(a => a.pos == grid.nodes[tX, tY].pos).realPos);
                        pPos = closedSet.Find(a => a.pos == grid.nodes[tX, tY].pos).parentPos;
                        tX = (int)pPos.x;
                        tY = (int)pPos.y;
                }
        }
        if(!pathCalced) //jos reittiä ei voitu laskea
        {
                print("failed to calc path");
                pathCalced = true;
        }
}
```

A*-Pseudokoodi

```
Lista AS //AukinaisetSolmut
Lista SS //SuljetutSolmut
Taulukko Ruudokko //Liikuttavan alueen pisteet
boolean reitti_löydetty = ei totta

funktio ATähti()
{
        int s //Silmukoiden määrä
        solmu KS //Käsiteltävä solmu
        while(reitti_löydetty) //Tehdään kunnes reitti on löydetty
        {
                Tarkista KS:n ympäriltä kaikki 8 naapuri solmua
                solmu tmpS = naapurisolmu
                JOS(tmpS ei ole AS:ssä JA tmpS ei ole SS:ssä JA tmpS on Ruudukossa)
                {
                        JOS(Ruudukon tmpS alkio on aukinainen)
                                Lisää tmpS AS:n
                }
                Poista KS AS:stä
                Lisää KS SS:n
                JOS(AS on tyhjä)
                {
                        Olet jumissa
                        Riko silmukka
                }
                KS = paras solmu AS:stä //Jolla pienin matkan + askelten summa
                JOS(KS:n matka kohteeseen on 1)
                {
                        reitti_löydetty = totta
                }
                s = s + 1
                JOS(s on jaollinen 100:lla)
                        Odota yhden silmukan verran
        }
}
```