REITINLÖYTÖALGORITMIT

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Ammattikorkeakoulututkinto opinnäytetyö

Tieto- ja viestintätekniikka, insinööri (AMK)

Syksy, 2022

Mikael Forsström

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Tieto- ja viestintätekniikka, insinööri (AMK) Tiivistelmä
Tekijä Mikael Forsström Vuosi 2022
Työn nimi Reitinlöytöalgoritmit
Ohjaaja Toni Laitinen

Tämän opinnäytetyön tehtävänä on tuoda lukijalle esille ymmärrys, miten kaksi eri reitinlöytöalgoritmia toimii ja mitä ne tarvitsevat toimiakseen.

Reitinlöytöalgoritmit tarvitsevat dataa maailmasta lyötääkseen reitin. Tähän tarpeeseen on kaksi suosittua tapaa tuoda kaikki tarpeellinen data reitinlöytöalgoritmille, grid ja NavMesh. Grid on käsitteenä hyvin yksinkertainen eikä ole mitenkään liitetty vain reitinlöytöalgoritmeihin. Ohjelmointityylejä gridin luomiseen on lukemattomia eikä tästä syystä ole mitenkään olennaista koodillisesti selittää, miten grid toimii. NavMesh on taas reitinlöytöön hieman spesiaalisoituneempi menetelmä saada tietoa maailmasta ja on ehdottomasti nykypäivän tärkein menetelmä gridin korvaajaksi. Ongelmana siinä on kuitenkin, että se on monimutkainen ja ohjelmistokoodin määrä on aivan valtava.

Reitinlöytöalgoritmeistä opinnäytetyössä käydään läpi BreathFirst ja A-star, nämä eivät ole ainoat reitinlöytöön käytössä olevat algoritmit vaan on näitäkin lukemattomia eri menetelmiä. Iso osa reitinlöytö algoritmeistä on variaatioita A-star algoritmistä ja kun ymmärtää miten A-star toimii, ei näiden muiden variaatioiden ymmärtäminen vaadi paljoa. BreathFirst on yksinkertaisin esimerkki, miten selvittää reittiä pisteestä *a* pisteeseen *b*. Vaikka BreathFirst ei ole järkevä valinta melkein mihinkään reitinlöytöön liittyvään käyttötarkoitukseen, on se hyvä työkalu ymmärtämään perusperiaatetta reitinlöydön takana.

Tärkein asia kuintenkin reitinlöydössä ja algoritmeissä on ainoastaan ymmärtää niiden toiminta. Melkein koskaan ei tule vastaan tapausta, jossa kannattaisi itse alkaa kehittämään omaa A-star-variaatiota omaan peliin, ohjelmaan tai robotiikkaan. Kun kehittää reitinlöytöä on suositeltavaa käyttää enemmän aikaa gridin tai NavMeshin suunnitteluun, sillä suurin osa kehitysajasta tulee menemään datan tuonnin optimointiin gridiltä tai NavMeshiltä. Kannattaa mieluummin käyttää valmiiksi tehtyjä toimivia plugineja tai Github-projekteja, koska oletuksena on että jos joutuu kehittämään omaa reitinlöytöä varmaan joutuu kehittämään kaikki muut työkalut esimerkiksi pelinkehitykseen ja silti lopputulos tulee todennäköisesti olemaan joko huonompi tai vastaava kun valmiiksi tarjolla olevat työkalut. Haluaako säästää 30-200€ tekemällä 20-2000 tuntia työtä.

Avainsanat Reitinlöytö, a*, a-star, breadth-first, grid
Sivut 32 sivua

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Bachelor of engineer, Information and communication technology Abstract
Author Mikael Forsström Year 2022
Subject Pathfinding algorithms
Supervisor Toni Laitinen

The objective of this thesis is to provide the reader with understanding on how two different pathfinding algorithms work and is needed for them to work properly.

Pathfinding algorithms need data about the world to find any path. For this requirement there is two popular ways to bring the necessary data for the pathfinding algoritm, grid and NavMesh. Grid as a concept is really simple and is not tied not only to pathfinding algorithms. There are countless coding styles for grid and for this reason it is not at all relevant to explain how grid works on code structure. NavMesh on the other hand is a bit more specialized method to provide information about the world to pathfinding algorithms and it is arguably the most important technology to replace grid. Main problem in NavMesh is that it is a lot more complex compared to grid and the amount of code in a working NavMesh generator is immense.

This thesis examines BreathFirst and A-star pathfinding algorithms. These are not the only pathfinding algorithms available, as there are numerous alternatives. Because most pathfinding algorithms are just different variations of A-Star, it is important to understand how A-star works as this will facilitate understanding most of the other variations. BreathFirst is one of the most simple pathfinding algorithms to find a route from point *a* to point *b*. Even if BreathFirst is not a good choice for almost any pathfinding related usage, it is a helpful tool for explaining the basic concepts behind pathfinding.

However the most important thing about pathfinding and algorithms in general is that developer understands how pathfinding works. Most likely there will never be a case where developer should start developing his own A-star variation for game, program or robotics. While developing pathfinding the same amount of planning should be used for grid or NavMesh, since most of the development time will be used for optimizing the data delivery of grid or NavMesh. In most cases developer should use ready made plugins or Github repositories, because as an assumption when developing his own pathfinding algorithms developer will need to develope most of the other tools needed for pathfinding and still the end result will be as good or worse as ready made tools. Does anyone want to save 30-200€ by working for 20-400 hours?

Keywords Pathfinding, a*, a-star, breadth-first, grid
Pages 32 pages

Sisällys

1 Johdanto

Tämän opinnäytetyön tavoitteena on luoda lukijalle ymmärrys miten kaksi eri reitinlöytöalgoritmiä toimii ja mitä ne tarvitsevat toimiakseen. Kun on ymmärtänyt nämä perusperjaatteet ja menetelmät toimivat reitinlöytöalgoritmeissä, opinnäytetyössä halutaan tuoda esille että suurimmassa osassa tapauksia on kannattavaa käyttää jo valmiiksi kehitettyjä koodeja ja muokata niistä omaan projektiin tarvittavat. Kun on sisäistänyt nämä aatteet tulisi osata arvioida paremmin oman projektin tarpeet ja paremmin ymmärtää menetelmiä miten minimoida turhaa laskemista omissa projekteissa.

Vaikka opinnäytetyössä on koodiesimerkit Breadth-first ja A-star algoritmistä näiden koodien täydellinen ymmärtäminen ei ole tavoite, vaan ymmärtää logikka perjaate näiden koodien takana. Kun ymmärtää perjaatteen miksi koodissa tehdään jossain kohdassa jotain kehittäjä ymmärtää miten jatkokehittää koodia eteenpäin.

2 Grid

Reitinlöytöalgoritmit toimiakseen tarvitsevat tietoa ympäröivästä maailmasta. Kun haluaa kulkea paikasta a paikkaan b, haluaa löytää tietynlaisen toivotun reitin. Toivottu reitti voi olla, lyhyin reitti paikan a ja b välillä, helpoin reitti, näyttävin reitti tai optimi reitti. Kun mietimme mitä tietoa tarvitsemme saavuttaaksemme halutun reitin pääsemme seuraaviin parametreihin.

Saadaksemme mitään reittiä täytyy meidän tietää maailman koko, missä kohtaa maailmaa on meidän lähtöpiste ja missä kohtaa maailmaa on meidän kohde. Tästä pystymme tekemään hyvin yksinkertaisen laskelman kuinka pitkä matka on paikan a ja b välillä. Ilman gridiä voimme nyt vetää suoran viivan paikan a ja b välille ja laskea etäisyyden suhteutettuna maailmaan. Tämä on varmasti lyhyin reitti mutta pitää sisällään pelkästään etäisyyden 2 pisteen välillä 2D maailmassa. Tämän voimme laskea kaavalla 1. (Kaava 1.)

Kaava 1. Etäisyys 2 pisteen välillä.

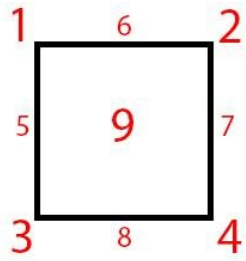
$$\text{Etäisyys} = \sqrt{[(x_2 - x_1)^2 + (y_2 - y_1)^2]}$$

Laskussa syötetään lähtöpisteen sijainti x_1 y_1 jossa x1 on lähtö pisteen sijainti x akselilla ja y1 on lähtöpisteen sijainti y akselilla. Sama tehdään kohde pisteelle x2 y2 jotka ovat kohdepisteen sijainti x ja y akselilla. Kun laskemme laskutoimituksen loppuun saamme etäisyyden pisteiden välillä. Sitten voimme vielä jakaa etäisyyden keskinopeudella jolloin saamme arvion ajasta joka menee kulkea pisteestä *a* kohteeseen *b*.

Tietokone ei kuitenkaan tiedä itse miten suhteuttaa etäisyyttä pisteen a ja b välille vaan tarvitsee informaatiota millä saada suhteutettua maailma numeroiksi joista saadaan etäisyydet laskettua.

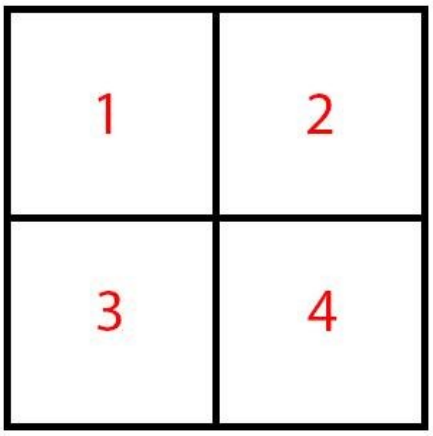
Jos otetaan neliön muotoinen esimerkki. (Kuva 1.)

Kuva 1. Ruutu.



Neliössä on neljä kulmaa, neljä sivua ja yksi keskipiste. Kun syötämme sivulle pituuden saamme laskettua pituuden neliön jokaiseen pisteeseen. Jos lisätään 3 neliötä esimerkkiin saamme hyvin yksinkertaisen 2*2 2d gridin. (Kuva 2.)

Kuva 2. 2*2 grid.



Nyt kuljettavien reittien määrä kasvaa huomattavasti, voimme kulkea neliöiden sivujen kautta, voimme kulkea neliöiden keskustojen kautta tai molempien. Ehdottomasti yleisin on kulkeminen neliön keskustasta toiseen, koska jos myöhemmin ottaa esteet mukaan gridille ja estää pääsyn jollekin neliöille on erittäin paljon monimutkaisempaa välttää reitin laskemista 8 neliön sivuun, kun taas estää reitin lasku 1 neliön keskustaan.

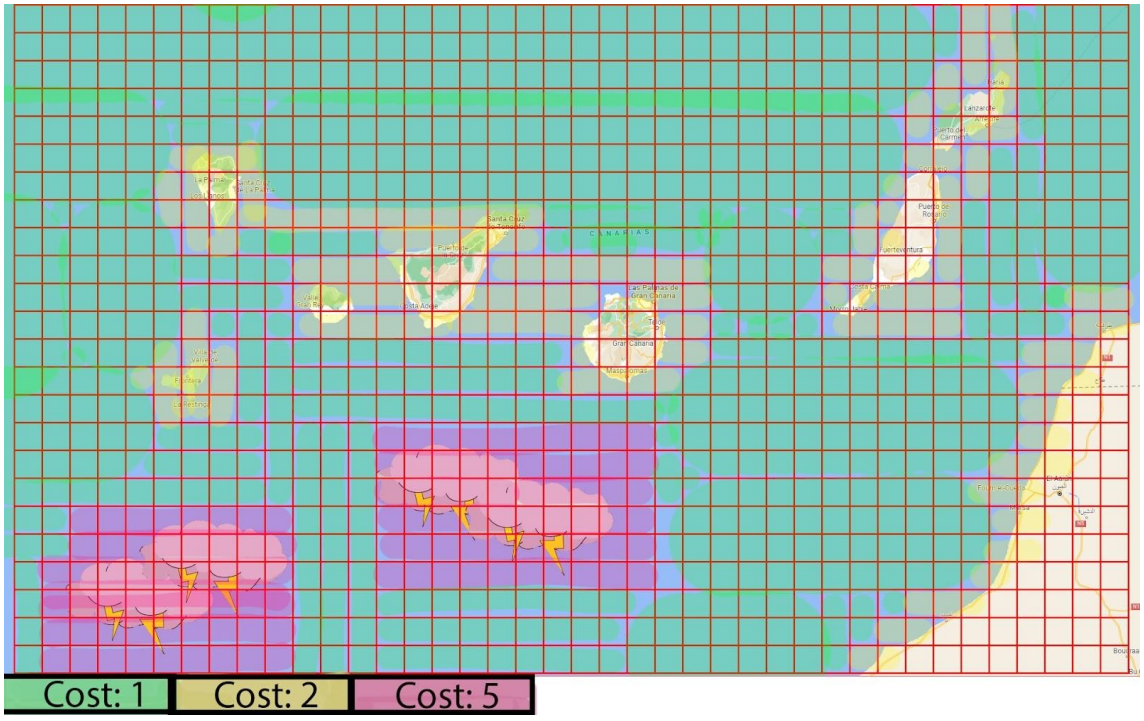
Olemme nyt käyneet läpi perus periaatteen gridistä, mutta aikaisemmin käydyssä gridissä ei ole ollu mitään tietokoneelle tunnistettavia tunnisteita. Kuten aikaisemmin mainittiin:

Tietokone ei tiedä maailmasta mitään. Tähän voi lisätä myös että tietokone ei näe mitään. Kaikki aikaisemmin käsitellyt mitta käsitteet on ollut mitä ihminen näkee kun se katsoo näytölle tai paperille. Koska emme ole kertoneet tietokoneelle koko ruutujen olemassaolosta vaan on tämä osa koko gridin periaatetta ja joutuu tämän määrittelemään koodia kirjoittaessa.

Kun meillä on koordinaattipisteet jokaisella ruudulla ruudukossa pystymme kertomaan, että olemme parhaillaan ruudussa x=2y=6 jolloin tietokone tietää missä ruudussa on aloituspiste. Voimme myös lisätä lisää parametreja ruutuihin kuten: minkä kokoinen ruutu on, pystyykö ruutuun liikkua, tulisiko ruutua välttää ja millä painoarvolla.

Nyt alkaa olla grid ruudukko, josta pystymme kertomaan tietokoneelle huomattavasti tietoa maailman tilanteesta ja mihin suuntaan tietokoneen kannattaa ohjata käyttäjä halutun reitin saavuttamiseksi.

Kuva 3. Grid jossa on painoarvoja ruuduilla.

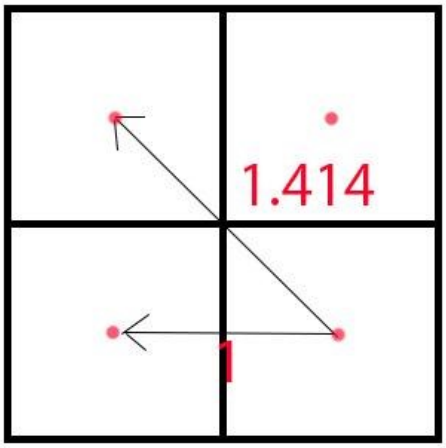


Kun käytämme aikaisemmin mainitsemia menetelmiä kuten painoarvoja kuvaamaan kuinka vaikea ruudun läpi on liikkua. Jos haluaisimme tarkistaa kuinka vaikea olisi kulkea gridin

ruudun x=21 y=13.Huomaamme ruudun olevan vaikeakulkuinen eli painoarvo on = 2, mutta mikäli ruudun läpi liikkuminen säästää 2 normaalin ruudun läpi liikkumisen kannattaa ruudun läpi liikkua.

Gridissä kannattaa myös huomioida liikkeen suunta. Gridissä voi liikkua kolmeen eri suuntaan vaaka, pysty tai viistoon. Viistoon kulkiessa matka neliön kulmasta vastakkaiseen kulmaan on 1.414 kertaa pidempi kuin jos menisit vaaka tai pystysuunnassa. Arvo 1.414 tulee kun otetaan luvusta 2 neliöjuuri ja luku 2 tulee matkasta joka tehtäisiin mikäli ei pystyisi viistoon kulkemaan eli yksi vaaka suuntaan ja yksi pysty suuntaan.

Kuva 4. Liikkeen kustannukset eri suuntiin.



Gridissä ei ole kuitenkaan pakko kulkea ruudusta ruutuun vaan vaan gridiin pystyy myös konfiguroimaan reitin kulkemaan reittipisteiden kautta määrittelemällä tietyt ruudut kuljettaviksi pisteiksi ja määrittelemällä kuljettavaksi pisteestä pisteeseen. Tällöin alue on rajattu gridiin mutta kuljettavat reitit on ennalta määritelty. Tätä voi olla järkevä ratkaisu vaikka jos miettii purjeveneellä purjehdittavia reittejä. Optimaalisesti meret ylitetään pasaatituulien kohdalta, jolloin tuuli puhaltaa yleensä optimaalisesta suunnalta ja tämän takia kuljettavat reitit ovat hieman rajalliset. Tämän takia purjehtijan reitti gridiin olisi optimaalista määrittää ainoastaan tietyt kohdat reitti pisteiksi jolloin gridin optimaalinen reitti olisi ainoastaan mahdollista pasaatituulien taikka virtauksien kohdalta.

Gridin optimointi tapoja on yhtä paljon kuin on tekijöitä. Optimointi menetelmien hyödyt tulisi laskea ennen menetelmän implemoimista ja jokaisen tulisi ymmärtää että mihin omassa projektissa suurin laskenta aika menee. Otetaan esimerkiksi gridi jonka koko on 2000*2000. Pahimmillaan voimme joutua etsimään reittiä 4 miljoonan pisteen kautta. Yhdellä reittiä etsivällä oliolla tämä ei välttämättä ole ylitsepääsemätön laskenta aika, mutta jos reittiä etsiviä olioita on 100 samanaikaisesti ja haluttu kohde piste muuttuu 30 sekunnin välein ei välttämättä pysy laskentateho enää mukana ja ohjelma tulee kaatumaan.

Yhtenä optimointi vaihtoehtona voi miettiä voiko gridin kokoa pienentää 1000*1000. Tällä saamme heti puolitettua laskenta ajan.

Toinen vaihtoehto voi olla, että jos gridiä on mahdollista piirtää uudelleen muutosten sattuessa esim RTS pelissä, ei ruutujen tarvitse olla saman kokoisia ja voi ruudut olla pienempiä lähellä reunoja, kulmia ja tapahtuma sijainteja.

Kolmantena vaihtoehtona abstrakti alue.

Abstraktilla alueella tarkoitetaan alueen jakamista suurempiin alueisiin, jotta saadaan rajattua reitin laskeminen pieniin pienempiin alueisiin. Jos otamme vaikka esimerkin 1000*1000 alueen ja jaamme tämän 100*100 abstraktiin alueeseen saamme rajattua laskettavan alueen 10 kertaa pienemmäksi.

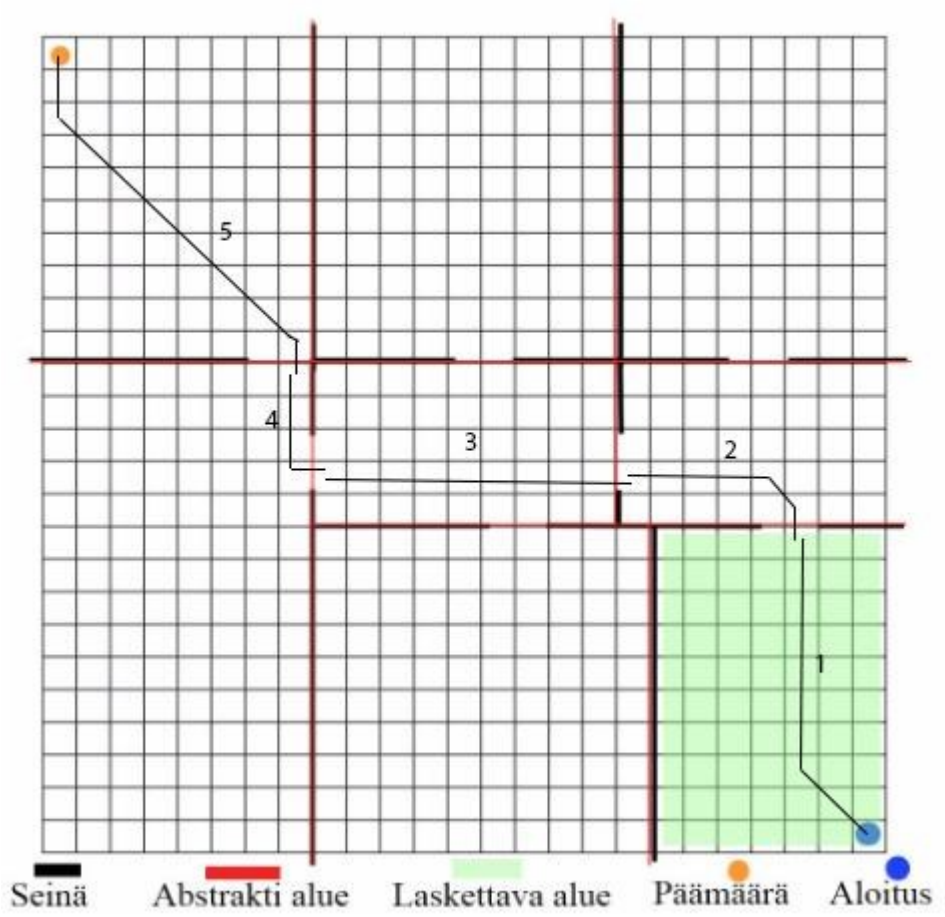
Abstrakti alue toimii seuraavalla tavalla. Joka kerta kun reittiä etsivä oli haluaa löytää reitin pisteestä *a* pisteeseen *b* etsii tämä ensin abstraktilla alueella reitin pisteiden a ja b välillä. Tästä tulee hieman ylimääräista tallennettavaa dataa jokaiselle oliolle, koska olion pitää muistaa hänelle reititetty reitti abstraktilla kartalla, sillä reittiä etsivä olio alkaa tämän jälkeen etsimään optimaalista reittiä ainoastaan siitä abstraktista ruudusta jossa olio parhaillaan on seuraavaan abstraktiin ruutuun ja jatkaa niin kauan kunnes olio on kulkenut jokaisen abstraktin ruudun läpi kohteesta a kohteeseen b.

Lisähyötyä tämän kaltaisesta optimoinnista tulee kun määränpää voi muuttua kesken reitin kulun tai reitille tulee uusia esteitä kesken reitin kulun tai vaikka reitti pisteiden välillä ei ole enää mahdollinen. Koska emme ole laskeneet koko reittiä pisteiden a ja b välillä vaan

olemme laskeneet vain 1/10 yhdellä kerralla emme hukkaa kaikkea laskentaan käytettyä aikaa joka kerta kun reitti muuttuu.

Abstrakti ei ole vain gridissä toimiva käsite vaan tätä voi hyödyntää monissa muissakin osa alueissa niin reitinlöydössä kuin muussakin laskennassa. Esimerkiksi NavMeshi:ssä alueen pystyy jakamaan pienempiin laskettaviin alueisiin. (Kuva 5.)

Kuva 5. Gridi jaettu abstrakteihin alueisiin.



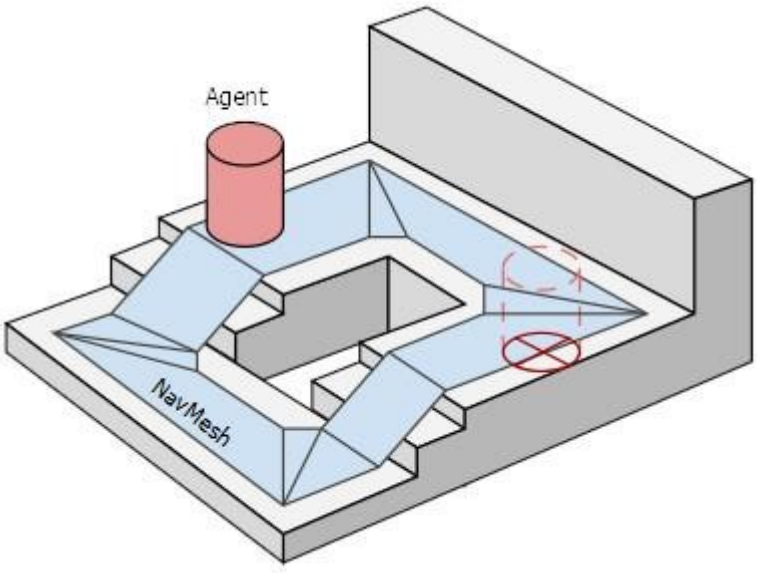
3 NavMesh

NavMesh on toinen tapa tuoda tietoa maailmasta reitinlöytö algoritmille. Gridin ruudukon sijasta NavMesh esittää maailman 2-D polygoni verkolla. Koska NavMesh on polygon verkko voi esimerkiksi kaltevia pintoja lisätä alueiksi joilla on mahdollista liikkua, jolloin liikuttavat alueet eri tasoisissa kohdissa saa sulavasti liitettyä toisiinsa.

NavMesh on huomattavasti suositumpi navigoinnin tapa koska kaikki pelikehitys alustat sisältävät yleensä automaattisen NavMesh pluginin/utility ominaisuuden. Näitä automaattisia ominaisuuksia on useimmiten kannattava käyttää “As is” periaatteella, eikä kannata yrittää edes muokata niitä. Nämä valmiit ominaisuudet toimivat erittäin hyvin, ne ovat hyvin optimoituja ja niitä on helppo käyttää.

NavMesh on erittäin uusi käsite joka on alunperin otettu käyttöön vasta 1984 robotiikkaan nimellä “meadow mapping” Ronald C. Arkining teknillisen raportin pohjalta (Arkin 1986.). Itse Navigation mesh nimellä videopeleihin vasta vuonna 2000 Greg Snookin artikkelin "Simplified 3D Movement and Pathfinding Using Navigation Meshes" pohjalta (Golodetz 2013.). Koska NavMesh ei todellakaan ole mikään vanha menetelmä, NavMesh menetelmiin tulee vielä parannuksia vaikka valitettavasti avoimen lähdekoodin NavMesh järjestelmiin nämä päivitykset ei yleensä yllä koska kehitys tapahtuu pääsääntöisesti suurissa yrityksissä kuten Unreal Engine pelikehitys alustalla tai Unity alustalla.

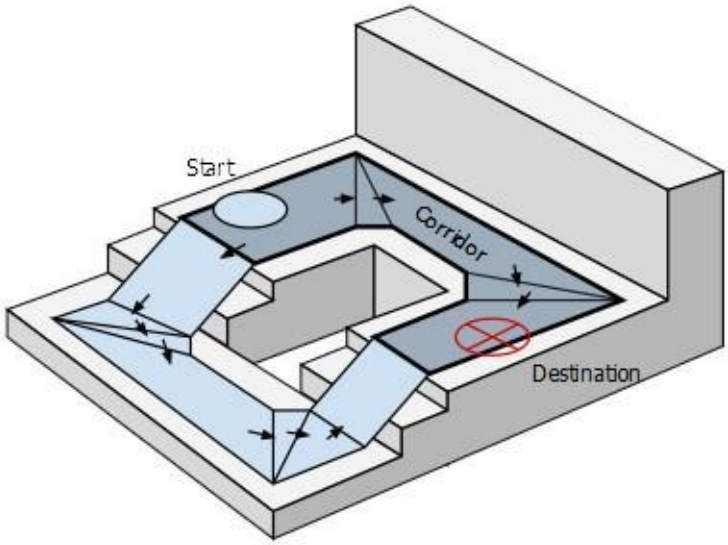
Kuva 6. NavMesh (docs.unity3d.com, (n.d.)).



NavMesh kertoo liikkuvalle oliolle 3 asiaa: liikuttavat alueet, miten alueet liittyvät toisiinsa ja miten vaikeakulkuinen alue on. Liikuttavat alueet ovat NavMeshin jokaiset polygonit. Polygonit vastaavat NavMeshissä samaa toimintaperiaatetta kun gridissä ruudut. Polygonista toiseen liikutaan polygonin reunaviivasta toisen polygonin reunaviivaan. Kun olio on jonkin

polygonin sisällä voi olio liikkua vapaasti suuntaan tai toiseen koska polygonissa ei ole esteitä eikä navigointia tarvita.

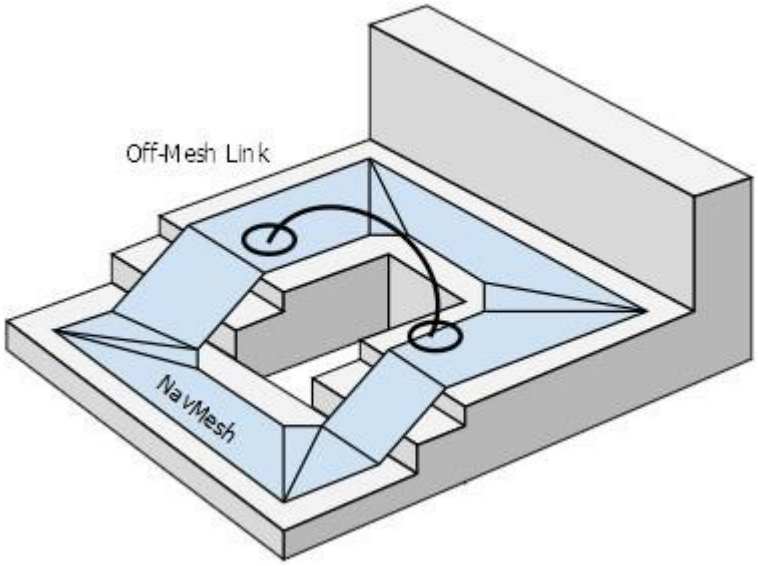
Kuva 7. NavMesh navigoivat suunnat (docs.unity3d.com, (n.d.)).



NavMesh ei yleensä pidä sisällään mitään etäisyys tietoja koska jokainen polygon voi olla eri kokoinen ja voi muuttua koska tahansa. Tästä syystä kuljetun etäisyyden saaminen nav meshillä tulisi laskea määrittämällä maailman koko ja laskemalla jokaisen matkan pituus nav meshin sisällä esim jos maailman koko on 100m*100m tulisi laskea vektori etäisyys NavMesh pisteestä *a* pisteeseen *b* ja pisteestä *b* pisteeseen *c*, josta saamme sitten helposti summattua vektoreiden pituudet yhteen.

NavMeshissä alueet voivat liittyä toisiinsa usealla tapaa. Polygoni liittyy toiseen polygoniin jolloin polygonista 1 voi navigoida polygoniin 2. NavMesh alueen 1 ja 2 välille voi määrittää erityisen kulkureitin jonka avulla NavMesh alue 1 on liitetty NavMesh alueeseen 2 jolloin näitä 2 aluetta voi kohdella eri abstrakteina alueina ja välttää ylimääräistä laskemista. Alueiden välille voi myös luoda niin sanotun off-mesh linkin jolla kulkeva olio voi kulkea NavMeshilla linkitettyyn kohtaan ja esimerkiksi hypätä toiselle NavMesh alueelle. (Kuva 8.)

Kuva 8. NavMesh off-mesh link (docs.unity3d.com, (n.d.)).



NavMeshin luonti raskas laskutoimitus ja tehdään usein ainoastaan kerran ohjelman käynnistyessä tai jos NavMeshin ei tarvitse koskaan muuttua voi sen tehdä kertaalleen ohjelmaa luodessa. Jos ison kartan kokoisen NavMeshin loisi uudestaan joka kerta kun kartta muuttuu pelin aikana, tulisi peliin yllättäviä 0.5-30 sekunnin pysähdyksiä aina kun kartta muuttuu. Tämä on yleensä jokaisen pelin / NavMeshiä hyödyntävän ohjelman suunniteltava itse erikseen. Normaalisti nämä muutokset voi laittaa laskeutumaan erilliseen threadiin ennen kun nämä muutokset ovat tapahtuneet, jolloin vain murto osa tehosta menee muutoksien laskemiseen ja NavMeshin muutokset tulevat tekoälylle tarpeeksi nopeasti, ettei pelaaja ehdi huomata mitää outoa toimintaa tekoälyssä tai mitään pätkimistä pelissä vaikka muutoksia NavMeshiin on laskettu.

Toinen yleinen optimointi on jos esimerkiksi pelissä on erittäin suuri liikuttava alue, voi olion navigointi aktivoitua vasta kun pelaaja pääsee tarpeeksi lähelle oliota. Tällöin on myös hyvä huomioida, että jos pelaaja ei näe oliota ollenkaan ei kannata edes piirtää tekoälyn 3d malleja jolloin säästyy huomattava määrä video muista.

On myös mahdollista tällaisissa erittäin suurissa navigoitavissa alueissa generoida jokaiselle tekoäly olioille omaa lähiympäristön NavMeshiä reaaliajassa jos NavMesh on mahdollista pitää riittävän pienenä. Tällaisissa tapauksissa tulisi kuitenkin huomioida, että jos on

useampi tekoäly navigoitavassa kohdassa tulisi niiden huomida toisensa ja hyödyntää naapuri olionsa generoimaa NavMeshiä välttääkseen päällekkäin menemistä.

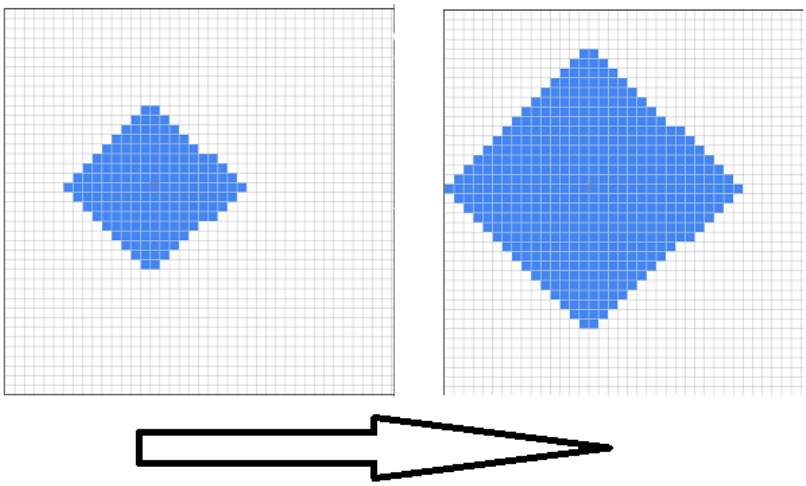
Navigoimisessa ja NavMeshissä etenkin kannattaa pitää mielessä, että yleisin ihmisen/pelaajan immersiota hajottava asia on reitinlöytö ja reitinlöytöön liittyen NavMeshin ongelmat. Koska jos ei tekoäly osaa edes kävellä suoraa viivaa pisteen a ja b välillä on navigoimisessa monimutkaisemmissa ympäristöissäkin erittäin suuria ongelmia. Reitinlöytö on useimmiten yksi pelin raskaimmista prosesseista ja kun 10 tekoälyä etsii reittiä joka on ihmisen silmissä uskottava reitti on monta huomioitavaa asiaa.

Kuuluisa avoimen lähdekoodin NavMesh on Mikko Monosen kehittämä Recast & Detour. Recast & Detour Navmesh teknologia on käytössä Unity alustalla ja osittain myös Epic Gamesin UE4 Alustalla (Recastnavigation.Nav-mesh toolset for games, 2022).

4 Breadth-first search

Breadth-first eli Leveyssuuntainen läpikäynti on yksi yksinkertaisimmista algoritmeistä löytää reitti pisteen a ja b välille. Breadth-first toiminta perustuu kaikkien läpi käytävien solmujen läpikäymistä järjestyksessä kunnes saavutaan tavoite solmuun jolloin lyhyin reitti on löytynyt pisteen a ja b välille jos pisteiden a ja b välillä on reitti mahdollinen. Ongelmana Breadth-first algoritmissä on että Breadth-first algoritmi tarkistaa kaikki solmut jokaiseen suuntaan piittaamatta kasvaako etäisyys kohteeseen, tämä johtuu siitä että Breadth-first algoritmi ei “näe” etäisyyksiä ollenkaan vaan näkee ainoastaan mitkä solmut ovat nykyisen tarkastettavan solmun naapureita ja mikä solmu oli seuraavaksi tarkastettavien listalla.

Kuva 9. Breadth-first hakemassa reittiä.

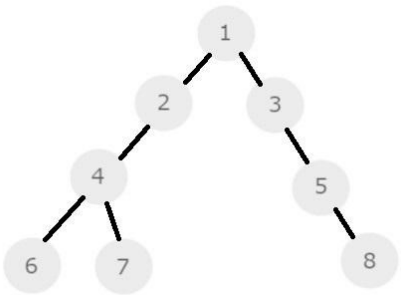


4.1 Breadth-first graafi

Breadth-firstin periaate on helpoin kuvata läpikäytävänä graafin kuvasarjana Breadth-firstin näkökulmasta. Kun tässä esimerkissä ei ole haettavaa reittiä vaan käydään vain läpi miten Breadth-first käy graafin läpi solmusta 1 solmuun 8.

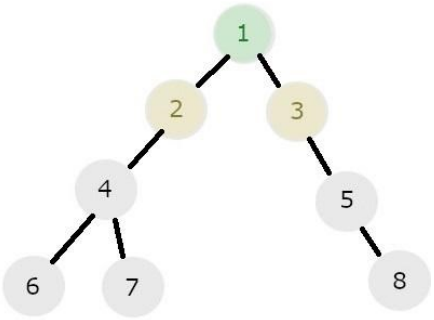
Aloitamme yksinkertaisella graafilla jossa on 8 solmua. (Kuva 10.)

Kuva 10. Breadth-first graafi solmuista.



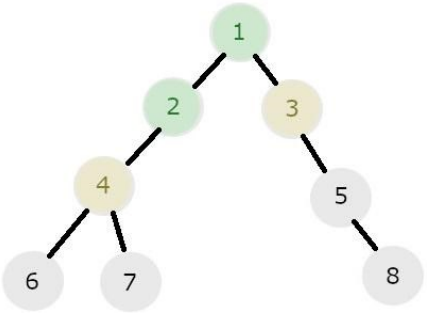
Kun aloitamme Breadth-first etsinnän solmusta 1 siirtyy tämä solmu ensiksi tarkastettavaksi jonka jälkeen lisäämme kaikki solmun 1 naapurit listalle jonottamaan tarkastettavaksi pääsemistä nämä solmut ovat tässä tapauksessa on 2 ja 3.

Kuva 11. Breadth-first graafi solmuista vaihe 1.



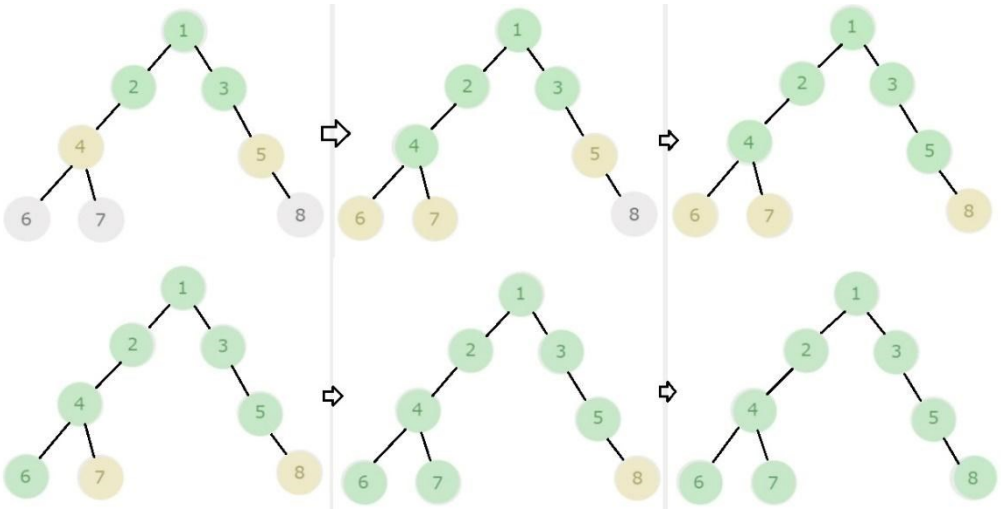
Kun solmun 1 naapurit on lisätty jono listalle, merkitään solmu 1 vierailuksi katso kuva 11. Tämän jälkeen Breadth-first ottaa listasta seuraavan jonossa, joka tässä tapauksessa on 2 ja taas lisätään jono listalle kaikki solmun 2 naapurit jotka on tässä vain solmu 4. (Kuva 12.)

Kuva 12. Breadth-first graafi solmuista vaihe 2.



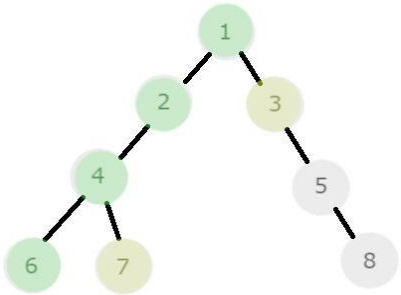
Kun solmun 2 naapurit on lisätty listalle merkitään solmu 2 vierailuksi. Tämä toistuu kunnes kaikki solmut esimerkissä on käyty läpi joka näyttää tässä esimerkissä kuvan 13 mukaiselta. (Kuva 13.)

Kuva 13. Breadth-first graafi. Solmujen loput vaiheet.



Breadth-first algoritmia ei pidä kuitenkaan sekoittaa Depth-first search algoritmiin (syvyyssuuntainen läpikäynti). Näiden kahden ero on pieni mutta toiminta täysin päinvastainen. Molemmat käyvät solmu graafia läpi järjestyksessä ilman mitää “näköä”, mutta kun Breadth-first käy järjestyksessä taso kerrallaan, Depth-first käy nimensä mukaisesti syvyyssuunnassa kaikki polut yksitellen järjestyksessä. Tämä pieni ero tekee Depth-first algoritmistä lähes käyttökelvottoman reitinlöytöön gridillä taikka NavMeshillä.

Kuva 14. Depth-first graafi solmuista vaihe 4.



4.2 Breadth-first koodiesimerkki

Kuva 15. Breadth-first algoritmi koko koodi (Pathfinding-Visualiser, 2022).

```
async bfs_Search(start, end) {

    let queue = new Queue();
    queue.enqueue(start);

    while (!queue.isEmpty()) {
      let node = queue.dequeue();

      if (node == end) {
        let current = end;
        let path = new Array();
        while (current != start) {
          current = current.cameFrom;
          path.push(current);
        }
        break;
      }

      let neighbors = this.returnNeighbors(node);

      for (let i = 0; i < neighbors.length; i++) {
        if (!neighbors[i].visited && neighbors[i].type != "Wall") {
          neighbors[i].visited = true;
          neighbors[i].cameFrom = node;
          queue.enqueue(neighbors[i]);
        }
      }
      await new Promise<void>(resolve =>
        setTimeout(() => {
          resolve();
        }, 0)
      );
    }
  }
```

Esimerkki muodostuu gridistä joka on 95 ruutua leveä ja 40 ruutua korkea. Gridin jokainen ruutu on objekti. (Kuva 15.)

Kuva 16. Ruudun datan kuvaus (Pathfinding-Visualiser, 2022).

```
shapes[i][j] = { x, y, i, j, type, F, G, H, neighbors, cameFrom, visited };
```

1. Ruutu objekti pitää sisällään muuttujat x ja y jotka ovat ruudun koordinaatit gridillä.

2. i ja j jotka ovat ruudun indexit listalla.
3. type on ruudun tyyppi kuten aloitus, lopetus tai seinä.
4. F,G ja H ovat A* algoritmin tarvitsemia muuttujia.
5. neighbors on ruudun viereiset ruudut.
6. cameFrom on tyhjä muuttuja johon kirjoitetaan, mistä ruudusta tultiin.
7. visited kertoo onko ruudussa jo käyty.

Yläpuolella oleva koodiesimerkki on Breadth-first algoritmistä Typescript koodikielellä. Koodiesimerkistä näemme että funktio ottaa sisään aloitus ja päätös pisteen. Aloituspiste on gridin ruutu eli shapes[4][4] ja lopetuspiste on gridin ruutu eli shapes[90][35].

Kuva 17. Breadth-first laskennan aloitus ja muuttujien alustus (Pathfinding-Visualiser, 2022).

```
async bfs_Search(start, end) {
    let queue = new Queue();
    queue.enqueue(start);
```

Breadth-first algoritmi aloittaa alustamalla aluksi uuden jonon (Queue()). Jono on sama kuin jono oikeassakin maailmassa jonossa on esineitä, jonoon voi tulla jonottamaan viimeiseksi, jonossa on ensimmäinen, jonosta voi poistua ja jono voi olla tyhjä. Kun jono on alustettu lisätään jonoon aloituspiste. (Kuva 17.)

Kuva 18. Breadth-first ylimmäinen while-silmukka ja tarkistus onko nykyinen solmu end solmu (Pathfinding-Visualiser, 2022).

```
    while (!queue.isEmpty()) {
      let node = queue.dequeue();

      if (node == end) {
        let current = end;
        let path = new Array();
        while (current != start) {
          current = current.cameFrom;
          path.push(current);
        }
        break;
      }
```

Tämän jälkeen Breadth-first aloittaa while-silmukan, jossa silmukan jokaisella kierroksella tarkistetaan onko jono tyhjä. Jos jono ei ole tyhjä suoritetaan silmukan sisältö. Silmukassa aloitetaan ottamalla jonon ensimmäinen pois jonosta ja tallentamalla tämä muuttujaan. Tulee huomioida että tämä jonon ensimmäinen on aloitusruutu joka pitää sisällään kaikki aloitusruudun datan. (Kuva 18.)

Tämän jälkeen tarkistamme onko parhaillaan tarkastettava ruutu päätös ruutu. Jos parhaillaan tarkasteltavana oleva ruutu on päätös ruutu, merkitsemme muuttujaan nimeltä current tarkasteltavana olevan ruudun päätös ruuduksi ja aloitamme uuden arrayn johon seuraavassa vaiheessa merkitsemme oikean reitin.

Tämän jälkeen luomme while-silmukan jossa käymme reitin läpi käänteisessä järjestyksessä. While-silmukan jokaisessa silmukassa tarkistamme ensiksi onko nykyinen ruutu aloitusruutu, jos ei, niin merkitsemme, mistä tulimme nykyiseen ruutuun (current.cameFrom) seuraavaksi current ruuduksi jonka jälkeen puskemme sen oikeaksi reitti ruuduksi path listaan. Tämän jälkeen While-silmukka pyörii niin kauan läpi ruutuja mistä tultiin, kunnes se on merkinnyt kaikki ruudut, josta päästiin aloitusruutuun. Tämän jälkeen on koko reitti löytynyt ja alkuperäinen while-silmukka loppuu break komennolla.

Koska ensimmäinen ruutu jota tarkasteltiin ei kuitenkaan ollut lopetus ruutu jatkuu ensimmäisen while-silmukan toiminta

Kuva 19. Breadth-first naapureiden lisääminen jonoon (Pathfinding-Visualiser, 2022).

```
let neighbors = this.returnNeighbors(node);

for (let i = 0; i < neighbors.length; i++) {
  if (!neighbors[i].visited && neighbors[i].type != "Wall") {
    neighbors[i].visited = true;
    neighbors[i].cameFrom = node;
    queue.enqueue(neighbors[i]);
  }
}
```

Breadth-First koodiesimerkissä seuraavaksi otamme parhaillaan tarkasteltavana olevan ruudun naapurit jotka ovat listan shapes[i][j] edelliset ja seuraavat ruudut, eli:

1. shapes[node.i-1][node.j] jossa node.i on nykyisen ruudun indexi i listalla ja node.j joka on nykyisen ruudun indexi j listalla. Koodi ottaa node.i-1, jotta saamme yläpuolella olevan ruudun.
2. shapes[node.i+1][node.j] Tässä saamme nykyisen ruudun alapuolella olevan ruudun.
3. shapes[node.i][node.j-1] Tässä saamme nykyisen ruudun vasemmalla olevan ruudun.
4. shapes[node.i][node.j+1] Tässä saamme nykyisen ruudun oikealla puolella olevan ruudun.

(Kuva 19.)

Kun ohjelmalla on kaikki neljä viereistä ruutua toistetaan kaikkien läpi ja tarkistetaan ettei ruuduissa ole käyty eikä ruutu ole seinä. Mikäli ruudussa on jo käyty tai ruutu oli seinä ohitetaan ruutu. Jos ruutu ei ollut kumpikaan näistä merkitään, että olemme jo käyneet ruudussa ja merkitsemme ruutuun että tulimme nykyisestä ruudusta. Tämän jälkeen lisäämme ruudun jonon viimeiseksi. Toistamme tämän kaikille neljälle ruudulle. Kun kaikki naapurit on lisätty jonoon aloitamme while-silmukan alusta.

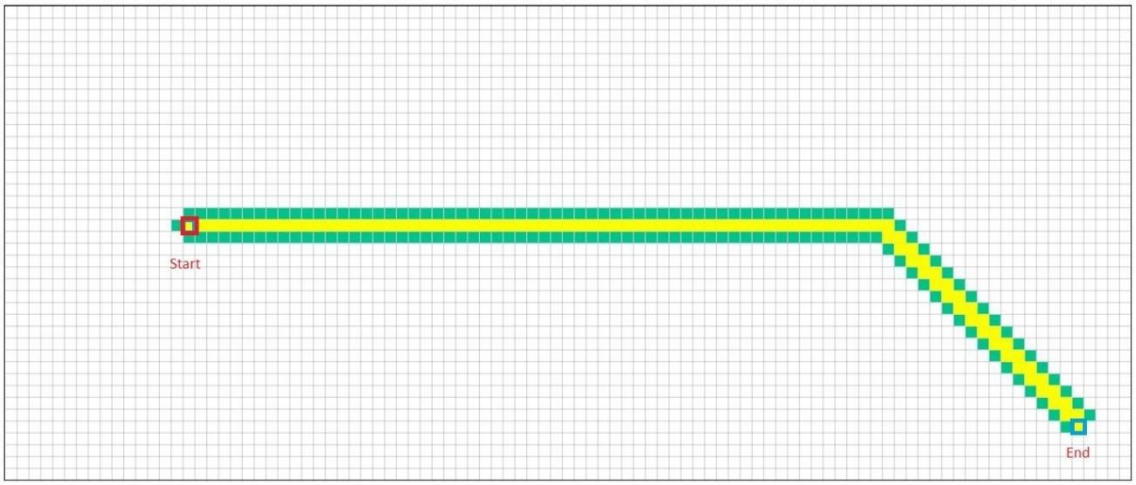
Toisella kierroksella on jonossa jo mahdollisesti 4 ruutua, 3 kierroksella 16 ja 4 kierroksella 64. while-silmukka jatkuu niin kauan kunnes päätös piste on löydetty tai kaikki mahdolliset ruudut on tarkistettu.

5 A-star / A*

Kun nykypäivänä kuulee reitinlöydöstä puhutaan yleensä A-star(A*) reitinlöytöalgoritmistä. A-star on heuristinen reitinlöytöalgoritmi, tämä tarkoittaa että A-star tekee datan tukemia arvauksia lyhyimmästä reitistä. Tällä taas tarkoitetaan ettei A-star etsi reittiä väärään suuntaan jos sen data “Näkee“ lyhyemmän reitin. Koska A-star ei käy tutkimassa ylimääräisiä reittejä jotka on sen mukaan pidempiä tai sokkeloiden tapauksessa vääriä. A-star käy salamannopeasti yhtä suuntaa läpi kunnes se löytää oikean reitin tai toteaa reitin olleen väärä ja alkaa tutkimaan datan mukaan seuraavaksi parasta reittiä kunnes oikea reitti on löydetty jos sellainen on mahdollinen. Koska A-star ei tarkasta ylimääräisiä reittejä on

normaalia, että A-star tekee suoran reitin pisteestä *a* pisteeseen *b* tarkistamatta mitään muita reittejä jos ei reitillä ole mitään esteitä.

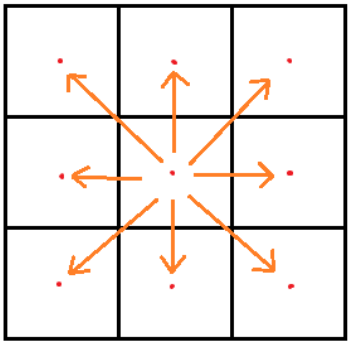
Kuva 20. A-star etsimässä reittiä ilman esteitä.



5.1 A-star toiminnallisuus kuvaus

A-starista tärkein asia ymmärtää on miten naapuri solmut (ruudut) toimivat reitin etsinnässä, näihin asioihin kuuluu, mahdolliset kulkureitit, ruutujen arvot ja A-Starin käyttämän reitin analyysiarvot. Aloitetaan yksinkertaisimmasta, A-starissa voi kulkea vaakasuuntaan, pystysuuntaan ja viistoon. (Kuva 21.)

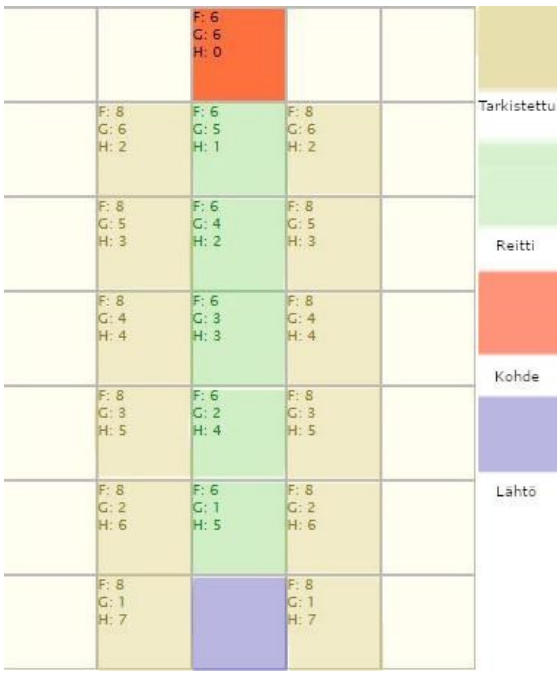
Kuva 21. A-star liike jokaiseen suuntaan.



Etäisyys naapuri ruutuun pysty ja vaakasuunnassa on 1, mutta etäisyys viistoon kulkiessa on hieman suurempi 1,414. Jos emme pystyisi kulkemaan viistoon ja haluaisimme päästä

oikealla yläpuolella olevaan ruutuun joutuisimme kulkemaan 2 ruudun kautta, ensiksi joko ylös ja sitten oikealle tai oikealle ja sitten ylös. Etäisyyden oikealla yläpuolella olevaan ruutuun saamme kun otamme luvusta 2 neliöjuuren, tästä jää jäljelle 1,414. Voimme vielä pyöristää tämän luvun 1,4. Yksinkertaisuuden nimissä voimme vielä kertoa luvut 10, niin pystymme käyttämään kokonaislukuja.

Kuva 22. A-star F, G, H arvot.



A-Starista tulee tutuksi heuristiset analyysiarvot G arvo, H arvo ja F arvo. Nämä arvot ovat keskeisessä roolissa A-starin toiminnallisuudessa, ja ovat suurimpia vaikuttajia A-starin reitinlöytöalgoritmissä.

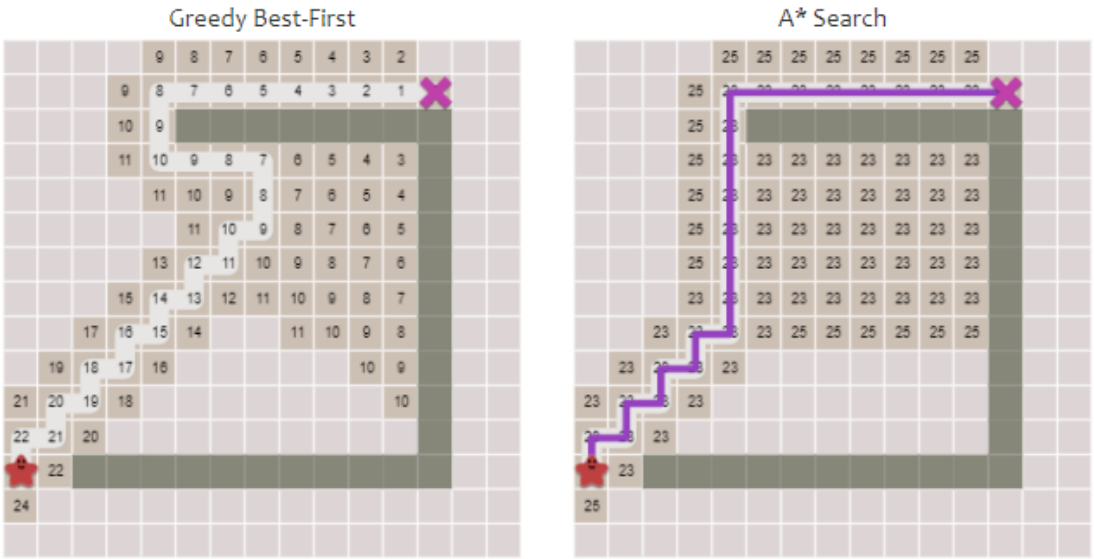
1. G arvo = etäisyys aloituspisteestä.
2. H arvo = etäisyys päätepisteeseen.
3. F arvo = G arvo + H arvo.

G arvo on yksinkertainen selittää kun on käytössä gridi jossa voi liikkua jokaiseen suuntaan ja kaikki arvot on kerrottu kokonais arvoiksi niin kuin aikaisemmin kuvasimme. Jos otamme 5x5 gridin jossa pystyy liikkumaan jokaiseen suuntaan laskemme jokaisella A-starin toisto silmukalla aina kun lisäämme gridin ruudun tarkistettavaksi sen etäisyyden lähtöpisteestä. Eli

jos olemme kulkeneet 3 ylös on G arvo 10*3 = 30. Jos olemme kulkeneet 3 viistoon on G arvo 14*3 = 42.

G arvosta tulee huomioida, että G arvo lasketaan aina kun ruutua ollaan lisäämässä tarkistettavien listalle ja G arvo lasketaan aina kuljetun reitin perusteella. Esimerkiksi kun reittiä lasketaan johonkin suuntaan päivittyy ruutujen G arvo sen perusteella, mistä ruutuun ollaan tultu. Joten reitit päivittyy vastaamaan lyhintä reittiä lähtöpisteeseen aina niitä tarkkaillessa. Tämä on yksinkertaista kuvata muutamalla kuvalla jossa näkyy, mistä mihinkin ruutuun ollaan tultu.

Kuva 23. RedBlobGames A-star jos G arvoa ei päivitä kesken reitin haun (Red Blob Games, 2016).



5.2 A-star heuristinen arvo

H arvo eli heuristinen arvo on A-starin tapa kuvata datan perusteella tehtyä arvausta etäisyydestä tavoite pisteeseen nykyisestä ruudusta. H arvo on A-starin monimutkaisin asia ymmärtää eikä sen toiminnallisuutta yleensä selitetä ollenkaan. Kun puhutaan eri A-star versiosta on yleensä juuri tämä heuristinen arvio johon on tehty muutoksia. Yleensä puhutaan Ahne A-star, yliarvioiva A-star tai aliarvioiva A-star mutta variaatioita on enemmän kuin tekijöitä.

Eri heuristisissa arvio menetelmissä tulee aina ottaa huomioon minkälainen navigoitava alusta on kyseessä. Onko NavMesh, gridi, voiko gridissä kulkea jokaiseen suuntaan vai vain pysty ja vaakasuuntaan, minkä muotoisesta gridistä on kyse, muodostuuko gridi neliöistä, hexagoneista vai octagoneista. Kaikkiin näihin on omanlaisensa variaatiot heuristisiin analyyseihin, joihinkin enemmän kuin toisiin.

Esimerkissä keskitytään 3 yksinkertaiseen neliöistä muodostuvaan gridiin tarkoitettua heuristista arvio menetelmään. Manhattan, Euclidean, Chebyshev.

5.3 Manhattan heuristinen analyysi

Manhattan etäisyyden arvio on yksinkertaisin ja on tarkoitettu grideille, joissa voi liikkua vain pysty ja vaakasuunnassa. Manhattan arvio yleensä yliarvioi etäisyyden päätepisteeseen koska se ei osaa ottaa vaakasuuntaisa liikettä huomioon.

Kuva 24. Manhattan heuristinen analyysi koodiesimerkki (AStar-Typescript, 2022).

```
const dx = Math.abs(currentReviewNode.x - endNode.x);
const dy = Math.abs(currentReviewNode.y - endNode.y);

switch (heuristicFunction) {
  case 'Manhattan':
    return (dx + dy) * weight;
}
```

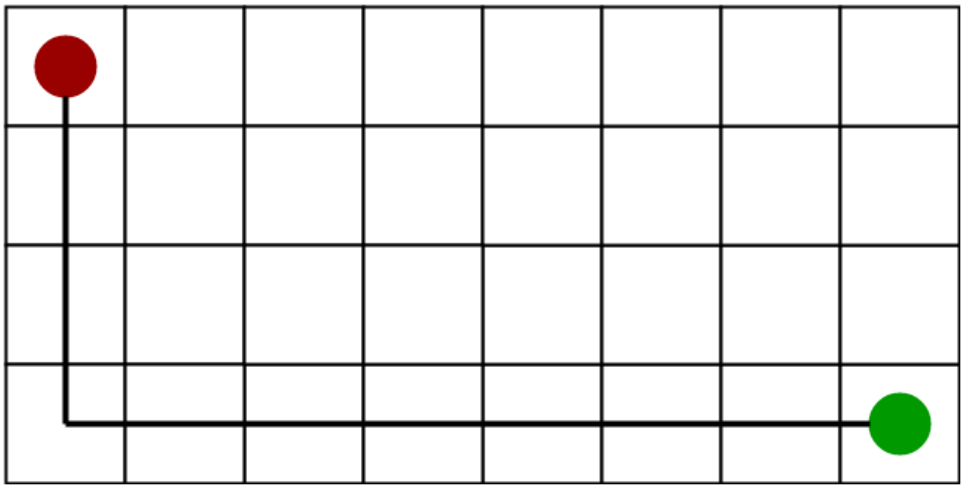
Manhattan heuristic koodiesimerkissä näemme heti kuinka yksinkertainen etäisyys analyysi Manhattan on. Näemme myös syyn minkä takia Manhattan ei sovi viistoon kuljettaviin grideihin. (Kuva 24.)

Käydään koodi läpi kohta kohdalta ja muunnetaan se selkokieliseksi.

1. Tallennamme kokonaisluku arvon currentReviewNode.x - endNode.x.
    a. Tässä otetaan vain parhaillaan lisätyn naapuri ruudun x sijainti ja vähennetään siitä tavoitepisteen x sijainti.
    b. Math.abs palauttaa aina positiivisen kokonailuku arvon (esim -5 = 5).
2. Toistetaan täysin sama Y arvoille.
3. (dx+dy) * weight laskemme aikaisemmin lasketut arvot yhteen ja kerromme painoarvolla.

Visuaalisesti kuvattuna Manhattan näyttäisi tältä. (Kuva 25.)

Kuva 25. Manhattan visuaalinen esimerkki (geeksforgeeks.org, (n.d.)).



5.4 Euclidean heuristinen analyysi

Euclidean heuristinen analyysi on yhtä yksinkertainen kun Manhattan analyysi. Euclidean analyysi on vain suora etäisyys pisteestä *a* pisteeseen *b* olettaen, että reitin kulmalla ei ole väliä. Tämä tapa yleensä aliarvioi reitin pituuden huomattavasti, koska reitinlöydössä gridillä ei yleensä saa liikkua muutakuin rajoitettuihin suuntiin.

Euclidean analyysiä voi käytännössä käyttää missä vain pisteestä *a* pisteeseen *b* etäisyyden mittaamiseen niin kauan kun pisteillä on x ja y arvo.

Kuva 26. Euclidean heuristinen koodiesimerkki (AStar-Typescript, 2022).

```
const dx = Math.abs(currentReviewNode.x - endNode.x);
const dy = Math.abs(currentReviewNode.y - endNode.y);
switch (heuristicFunction) {
  case 'Euclidean':
    return Math.sqrt(dx * dx + dy * dy) * weight;
}
```

Koodiesimerkistä huomaa, että Euclidean analyysi on melkein täysin sama kuin Manhattanin analyysissä. (Kuva 26.)

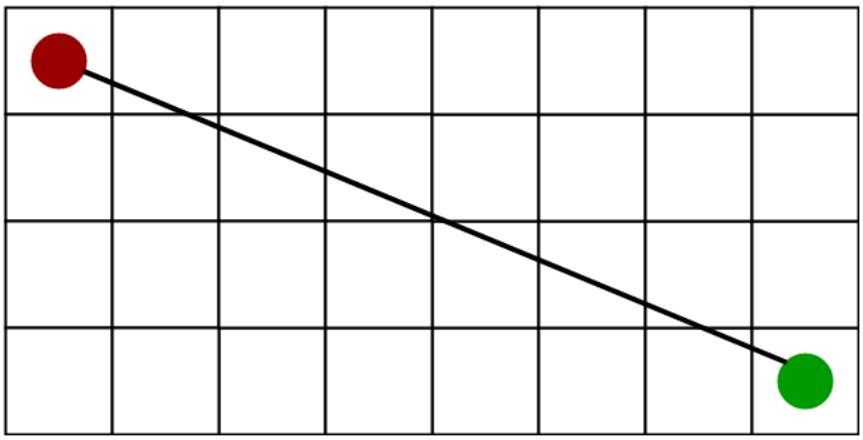
Käydään koodi läpi kohta kohdalta ja muunnetaan se selkokieliseksi.

1. Tallennamme kokonaisluku arvon currentReviewNode.x - endNode.x.
    a. Tässä otetaan vain parhaillaan lisätyn naapuri ruudun x sijainti ja vähennetään siitä tavoitepisteen x sijainti.
    b. Math.abs palauttaa aina positiivisen kokonailuku arvon (esim -5 = 5).
2. Toistetaan täysin sama Y arvoille.
3. Otetaan neliöjuuri (dx*dx+dy*dy) * weight.
4. Tämä funktio voi näyttää tutulta koska se on aivan tavallinen pythagoraan lause. (Kaava 2.)

Kaava 2. Pythagoraan lause.

$$c = \sqrt{a^2 + b^2}$$

Kuva 27. Euclidean visuaalinen esimerkki (geeksforgeeks.org, (n.d.)).



5.5 Chebyshev heuristinen analyysi

Chebyshevin etäisyys on tarkoitettu grideille jossa voi liikkua kaikkiin 8 suuntaan. Chebyshevin etäisyys on myös tunnettu shakki etäisyytenä ja siksi usein kuvataan minimi liikkeillä jota kuninkaalla menee liikkuessa pisteestä *a* pisteeseen *b*. Chebyshevin etäisyyden toiminnallisuus perustuu ajatukseen, että kaikki liikkeet maksavat vain 1, tällöin tarvitsee ainoastaan tarkistaa kumpi on suurempi etäisyys (x2-x1) vai (y2-y1). Koska reitti mihin vain pisteeseen tulee olemaan yhtäsuuri kuin suurempi etäisyys kahdesta.

Kuva 28. Chebyshev heuristinen analyysi visuaalinen esimerkki (Wikipedia, (n.d.)).

Kuvassa 28 voit testata että kulkiessa pisteestä *a* pisteeseen *b*, tarvittavat liikkeet on aina yhtä paljon kuin suurempi arvoista x tai y. (Kuva 28.)

Kuva 29. Chebyshev heuristinen analyysi koodiesimerkki (AStar-Typescript, 2022).

```
const dx = Math.abs(currentReviewNode.x - endNode.x);
const dy = Math.abs(currentReviewNode.y - endNode.y);

switch (heuristicFunction) {
  case 'Chebyshev':
    return Math.max(dx, dy) * weight;
}
```

Chebyshevin etäisyyden koodiesimerkki on myös erittäin yksinkertainen ja hyödyntää täysin samaa periaatetta kuin aikaisemmatkin koodiesimerkit. (Kuva 29.)

1. Tallennamme kokonaisluku arvon currentReviewNode.x - endNode.x.
    a. Tässä otetaan vain parhaillaan lisätyn naapuri ruudun x sijainti ja vähennetään siitä tavoitepisteen x sijainti.
    b. Math.abs palauttaa aina positiivisen kokonailuku arvon (esim -5 = 5).
2. Toistetaan täysin sama Y arvoille.
3. Math.max palauttaa 2 syötetystä arvosta suuremman, esim Math.max(3,5) = 5. Tämän jälkeen taas vain kerromme painoarvolla.

5.6 A-star koodiesimerkki

A-star koodiesimerkki on yksi mahdollisista toteutustavoista, jotta koodista on saatu yksinkertaisempi siitä on poistettu ylimääräisiä asioita jotka eivät vaikuta itse A-starin toimintaan, näitä vaikuttamattomia asioita on esim: tutkittujen reittien piirto, aloitus ja lopetus ruudun indexin etsintä, ymt. Jaetaan koodiesimerkki muutamaan osaan, jotta saadaan yksinkertaistettua toiminnallisuuden läpikäyntiä.

Kuva 30. A-star muuttujien alustus (Pathfinding-Visualiser, 2022).

```
async a_star_search(start, end) {
    let openSet = [];
    let closedSet = [];
    let path = [];
    this.findNeighbors();
    openSet.push(start);
```

Ensimmäisenä A-starissa alustetaan aina listat openSet=[] ja closedSet=[]. Nämä kaksi listaa ovat välttämättömät A-starin toiminnan kannalta. openSet listaan lisätään tarkasteltavana olevan solmun (ruudun) kaikki naapuri solmut (ruudut). Tällä mahdollistetaan se että pystymme koko ajan tarkistamaan tältä openSet listalta muutaman solmun joukosta parhaimman mahdollisen reitin ilman, että tarvitsee käydä tarkistamassa ylimääräisiä solmuja (ruutuja). (Kuva 30.)

closedSet pitää sisällään solmut (ruudut) jotka olemme jo käyneet tarkistamassa openSet listalta. Aina kun openSet listalla tarkistetaan solmu ja olemme lisänneet sen naapurit openSet listalle, lisäämme sen tarkasteltavana olevan solmun closedSet listalle.

Path = [ ] lista on vain lista johon tässä esimerkissä lisäämme reitin kun olemme löytäneet kaikki solmut pisteestä start pisteeseen end. (Kuva 30.)

findNeighbors() funktio merkkaa jokaiselle ruudulle sen naapurit shapes[i][j].neighbors[] listaan. Näin ei kannata tehdä tätä naapureiden tietuetta, vaan parempi tapa olisi hakea naapurit silmukan toiston yhteydessä koska jokainen naapuri on aina vain [i], [j] indeksistä seuraava tai edellinen.

Kun muuttujat on alustettu puskemme ensimmäisen solmun openSet listaan ja aloitamme itse reitin etsimisen.

Kuva 31. A-star laskennan ylimmäisen while-silmukka (Pathfinding-Visualiser, 2022).

```
while (openSet.length > 0) {
  let lowestIndex = 0;
  //find lowest index
  for (let i = 0; i < openSet.length; i++) {
    if (openSet[i].F < openSet[lowestIndex].F)
      lowestIndex = i;
    else if (openSet[i].F === openSet[lowestIndex].F) {
      if (openSet[i].H < openSet[lowestIndex].H) {
        lowestIndex = i;
      }
    }
  }
  let current = openSet[lowestIndex];  //current node
```

A-star pyörii while-silmukassa kunnes solmujen määrä openSet listassa on 0 tai kunnes seuraavassa kohdassa testaamme onko seuraavaksi tarkastettavaksi tuleva solmu loppupiste. Heti ensimmäisenä alustamme lowestIndex muuttujan numeroksi. (Kuva 31.)

Tämän jälkeen käymme läpi for-silmukassa kaikki openSet listassa olevat solmut ja tarkistamme minkä solmun.F arvo on pienin. Jos useammalla solmulla on sama F arvo tarkistamme solmun.H arvosta, mikä näisistä solmuista on lähin loppupisteeseen. Mikäli useilla solmuilla on sama .F arvo ja sama etäisyys päätepisteestä valitaan ensimmäinen solmu jonka .F arvo laitettiin pienimmäksi.

Kuva 32. A-star tarkistus onko nykyinen solmu end solmu (Pathfinding-Visualiser, 2022).

```
//if reached the end
    if (openSet[lowestIndex] === end) {
      let temp = current;
      path.push(temp);
      while (temp.cameFrom) {
        path.push(temp.cameFrom);
        temp = temp.cameFrom;
      }
      // DONE Path Found
      //draw path
      for (let i = path.length - 1; i >= 0; i--) {
        this.ctxGrid.fillStyle = "#ffff00";
        this.ctxGrid.lineWidth = this.lineWidth;
        this.drawNode(path[i].x, path[i].y, "#ffff00")
        await new Promise<void>(resolve =>
          setTimeout(() => {
            resolve();
          }, this.animDelay)
        );
      }
      break;
    }
```

Jokaisella kerralla kun otamme uuden solmun tarkistamme if lausekkeella onko edellisessä kohdassa seuraavaksi valitsemamme solmu end solmu. (Kuva 32.)

Jos end solmu on löydetty, piirrämme reitin aloitusruudusta loppupisteen ruutuun . Reitin ruutuun saamme tallentamalla ensiksi nykyisen solmun väliaikaiseen muuttujaan ja sen jälkeen puskemme muuttujan path[] listaan. Tämän jälkeen teemme while-silmukan, joka toistaa niin kauan kun löytyy solmuja joilla on solmu, josta ne ovat tulleet. While-silmukassa

tallennamme jokaisella kierroksella solmun path[] listaan ja vaihdamme temp muuttujan solmun tilalle seuraavan solmun, josta tultiin (.cameFrom solmu).

Kun while-silmukka on valmis on meillä reitti start pisteestä loppupisteeseen tallennettuna path[] listaan. Tämän jälkeen ohjelmassa piirrämme reitin gridiin for-silmukalla, joka käy path[] listan jokaisen solmun läpi.

Kuva 33. A-star kun reittiä ei löytynyt edellisen solmun pusku closedSet listaan (Pathfinding-Visualiser, 2022).

```
this.removeFromArray(openSet, current);
closedSet.push(current);
```

Aloitamme poistamalla parhaillaan tarkasteltavana olevan openSet listalta ja siirrämme sen closedSet listalle merkkaamaan, että tämä solmu on jo tarkistettu. (Kuva 33.)

Kuva 34. A-star tarkasteltavan naapurien haku (Pathfinding-Visualiser, 2022).

```
let my_neighbors = current.neighbors;
for (let i = 0; i < my_neighbors.length; i++) {
  var neighbor = my_neighbors[i];
```

Seuraavaksi tallennamme parhaillaan tarkasteltavana olevan solmun naapurit muuttujaan ja aloitamme for-silmukan jossa käymme kaikki naapuri solmut läpi yksitellen. (Kuva 34.)

Kuva 35. A-star naapureiden lisäys openSet listalle ja G arvojen päivitys (Pathfinding-Visualiser, 2022).

```
if (!closedSet.includes(neighbor) && neighbor.type != "Wall") {
  let tempG = current.G + (neighbor.x !== current.x ||
  neighbor.y !== current.y ? this.weight : this.weight * 1.414);
  let newPath = false;
  if (openSet.includes(neighbor)) {
    if (tempG < neighbor.G) {
      neighbor.G = tempG;
      newPath = true;
    }
  } else {
    neighbor.G = tempG;
    newPath = true;
    openSet.push(neighbor);
  }
```

Tarkistamme ensimmäisenä if lauseessa ettei naapuri solmu ole jo closedSet listassa ja naapuri solmun tyyppi ei ole seinä katso myös kaava 3, joka on tempG arvon loppuosa. (Kuva 35.)(Kaava 3.)

Kaava 3. Naapureiden tarkistus.

(neighbor.x !== current.x || neighbor.y !== current.y ? this.weight : this.weight * 1.414)

Kaavan 3 loppuosassa on hieman monimutkainen rivi mutta rivi suomennettuna pseudokoodilla on hieman helpompi ymmärtää. (Kaava 4.)

Kaava 4. Boolean jos testi.

muuttuja ? arvo jos tosi : arvo jos epätosi

Näillä kaavoilla 3 ja 4 meinataan, että jos gridissä kuljettaisiin viistoon olisi molemmat naapurin x ja y arvot eri kun nykyisen solmun x ja y arvot jolloin maksaisi reitin kulkeminen 1.414.

Tämän jälkeen tarkistamme if lausekkeessa onko naapuri jo openSet listassa. Jos on tarkistamme onko tähän naapuriin löytynyt lyhyempi reitti tämän uuden solmun kautta. Jos

lyhyempi reitti löytyi tähän jo aikaisemmin lisättyyn naapuri solmuun merkitsemme tämän uuden G arvon tämän naapurin G arvoksi ja laskemme sille uuden H arvon (Etäisyys loppupisteestä) ja F arvon (Yhteenlaskettu G + H). Jos naapuri oli jo openSet listassa ja uusi reitti naapuriin ei ollut lyhyempi emme laske naapurille uusia arvoja.

Kun naapuri solmu on kokonaan uusi eli sitä ei löydy openSet listasta merkitsemme naapurille G arvon ja lisäämme sen openSet listaan.

Kuva 36. A-star naapurin F ja H arvojen päivitys (Pathfinding-Visualiser, 2022).

```
          if (newPath) {
            neighbor.H = this.heuristic(neighbor, end, "Manhattan",
                         this.weight);
            neighbor.F = neighbor.G + neighbor.H;
            neighbor.cameFrom = current;
          }
        }
      }
    }
  }
```

Viimeisenä kun naapuri solmulle lasketaan H arvo (Etäisyys loppupisteestä) lasketaan tämä aikaisemmin käydyllä Manhattan tyylillä ja tämän jälkeen laskemme naapurille F arvon (Yhteenlaskettu G + H). Viimeisenä merkitsemme naapurille, mistä solmusta naapuriin tultiin. Tässä kannattaa huomioida, että jos naapuri solmu oli jo openSet listalla ja merkittiin, että sille löydettiin uusi lyhyempi reitti päivitetään sille uusi naapuri solmu merkkaamaan tätä uutta lyhyempää reittiä. (Kuva 36.)

Tämän jälkeen toistetaan tämä lopuille current solmun naapurille ja aloitamme while-silmukan alusta. While-silmukka pyörii niin kauan kunnes löytyy reitti end solmulle tai openSet listalla ei ole enään yhtään solmua, joka tarkoittaa sitä ettei reitti end solmuun ole mahdollinen.

6 Pohdinta

Lopuksi haluaisin vielä painottaa, että jokaiseen projektiin tulisi aina miettiä, mitä projekti vaatii reitinlöydöltä. Kun tämä on saatu suunniteltua, kannattaa aina tarkistaa löytyykö jo valmiina vastaavaa reitinlöytömenetelmää ilmaiseksi githubista tai maksullisena kehitysympäristön kaupasta. Jos käytää ison osan budjetista ensimmäseen prototyyppiin vain sen takia että haluaisi tehdä kaiken itse, kasvaa kokoajan riski siihen, että koko projektin budjetti tulee ylittymään huomattavasti tai vielä pahemmassa tapauksessa koko projektin joutuu keskeyttämään.

Vaikka reitinlöydössä itse reitinetsintä on painava tekijä, kehityksessä suurimmat haasteet ja työ tulee menemään toimivan gridin tai NavMeshin kehittämiseen ja optimointiin. Tämä johtuu siitä, että kaikissa projekteissa on tarvittavaa optimoida reitinlöydössä tapahtuvaa laskentaa jollain tavalla. Koska itse reitinlöytöalgoritmia ei pysty optimoimaan vaan ainoastaan vaihtamaan, ainoa mahdollisuus optimointiin on miten ja kuinka paljon tuot dataa reitinlöytöalgoritmille.

Tämän projektin Breadth-first- ja A-star -koodiesimerkeissä on käytetty aikaisempien ohjeiden mukaisesti pohjana Githubista Ishaan35 Pathfinding-Visualizer-esimerkkiä ja muokattu siitä tilanteeseen sopiva koodi (Pathfinding-Visualiser. Github, 2022). Tuota alkuperäistä github-koodia ei voi suositella kenenkään käytettäväksi missään oikeassa tilanteessa, koska se sisältää valtavia suunnitteluvirheitä, jotka ovat tuhonneet koko algoritmien idean.

Lähteet

A* Search Algorithm. 30.5.2020. geeksforgeeks. Haettu 31.5.2022.

https://www.geeksforgeeks.org/a-search-algorithm/

AStar-Typescript. (n.d.). Github. Haettu 6.2.2022.

https://github.com/digitsensitive/astar-typescript/blob/master/lib/core/heuristic.ts

Chebyshev distance. (n.d.). Wikipedia. Haettu 31.5.2022.

https://en.wikipedia.org/wiki/Chebyshev_distance

Golodetz, S. 2013. "Automatic Navigation Mesh Generation in Configuration Space". Overload. ACCU. Haettu 30.5.2022

https://accu.org/journals/overload/21/117/golodetz_1838/

Arkin, R. 1986. "Path Planning for a Vision-Based Autonomous Robot" (Technical Report). University of Massachusetts. Haettu 30.5.2022

https://www.cc.gatech.edu/ai/robot-lab/online-publications/ieee/path86.pdf

Pathfinding-Visualiser. 2.6.2020. Github. Haettu 7.3.2022.

https://github.com/Ishaan35/Pathfinding-Visualizer

Recastnavigation.Nav-mesh toolset for games. 22.2.2016. Haettu 30.5.2022

https://github.com/recastnavigation/recastnavigation

Red Blob Games. 2016. Intorduction to A*. Haettu 31.5.2022.

https://www.redblobgames.com/pathfinding/a-star/introduction.html