Flow Field -algoritmin soveltaminen Supreme Commander 2:n polunetsinnässä

Turun yliopisto
Tietotekniikan laitos
TkK-tutkielma
Tieto- ja viestintätekniikka
Huhtikuu 2024
Onni Kohtamäki

TURUN YLIOPISTO
Tietotekniikan laitos

Onni Kohtamäki: Flow Field -algoritmin soveltaminen Supreme Commander 2:n polunetsinnässä

TkK-tutkielma, 27 s.
Tieto- ja viestintätekniikka
Huhtikuu 2024

Polunetsintäalgoritmien tärkeä tehtävä on ohjata pelihahmoja pelikentällä paikasta A paikkaan B. Polunetsintäalgoritmin rakentaminen on haastavaa, koska sen tahdotaan toimivan nopeasti, tuottavan lyhyitä polkuja ja käyttävän muistia sekä laskutoimituksia mahdollisimman vähän. Reaaliaikastrategiapeleissä erityinen ongelma polunetsintäalgoritmeille on, että niiden täytyy pystyä täyttämään aiemmat kriteerit ja kyetä samanaikaisesti käsittelemään useampia eri yksiköitä eri maastotyypeissä. Erilaisia kehitettyjä polunetsintäalgoritmeja ovat leveyshakua, syvyyshaku, Dijkstran algoritmi, A*-algoritmi ja Flow Field -algoritmi.
A*- ja Flow Field -algoritmi erottuvat joukosta, koska ne kykenevät heuristiikkaan, joka antaa niille suuntavaiston polunetsinnän määränpäästä. Heuristiikan avulla polunetsintäalgoritmi käyttää vähemmän tietokoneresursseja ja samalla määrittää polun nopeammin. Näin ollen reaaliaikastrategiapeleissä yleisesti käytettyjä algoritmeja ovat A* ja Flow Field.
Tässä tutkielmassa pohditaan näiden molempien algoritmien eroja reaaliaikastrategiapelien polunetsintäalgoritmeina ja sitä, kuinka *Supreme Commander 2:ssa* sovelletaan Flow Fieldia polunetsinnän tehostamiseksi. Molemmat polunetsintäalgoritmit ovat hyvin samanlaisia, mutta ne eroavat polunetsintäpyyntöjen käsittelyssä. A* tuottaa polunetsinnässään vain yhden polun määränpäähän ja kykenee käsittelemään vain yhden polunetsintäpyynnön kerrallaan. Flow Field -algoritmi tuottaa useampia polkuja määränpäähän ja pystyy kerralla käsittelemään useampia polunetsintäpyyntöjä. Flow Fieldissa on myös hyödyllistä, että useampien polkujen avulla voidaan vähentää konfliktien syntymistä, kun monia yksiköitä liikkuu samanaikaisesti pelikentällä. *Supreme Commander 2:ssa* Flow Field -algoritmia sovelletaan A*-algoritmin ja erilaisien menetelmien, kuten sektorien, portaalien ja näkölinjan, kanssa tehokkaan polunetsintäalgoritmin saavuttamiseksi. *Supreme Commander 2:n* sovellettu polunetsintäalgoritmi etsii laadukkaampia polkuja nopeammin ja kuluttaa vähemmän muistia ja laskutoimituksia kuin pelkkä Flow Field -algoritmi.

Asiasanat: polunetsintäalgoritmi, reaaliaikastrategia, RTS, A*, Flow Field

Sisällys

Kuvat

Taulukot

1 Johdanto

Polunetsinnällä tarkoitetaan reitin etsimistä paikasta A paikkaan B. Videopeleissä polunetsinnällä haetaan reittejä, joita pitkin pelihahmot voivat liikkua pelimaailmassa. [1] Reaaliaikastrategiapeleissä (engl. Real-time strategy, RTS) polunetsintä on tärkeä osa peliä, koska pelimaailmassa on usein monia pelihahmoja, jotka voivat yksilöllisesti liikkua pelimaailmassa samaan aikaan eri paikkoihin omilla tavoillaan. [2] Polkuja etsitään RTS-peleissä polunetsintäalgoritmeilla, jotka pystyvät tutkimaan pelimaailmaa ja sen kautta löytämään reitin haluttuun kohteeseen.

Polunetsintäalgoritmeja arvioidaan niiden nopeudessa, muistin sekä laskutoimituksien kulutuksessa ja niiden kyvykkyydessä löytää lyhyitä polkuja. Yleisimpiä tutkittuja polunetsintäalgoritmeja ovat leveyshaku, syvyyshaku, Dijkstran algoritmi ja A*-algoritmi. [3] [4] Näistä kaikista algoritmeista A* on ollut suosituin RTS-peleissä, koska se pystyy säännöllisesti ja luotettavasti löytämään lyhyimmän polun haluttuun kohteeseen. [1] Toinen hieman uudempi polunetsintäalgoritmi on Flow Field -algoritmi, joka suorittaa polunetsinnän hyvin samalla tavalla kuin A*, mutta tuottaa haun kautta useampia polkuja määränpäähän.

Kuten aikaisemmin mainittiin useimmat suosituimmat RTS-pelit, kuten *Starcraft 2* [5], käyttävät A*-algoritmiin perustuvia polunetsintäalgoritmeja. Kuitenkin vuonna 2010 julkaistussa *Supreme Commander 2:ssa* pelin polunetsintää haluttiin parantaa käyttämällä Flow Field -algoritmia [6], jonka *Supreme Commander 2:sta* tekevän Gas Powered Games:n toimitusjohtaja Chris Taylor näki mullistavana uu-

distuksena polunetsinnälle [7]. Näin ollen *Supreme Commander 2* on sopiva peli tutkia sen selvittämiseksi, että, kuinka hyvin Flow Fieldiin perustuva polunetsintäalgoritmi soveltuu RTS-peleihin.

Tämän tutkielman tutkimuskysymykset ovat:

1. Kuinka A* ja Flow Field algoritmit eroavat toisistaan reaaliaikastrategiapelien polunetsintäalgoritmeina?

2. Kuinka *Supreme Commander 2:ssa* sovelletaan Flow Fieldia tehokkaan polunetsinnän saavuttamiseksi?

Tässä tutkielmassa hakulausekkeita haettiin IEEE Xplore, Web of Science ja Google Scholar tietokannoissa. Tutkielmassa käytettiin hakulausekkeita: hakulauseke 1 - *pathfinding AND (RTS OR "real-time strategy" OR "real time strategy")* ja hakulauseke 2 - *("flow field"OR "flowfield") AND (RTS OR "real-time strategy"OR "real time strategy")*. Taulukossa 1.1 esitetään hakulausekkeilla saatujen tulosten määrä. Osa hakulausekkeiden tuloksista löytyi useammasta tietokannasta, ja näin ollen osa tutkielmassa käytetyistä tuloksista on merkitty useampaa kertaa eri tietokantojen kohdalle.

Taulukko 1.1: Hakulausekkeiden tulokset

	Hakulauseke 1		**Hakulauseke 2**	
Tietokanta	**Tulokset**	**Aineistoon**	**Tulokset**	**Aineistoon**
IEEE	303	5	1	0
Web of Science	11	6	2	0
SpringerLink	1160	3	402	0
Google Scholar	2970	15	2990	3

Eräs ongelma lähteiden haussa oli, että tieteellisiä tutkimuslähteitä Flow Fieldin käytöstä RTS-peleissä löytyi suppeasti. Tämä voi johtua siitä, että Flow Field on vielä suhteellisen uusi polunetsintäalgoritmi RTS-peleissä, minkä vuoksi tutki-

mukset siitä ovat vähäisiä. Tästä syystä tähän tutkielmaan on otettu ei-tieteellisiä raportteja ja artikkeleita pelikehittäjiltä, jotka ovat työskennelleet Flow Field -polunetsintäalgoritmien kanssa. Hakuterminä näitä lähteitä etsiessä oli *Pathfinding*.

Tutkielman seuraavassa luvussa käydään läpi, mitä reaaliaikastrategiapelit ja polunetsintäalgoritmit tarkoittavat. Luvussa tarkastellaan eri polunetsintäalgoritmeja, jotka eivät perustu heuristiikkaan ja selitetään, mikä niissä on ominaista polunetsintäalgoritmeina. Luvussa 3 tarkastellaan A*- ja Flow Field -algoritmien soveltuvuutta RTS-peleissä ja *Supreme Commander 2:n* Flow Fieldiin perustuvan polunetsintäalgoritmin tehokkuutta. Luvun lopussa esitetään vastaukset tutkimuskysymyksiin. Luvussa 4 käydään läpi kaikki oleellisimmat asiat, jotka aikaisemmissa luvuissa on käsitelty ja pohditaan RTS-pelien polunetsintäalgoritmien tulevaisuuden tutkimuksen kohdistamista.

2 Tausta

2.1 Reaaliaikastrategiapeli

Reaaliaikastrategiapelit ovat pelejä, joissa pelaajat hallinnoivat erilaisia rakennuksia ja yksiköitä ja tekevät näitä käyttäen strategisia päätöksiä, jotka johtavat pelin voittamiseen. Tyypillisessä RTS-pelissä pelaajan tehtävänä on kerätä resursseja, rakentaa rakennuksia ja koota sekä ohjata yksiköitä. [8] Pelaajan tavoite RTS-pelissä on päihittää vastustaja, joka taas vastaavasti pelaajan tavoin pyrkii omilla resursseillaan ja joukoillaan kukistamaan pelaajan. RTS-peleissä ominaista on ajan kuluessa tapahtuva pelin laajentuminen, joka johtaa pelaajan vastuuksien ja pelin haasteellisuuden kasvuun. Usein pelin alussa strateginen päätöksenteko on helppoa ja se keskittyy ainoastaan resurssien kumuloimiseen. Resurssien avulla pelaaja kykenee rakentamaan rakennuksia ja joukkoja, joilla pelaaja voi taistella vastustajaa vastaan. Pelin edistyminen johtaa uusien pelimekaniikkojen avautumiseen, ja pelaajan haasteeksi tulee tasapainoittelu näiden eri mekaniikkojen välillä. Vaikka pelaaja saisi armeijan koottua, ei se tarkoita, että hänen tulisi lopettaa resurssien kerääminen, sillä pelissä on aina vaara, että vastustaja voi tuhota armeijan, jolloin sellaiseen tilanteeseen varautuminen on kannattavaa ja pelaajalla tulisi olla resursseja säilössä armeijan uudelleen rakentamiseen. [9]

Kuitenkaan RTS-pelit eivät ole pelkästään strategista ajattelua vaan ne vaativat myös nopeaa ajattelua ja päätöksentekoa. RTS-peleissä aika kulkee reaaliaikaisesti,

jolloin pelikentällä tapahtuvat muutokset tarvitsevat pelaajalta toimintaa välittömästi. [10] Näin ollen pelaa täytyy miettiä ja toteuttaa päätös nopeasti tilanteen tapahtumisen jälkeen. Reaaliaikaisuus erottaa RTS-pelit monista muista tavallisemmista strategiapeleistä, joissa aika liikkuu vain rajoitteellisesti, kuten vuorottaisesti. Tällaisissa peleissä ajan seisahtamisen tarkoituksena on painottaa strategista ajattelua ja päätöksien suunnittelua. RTS-peleissä pelaaja asetetaan jatkuvan jännityksen ja valppauden tilaan, mikä tuo pelaajille aivan omanlaisia haasteita, joiden ratkomiseen taas tarvitaan tehokasta strategista suunnittelua sekä nopeata päättelykykyä. [11] Kuvassa 2.1 esitetään näkymä Blizzard Entertainmentin reaaliaikastrategiapelistä Starcraft 2.

Kuva 2.1: Pelaajan näkymä Starcraft 2:ssa.

Kuva 2.1 on otettu pelin kampanjan tilanteesta, jossa vihollisen yksiköt hyökkäävät pelaajan tukikohtaan. Kuvassa näkyy vihollisen ja pelaajan yksiköiden taistelu, pelaajan resurssien kerääjät louhimassa mineraaleja ja eri pelaajan tukikohtaan kuuluvat rakennukset. Tukikohta hyökkäykset ovat hyvin tyyppillisiä tapahtumia Starc-

raft 2, koska niillä voidaan saada aikaan suurta vahinkoa vastustajaan. Vuonna 2010 julkaistu Starcraft 2 on edelleen yksi suosituimmista RTS-peleistä.

2.2 Polunetsintäalgoritmit ilman heuristiikka

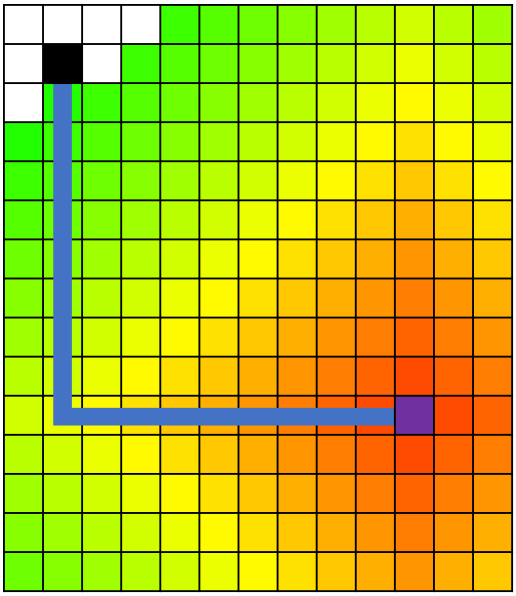
Joukkojen liikuttaminen pelimaailmassa on yksi tärkeimmistä toiminnoista RTS-peleissä [12]. Resurssien kerääminen, tiedustelu ja vihollista vastaan hyökkääminen on mahdollista vain, jos pelaajalla on kyky liikuttaa joukkojaan ympäri pelimaailma. Siksi RTS-peliin pitää rakentaa järjestelmä, jolla pelaaja voi antaa liikkumiskomentoja yksiköilleen, jotka sitten komentoa noudattaen kulkevat tiettyä reittiä komennon määrittämään kohteeseen. RTS-peleissä tämä järjestelmä toteutetaan polunetsintäalgoritmeilla, jotka pystyvät tutkimaan pelikenttää ja etsimään reitin haluttuun määränpäähän. Kuitenkin useimmiten mikä tahansa reitti ei kelpaa vaan polunetsinnällä halutaan löytää kaikista lyhyin tai nopein reitti. [13]

Jotta polunetsintäalgoritmia voidaan hyödyntää, tarvitaan jonkinlainen tietorakenne, joka jakaa pelikentän useampaan pienempään osaan ja esittää nämä osat solmuina. Polunetsintäalgoritmi kykenee tutkimaan näitä solmuja ja arvioimaan mitä solmuja pitkin reitti kannattaa muodostaa. Usein RTS-pelien pelikenttä esitetään ruudukkona, jossa jokainen ruutu on solmu, joka esittää tiettyä osaa pelikenttää. Tällaisessa tietorakenteessa polunetsintäalgoritmi kykenee tutkimaan pelikenttää ja löytämään reitin osoittamalla niihin ruutuihin, joiden kautta pitää kulkea. [13]

Ruudukon tarkoituksena on usein erottaa kulkukelpoiset osat pelikentällä. Eri tekijät pelikentällä, kuten maasto, rakennukset tai yksiköt, voivat rajoittaa kulkukelpoisuutta, ja polunetsintäalgoritmi tarvitsee tiedon näiden tekijöiden olemassa olosta. Ruudukon avulla pelikenttä voidaan erottaa niihin ruutuihin, joissa pelikenttä on kulkukelpoista, ja ruutuihin, joissa kulkeminen on estettyä. Tutkiessaan ruudukkoa polunetsintäalgoritmi lukee esteitä sisältävät ruudut ja välttää reitin rakentamista näiden ruutujen kautta. [14]

2.2.1 Leveyshaku

Esteettömässä ruudukossa eli ruudukossa, jossa ei ole ruutuja, joiden läpi ei voi kulkea, polunetsintäalgoritmin täytyy yksinkertaisesti etsiä suorin reitti määränpäähän. Polunetsinnän ensimmäinen ongelma syntyy tässä vaiheessa, kun ei tiedetä, missä määränpää on ja mistä aloittaa polun etsiminen. Helppo ratkaisu tähän voisi olla leveyshaku, joka hakee määränpäätä aaltoa muistuttavalla tavalla. Leveyshaku tarkastaa ensin aloituspaikan jokaisen naapuriruudun, minkä jälkeen se tarkastaa niiden jokaisen naapuriruudun naapuriruudun. Leveyshaku tarkastaa järjestyksessä jokaisen naapurin naapurin, kunnes jonkin naapuriruudun naapuriruutu on määränpää, jolloin leveyshaku luo lyhyimmän polun tarkasteluista ruuduista määränpäähän. [15] Kuvassa 2.2 esitetään, kuinka leveyshaku etsii polun kentältä.



Kuva 2.2: Leveyshaku. Violetti ruutu osoittaa aloituspaikkaa ja musta määränpäätä. Muut värilliset ruudut osoittavat tarkasteltuja ruutuja järjestyksessä punaisesta vihreään. Sininen viiva merkitsee algoritmin rakentamaa polkua.

Kuvassa 2.2 leveyshaku tarkastaa ruutuja myötäpäiväisessä järjestyksessä länsi, pohjoinen, itä ja etelä. Leveyshaku tarkastaa ensin aloitusruudun naapuriruudut, minkä jälkeen aloitusruudun naapuriruutujen naapuriruudut ja seuraavat naapuriruudut. Leveyshaku tarkastaa ensin jokaisen suunnan lähimmäiset ruudut varmis-

taakseen, että lyhyin reitti määränpäähän löydetään taatusti. Kun algoritmin tarkastelussa olevan ruudun naapuriruutu on määränpää, leveyshaku lopettaa haun. Leveyshaku määrittää lyhyimmän polun tarkasteluja ruutuja pitkin.

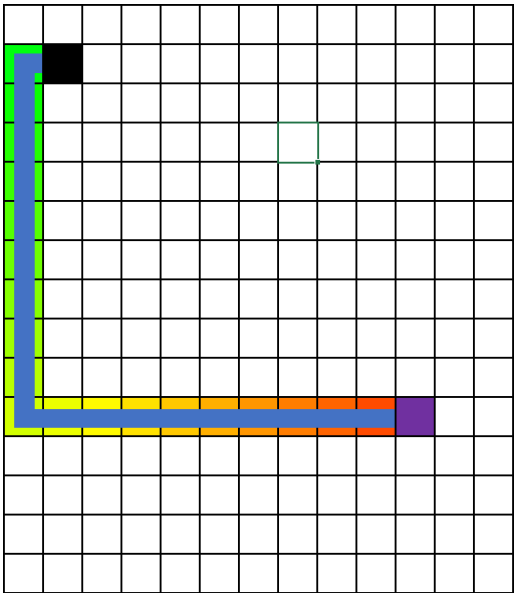
Leveyshaku on todella luotettava polunetsintäalgoritmi, koska se varmasti löytää aina lyhyimmän polun määränpäähän. [2] Leveyshaussa on kuitenkin ongelmia, sillä siitä tulee hitaampi, mitä kauempana määränpää on aloituspaikasta. Tämä johtuu siitä, että se tarkastaa jokaiseen vierekkäisen ruudun ennen kuin se saapuu määränpäähän. [3] Näin ollen leveyshaku tuhlaa resursseja tarpeettomien ruutujen tarkasteluun, jolloin kaukaisiin määränpäihin polunetsiminen on huomattavasti kalliimpaa kuin lähellä oleviin.

2.2.2 Syvyyshaku

Syvyyshaussa polku haetaan valitsemalla jokin aloitusruudun viereinen ruutu ja tarkastelemalla siitä lähtien aina viereistä ruutua, joka on kaikista kaukaisimpana aloitusruudusta. Kun viereinen ruutu on määränpääruutu, syvyyshaku lopettaa tarkastelun ja määrittää polun tarkasteltuja ruutuja pitkin. [16] Kuvassa 2.3 esitetään, kuinka syvyyshaku etsii polun kentältä.

Kuvassa 2.3 syvyyshaku tarkastaa ruutuja järjestyksessä länsi, pohjoinen, itä ja etelä. Algoritmi tarkastaa aina lännessä olevan naapuriruudun, kunnes pelikentän reuna tulee vastaan. Syvyyshaulla ei tässä vaiheessa ole enää läntistä naapuriruutua, jolloin alkaa tarkastaa seuraavaa ilmansuuntaa pohjoista. Syvyyshaku tarkastaa pohjoisia naapuriruutuja, kunnes määränpääruutu on tarkastelussa olevan ruudun naapuriruutu, jolloin tarkastelu päättyy. Nyt algoritmi tietää reitin aloituspaikasta määränpäähän, ja se määrittää polun tarkasteltuja ruutuja pitkin.

Syvyyshaun ongelma on sen liiallinen epäsäännöllisyys. Syvyyshaussa esiintyy usein tilanteita, joissa määränpää löydetään todella hitaasti tai nopeasti. Tämä johtuu siitä, että syvyyshaun nopeus riippuu täysin määränpään sijainnista ja ensim-



Kuva 2.3: Violetti ruutu osoittaa aloituspaikkaa ja musta määränpäätä. Muut värilliset ruudut osoittavat tarkasteltuja ruutuja järjestyksessä punaisesta vihreään. Sininen viiva merkitsee algoritmin rakentamaa polkua.

mäiseksi tarkasteltusta ruudusta. Määränpään löytäminen syvyyshaussa on aina satunnaista, ja niinpä syvyyshaku ei pysty takaamaan, että polku määritetään tehokkaasti. Toinen ongelma syvyyshaussa on, että se ei löydä aina kaikista lyhyintä reittiä. Leveyshaku löytää aina kaikista lyhyimmän reitin, koska se tarkastaa kaikki lyhyimpään reittiin sisältyvät ruudut. Näin ei tapahdu syvyyshaussa vaan se hakee ruutuja vain niin kauan, kunnes määränpää on löydetty. Ruudut, joiden kautta lyhyin reitti kulkee, saattavat jäädä tarkistamatta, jolloin syvyyshaku määrittää huonomman polun kuin olisi mahdollista. Täten syvyyshaku ei pysty takaamaan löytämäänsä reittiä aina kaikista lyhyimmäksi. [17]

2.2.3 Dijkstran algoritmi

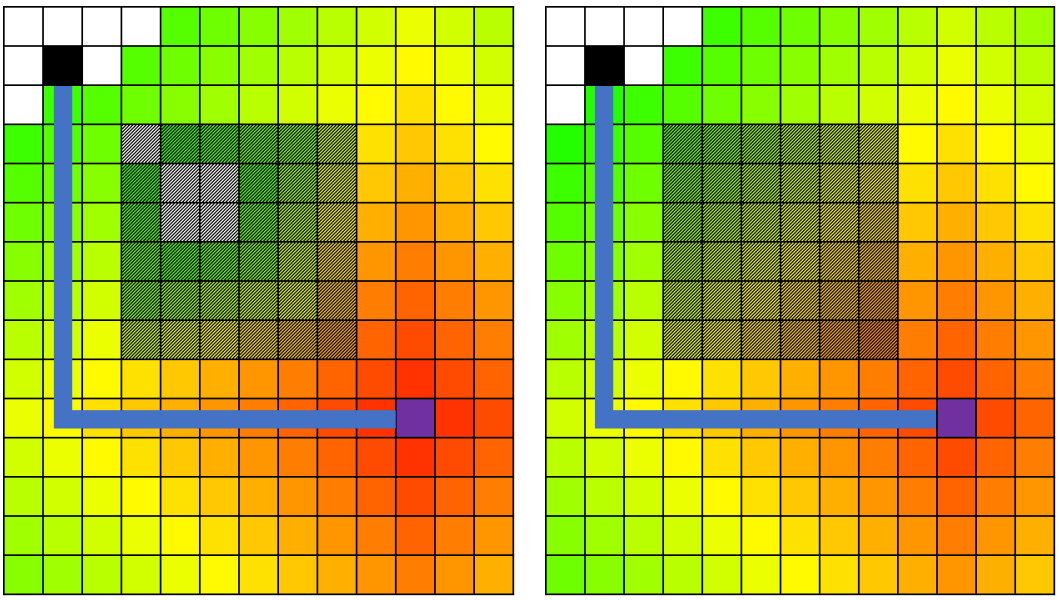
Leveys- ja syvyyshaku eivät kuitenkaan ole kovin optimaalisia videopeleihin. Leveyshaun käyttäminen vaatii paljon laskutoimituksia, ja syvyyshaku on liian epäluotettava löytämään määränpään tarpeeksi säännöllisessä ajassa. Myös syvyyshaun

löytämät reitit eivät aina ole kaikista lyhyimpiä. Näistä syistä leveys- ja syvyyshaku eivät ole kovin suosittuja RTS-pelien polunetsintäalgoritmeina.

Eräs hieman kehittyneempi polunetsintäalgoritmi on Dijkstran algoritmi [18], joka etsii lyhyimmän reitin ottamalla huomioon haussa ruutuun siirtymisen kustannuksen. Dijkstran algoritmi muistuttaa paljon leveyshakua, sillä se tarkastelee järjestyksessä kaikki vierekkäiset ruudut aloitusruutuun ennen kuin se siirtyy seuraavaksi vierekkäisimpiin. Kuitenkin Dijkstran algoritmi eroaa leveyshaussa siinä, että Dijkstran algoritmi tarkastelee aina sen ruudun, johon on kustannukseltaan lyhyin matka aloitusruudusta. Tätä varten pelikentän perusteella rakennetaan kustannuskenttä, joka esittää jokaisen ruudun kohdalla siihen siirtymisen kannattavuutta eli kustannusta. Nämä ruutujen kustannusarvot esitetään kokonaislukuina, ja tietyn polun kustannusta voidaan arvioida yhteenlaskemalla kaikkien kuljettujen ruutujen kustannusarvot. Dijkstran algoritmissa hyödynnetään kustannuskenttää lyhyimmän reitin selvittämiseksi. [17] [18]

Dijkstra käyttää kustannusfunktiota arvioimaan kaikkien mahdollisten polkujen kustannusta. Dijkstran algoritmi tutkii aina sen ruudun vierekkäiset ruudut, johon on kustannukselta pienin polku. Kun ruudun kaikki vierekkäiset ruudut on tarkasteltu, Dijkstran algoritmi siirtyy uuteen ruutuun, johon on kustannusfunktion perusteella lyhyin matka aloituspaikasta. Dijkstran algoritmi toistaa tätä, kunnes vierekkäinen ruutu on määränpää. [18] Kuvassa 2.4 esitetään, kuinka Dijkstran algoritmi ja leveyshaku etsivät polun kentältä, jossa kaikki ruudut eivät ole kustannukseltaan saman arvoisia.

Kuvassa 2.4 molemmat algoritmit tarkistavat ruutuja järjestyksessä läntinen, pohjoinen, itäinen ja eteläinen ruutu. Dijkstran algoritmi käyttää lisäksi ruutujen kustannusarvoja päätelläkseen, mitä ruutuja kannattaa tarkastella lyhyimmän polun löytämiseksi. Tästä syystä Dijkstran algoritmi pyrkii välttämään korkean kustannusarvon ruutujen tarkastelua. Näin ollen leveyshaku tarkistaa viisi ruutua enem-



(a) Dijkstran algoritmi. (b) Leveyshaku.

Kuva 2.4: Violetti ruutu osoittaa aloituspaikkaa ja musta määränpäätä. Muut värilliset ruudut osoittavat tarkasteltuja ruutuja järjestyksessä punaisesta vihreään. Sininen viiva merkitsee algoritmin rakentamaa polkua. Molempien kenttien keskellä on tummennettu alue, jossa siirtyminen on kustannukselta kalliimpaa. Tavallinen ruutu on kustannukselta 1 ja raidoitettu ruutu 3.

män kuin Dijkstran algoritmi. Koska näiden viiden ruudun kustannus on kohtuullisen suuri ja niiden läpi ei kulje paras mahdollinen polku määränpäähän, ei näiden viiden ruutujen tarkistaminen Dijkstran algoritmin mukaan kannattavaa. Kun pelikentälle halutaan antaa kustannukseltaan eri arvoisia ruutuja, Dijkstran algoritmi pystyy välttämään kustannukseltaan korkeita polkuja ja samalla tarkastella vähemmän ruutuja, mikä taas säästää tarvittavien laskutoimituksien määrää ja muistia.

Mikäli kaikilla ruuduilla on sama kustannusarvo, Dijkstran algoritmi toimii aivan samalla tavalla kuin leveyshaku. Kuitenkin, jos ruuduille annetaan eri kustannusarvoja, huomataan, että Dijkstran algoritmi suosii polkuina ruutuja, joihin siirtymisen kustannus on pienin. RTS-peleissä ruutujen kustannusta voidaan käyttää esittämään pelin eri maastotyyppejä. Esimerkiksi maasto voi olla suo, jolloin tyypillisesti siinä liikkuminen kustannukseltaan korkeaa ja näin ollen hidasta. Luotaessa eri maastotyyppejä RTS-pelin kentälle, voi olla, että lyhyin reitti määränpäähän on kulkea

kustannukseltaan kalleinta polkua. Tästä syystä polku ei kuitenkaan ole kaikista paras polku, ja Dijkstran algoritmi osaa löytää muita lyhyitä polkuja, jotka kulkevat helpommin kuljettavaa maastoa pitkin nopeammin määränpäähän. Näin ollen todellinen lyhyin polku on Dijkstran löytämä polku. [1]

3 Flow Field ja A*-algoritmi reaaliaikastrategiapeleissä

3.1 A*:in ja Flow Fieldin soveltaminen reaaliaikastrategiapelissä

A*-algoritmi on pitkään ollut vanhempien RTS-pelien suosima polunetsintäalgoritmi. [14] [1] Kuitenkin Flow Field -algoritmi, joka on nykyaikaisempi polunetsintäalgoritmi, on tullut käyttöön uudemmissa RTS-peleissä *Supreme Commander 2:ssa* ja *Age of Empires IV:ssa* [19]. Syy tälle on se, että Flow Field kykenee älykkäämpään sekä tehokkaampaan polunetsintää kuin vanhempi A*-algoritmi. Tässä luvussa käydään syitä sille, miksi RTS-pelit, kuten *Supreme Commander 2* ja *Age of Empires IV* siirtyivät käyttämään Flow Fieldia A*-algoritmin sijaan.

3.1.1 A*-algoritmi

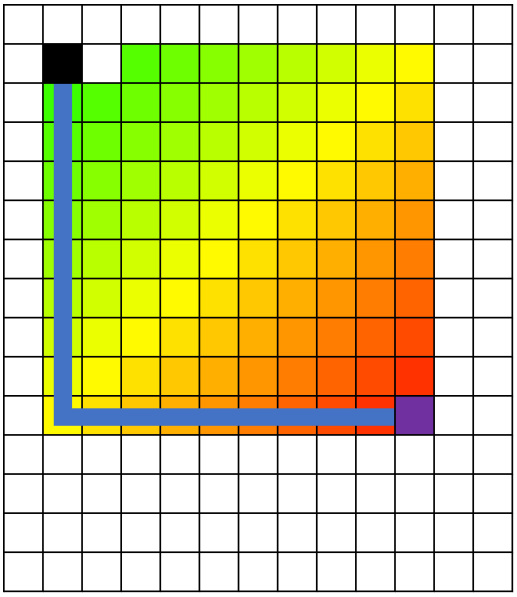
Dijkstran algoritmi kärsii samoin kuin leveyshaku turhan suuresta resurssien käytöstä, koska se joutuu polunetsinnässään tarkastelemaan liian monta ruutua ennen kuin se löytää määränpään. [2] Siltä puuttuu eräänlainen suuntavaisto, joka ohjaisi sitä tarkastelemaan vain niitä ruutuja, jotka olisivat oleellisia lopullisessa polussa. A*-algoritmi on yksi luotettavimmista polunetsintäalgoritmeista, koska se kykenee

aina löytämään parhaan mahdollisen reitin suhteellisen säännöllisellä tehokkuudella. A*-algoritmi perustuu heuristiikkaan, joka ohjaa polunetsintää lähemmäksi maalia.

A*-algoritmi toimii hyvin samantapaisesti kuin Dijkstran algoritmi siinä suhteen, että algoritmilla on käytössä kustannusfunktio, joka määrittelee seuraavan tarkasteltavan ruudun, johon päästään lyhyintä reittiä aloituspaikasta. Tämän lisäksi A*-algoritmissa otetaan huomioon myös matka ruudusta määränpäähän käyttämällä heuristista funktiota. Tätä varten täytyy luoda kustannuskenttä ja heuristisen arvojen kenttä eli integraatiokenttä, joka asettaa jokaisen ruudulle oman heuristisen arvon. [6] Heuristinen arvo saadaan laskemalla kustannuskentän perusteella kyseisen ruudun ja määränpään välisen polun kustannus. Kun kaikilla integraatiokentän ruuduilla on heuristiset arvot, A*-algoritmi kykenee vertailemaan ruutujen etäisyyttä maalista. [20]

A*-algoritmin haku tuotetaan samalla tavalla kuin Dijkstran algoritmissa, mutta A* käyttää omaa kustannusfunktiota ruutuun siirtymisen kannattavuuden arvioimisessa. A*:in kustannusfunktio tuottaa tuloksen laskemalla heuristisen funktion sekä Dijkstran algoritmin kustannusfunktion tulokset yhteen. A* aloittaa haun tutkimalla ensin aloitusruudun naapuriruudut, minkä jälkeen se siirtyy seuraavan tarkasteltavaan ruutuun, jonka kustannusfunktion toteama tulos pienin. A*-algoritmi tutkii uuden ruudun naapurit ja kustannusfunktion perusteella valitsee seuraavan ruudun. A* toistaa tätä, kunnes se löytää määränpään, ja määrittää lyhyimmän polun aloituspaikasta määränpäähän. [20] Kuvassa 3.1 esitetään, kuinka A*-algoritmi etsii polun kentältä.

Kuvassa 3.1 A*-algoritmi tarkastaa ruutuja myötäpäiväisessä järjestyksessä länsi, pohjoinen, itä ja etelä. Kuitenkin A* tarkastelee kuvassa vain läntisiä ja pohjoisia ruutuja, koska näiden ruutujen heuristiset arvot ovat pienempiä kuin itäisten ja eteläisten. Heuristiset arvojen avulla A*-algoritmi kykenee vaistomaan suunnan, jossa määränpää on. Kaikkien ruutujen kustannus on sama, ja näin ollen A* tarkastaa



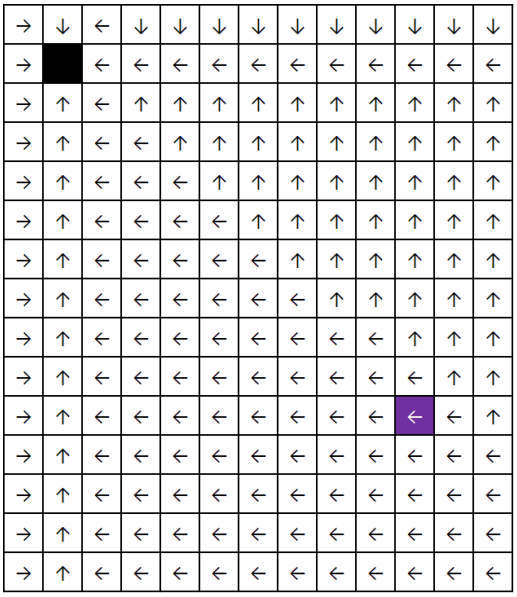
Kuva 3.1: Violetti ruutu osoittaa aloituspaikkaa ja musta määränpäätä. Muut värilliset ruudut osoittavat tarkasteltuja ruutuja järjestyksessä punaisesta vihreään. Sininen viiva merkitsee algoritmin rakentamaa polkua.

ruudut järjestyksessä aloitusruudusta kaikki läheisimmät ruudut ennen kuin siirtyy seuraavaksi läheisempiin. Tällä tavalla A* voi olla varma, että se löytää lyhyimmän reitin määränpäähän. Kun A* tarkastelee ruutua, jonka naapuriruutu on määränpää, se päättää haun ja määrittää lyhyimmän polun aloituspaikasta määränpäähän. Heuristiikan avulla A*-algoritmi löytää polun tarkastelemalla vähempää määrää ruutuja ja samalla säästää laskutoimituksia ja muistia. Tämä ja lisäksi, että se löytää lyhyimmän polun, on tehnyt A*-algoritmista suosituin polunetsintäalgoritmin RTS-peleissä. [1]

3.1.2 Flow Field -algoritmi

Samalla tavalla kuin A*-algoritmissa Flow Fieldissa jokaisella ruudulla on oma kustannusarvonsa ja heuristinen arvo. Tätä varten Flow Fieldille täytyy luoda kustannuskenttä ja integraatiokenttä. Flow Field -algoritmi kuitenkin eroaa A*-algoritmin haussa siinä, että Flow Field tarkastelee jokaisen pelikentän esteettömän ruudun. Alkaen aloituspaikasta leveyshaun tapaisesti Flow Field -algoritmi tarkastelee jokaisen

naapuriruudun ja niiden naapuriruudut. Tarkastellessa ruutua Flow Field -algoritmi selvittää jokaisen naapuriruudun kustannusarvon ja heuristisen arvon ja laskee nämä arvot yhteen. Algoritmi määrittää naapuriruudun, jonka yhteenlaskun tulos on pienen, ja asettaa tarkasteltuun ruutuun vektorinuolen, joka osoittaa tähän naapuriruutuun. Kun Flow Field tarkastellut jokaisen esteettömän ruudun, haku päättyy, ja Flow Field -algoritmi luo uuden Flow Field -kentän, jossa jokaisessa esteettömässä ruudussa on vektorinuoli, joka osoittaa seuraavaan esteettömään ruutuun. Vektorinuolista muodostuu virtaus (engl. flow), joka mahdollistaa niitä seuraamalla suunnistamisen mistä tahansa esteettömästä ruudusta määränpäähän. [14] [6] Kuvassa 3.2 esitetään, kuinka Flow Field on määrittänyt virtauksen kentälle.



Kuva 3.2: Violetti ruutu osoittaa aloituspaikkaa ja musta määränpäätä. Nuolet ovat Flow Fieldin vektorinuolia, jotka kokonaisuudessa muodostavat Flow Fieldin virtauksen.

3.1.3 Suurien yksikkömäärien polunetsintä

Flow Fieldin polunetsintä eroaa A*-algoritmissa siinä, että se pystyy samanaikaisesti laskemaan useamman polun tiettyyn määränpäähän monista eri lähtöpaikoista. Tämä hyödyllinen ominaisuus RTS-peleissä, sillä useammissa tilanteissa pelissä tarkoi-

tuksena on siirrellä useampia yksiköitä kerralla. Joissakin RTS-peleissä kentällä voi olla jopa tuhansia yksiköitä, jotka vaativat polunetsintää samanaikaisesti, mikä liian raskasta A*-algoritmin suoritettavaksi. [14] Flow Field -virtaus mahdollistaa sen, että kaikki yksiköt saavat saman polunetsintätuloksen, mikä on päinvastoin A*:n polunetsintään, jossa jokainen yksikkö hakee oman polun määränpäähän. Flow Fieldin tavalla suurille määrille yksiköitä voidaan yhdellä polunetsintäkerralla löytää polut määränpäähän, kun taas A* joutuu suorittaa useampia yksittäisiä polunetsintöjä, mikä hidastaa algoritmin toimintaa.

Tutkimuksessa [21] vertailtiin A*:n ja Flow Fieldin nopeutta johdattaa yksiköitä määränpäähän. Tutkimusta varten tutkijat loivat simulaation torninpuolustuspelistä. Torninpuolustuspelit ovat reaaliaikastrategiapelejä, joissa tarkoituksena erilaisia torneja tai yksiköitä käyttäen estää pelikentällä liikkuvia vihollisia pääsemästä tiettyyn annettuun määränpäähän. Tutkimus suoritettiin kolmella erilaisella pelikentällä, joista jokaisesta otettiin kymmenen otosta kummallekkin polunetsintäalgoritmille. Jokaisessa otoksessa polut etsittiin kymmenelle yksikölle. Tehokkaampi polunetsintäalgoritmi määriteltiin ottamalla aikaa kummankin algoritmin kohdalla ja katsomalla, kuinka nopeasti yksiköt pääsivät maaliin algoritmia käyttäen. Algoritmien otosten ajoista laskettiin keskiarvo jokaisen pelikentän kohdalla, ja tehokkaampaa algoritmia arvioitiin vertailemalla näitä keskiarvoja. Taulukko 3.1 esittää tutkimuksessa A*- ja Flow Field -algoritmeilla saadut keskiarvoiset aikatulokset.

Taulukko 3.1: Tutkimuksen [21] tulokset.

	Flow Field	A*
Pelikenttä 1	3.1527 s	3.7498 s
Pelikenttä 2	3.5822 s	4.4140 s
Pelikenttä 3	14.4671 s	15.025 s

Tutkimuksen lopputuloksena havaittiin, että Flow Field -algoritmi löytää nopeammin polun simuloidussa torninpuolustuspelissä. Jokaisessa kolmessa testatussa pelikentässä Flow Fieldilla yksiköt saapuivat määränpäähän keskiarvoisesti nopeam-

min kuin A*:lla. Näiden testien perusteella RTS-peleissä Flow Field on nopeampi polunetsintäalgoritmi kuin A*.

3.1.4 Samanaikaisesti liikkuvat yksiköt

Ongelmia ilmenee lisää, kun A*-algoritmilla etsitään polkuja samanaikaisesti useammalle yksikölle. [17] *Supreme Commanderissa* polunetsintä oli toteutettu A*-algoritmilla, mikä oli aiheuttanut ongelmia, kun yksiköt kohtasivat toisensa pelikentällä. Pelissä toisiinsa törmäävät yksiköt pysähtyivät ennen törmäystä ja jäivät odottamaan, kunnes yksikköjen tiellä ei ollut mitään. Ratkaisuna olisi voinut olla, että yksiköt etsisivät uudet polut, mutta tämä olisi johtanut suurempaan ongelmaan, jossa uusi polku johtaisi toiseen törmäystilanteeseen ja yksikkö joutuisi uudestaan laskemaan polun, joka johtaisi kohti uutta törmäystä. Näin ollen ongelma jätettiin korjaamatta, mikä tarkoitti sitä, että pelaajan käskyttämät joukot eivät saapuneet määränpäähän. Pelaajan ärsytykseksi hänen täytyi antaa uusia liikkutamiskomentoja joukoilleen vähään väliä matkan varrella. [6]

Flow Fieldin polunetsinnässä näitä mahdollisia törmäystilanteita voidaan vähentää virtauksen avulla. Koska Flow Field tuottaa useamman polun määränpäähän, useimmissa tilanteissa yksikkö pystyy välittömästä siirtyä seuraamaan uutta polkua, mikäli se eksyy vanhalta polultaan. Fysiikan lisääminen peliin mahdollistaa yksiköitä siirtämään toisia yksiköitään tieltään, jolloin törmäystilanteet voidaan ratkaisemaan niiden syntyessä. [6]

3.2 Flow Field -algoritmin soveltaminen *Supreme Commander 2:ssa*

Flow Fieldin lisäksi polunetsinnässä voidaan hyödyntää erilaisia menetelmiä, joita voidaan käyttää tehokkaan polunetsinnän saavuttamiseksi. Menetelmistä kertoo

artikkelissaan [6] Elijah Emerson, joka oli mukana *Supreme Commander 2:n* kehityksessä. *Supreme Commander 2:ssa* ratkaisuna aikaisemman pelin polunetsintä ongelmiin oli käyttää Flow Field -algoritmia pääsääntöisenä polunetsintäalgoritmina A*-algoritmin sijaan. *Supreme Commander 2:ssa* käytettyjen menetelmien tarkoituksena on polunetsintään menevän ajan sekä muistin ja laskutoimituksien säästäminen. Flow Fieldin käyttäminen yksinään ilman näitä menetelmiä on tietokoneelle hyvin raskasta ja epätehokasta, minkä vuoksi niiden toteuttaminen sen rinnalla on erittäin tärkeää. Flow Field -algoritmin ja näiden menetelmien käyttö polunetsinnässä mahdollistaa suurempien pelikenttien ja yksikkömäärien käytön ilman raskaita kuluja tietokoneen resursseille.

3.2.1 Sektorit

Flow Field -algoritmi muodostaa polun jokaisesta ruudusta määränpäähän, vaikka suurinta osaa poluista ei koskaan käytettäisi. Tämä tarkoittaa sitä, että nämä turhat polut nielevät tietokoneresursseja tarpeettomasti ja vievät aikaa muiden tarpeellisten polkujen muodostumiselta. Tämä ongelma kasvaa suuremmaksi, kun pelikentän koko kasvaa. Pelikentän jakaminen sektoreihin vähentää Flow Fieldin muodostamien polkujen määrä. Sektoreilla voidaan kohdistaa Flow Fieldin algoritmin polunetsintä vain niihin sektoreihin, joiden läpi yksiköt aikovat kulkea pelikentällä. Tällöin polunetsintä voidaan jättää pois tarpeettomilta sektoreilta, jolloin säästetään aikaa sekä tietokoneresursseja. [6]

3.2.2 Portaaligraafi

Sektorien lisäksi tarvitaan portaaligraafi, jotta sektoreita voidaan todellisuudessa hyödyntää. Sektorit ovat itsessään hyödyttömiä, mikäli ei pystytä määrittämään, mitä sektoreita polunetsinnässä tullaan tarvitsemaan. Portaaligraafi on verkosto portaaleja, jotka ovat sijoitettu sektorien naapurisektorien väliin, ja niiden läpi on

mahdollista siirtyä sektorista portaalin vastapuoliseen sektoriin. Portaaleita voidaan rakentaa vain tyhjien ruutujen väliin, eli niitä ei pysty rakentaa sellaisten ruutujen väliin, joiden välinen siirtyminen ei ole mahdollista esimerkiksi esteen vuoksi. Portaalien tarkoituksena on pystyä navigoimaan niitä pitkin sektorista toiseen ilman, että tarvitsee käyttää pelikentän ruudukkoa siirtymiseen. Tällä tavalla voidaan paljon nopeammin selvittää, mistä sektoreista halutaan siirtyä mihin sektoreihin. Tämä hyödyllistä, mikäli halutaan nopeasti määrittää tarvittavat sektorit Flow Field -algoritmille. [6]

3.2.3 Näkölinja

Flow Field -algoritmin polunetsintää voidaan tehostaa näkölinjan avulla lisää. Näkölinjan (engl. line of sight, LOS) avulla yksiköille voidaan antaa suorin reitti määränpäähän ja samalla voidaan kitkeä ruutujen määrä, joissa Flow Field -algoritmi tarvitsee suorittaa. Näkölinja toimii siten, että jokaiselle ruudulle, josta voidaan nähdä suora linja määränpäähän, asetetaan näkölinjan omaaviksi. Kun yksikkö siirtyy ruutuun, josta on näkölinja, yksikkö lopettaa Flow Field -virtauksen seuraamisen, ottaa näkölinjan määränpäähään ja lähtee liikkumaan suorinta reittiä sitä kohti. Näkölinjan käyttäminen kannattaa, koska näkölinjojen asettaminen on suhteellisen halpa operaatio tietokoneen resursseja kohden ja se tuottaa parhaimman mahdollisen reitin tilanteessa. [6]

3.2.4 Polunetsintäpyyntö

Polunetsintäprosessi alkaa pelaajan liikkumiskomennon antamisesta, joka johtaa polunetsintäpyynnön luomiseen. Tämä polunetsintäpyynnön tarkoituksena on luoda Flow Field -polut annetuista aloituspaikasta tai -paikoista tiettyyn määränpäähän pelikentällä. Ensin A*-algoritmia käyttäen algoritmi hakee portaaleita pitkin polun yhdestä aloituspaikasta määränpäähän. Kun A* löytää polun portaaligraafista, se

luo portaaleista linkitetyt listan, jossa on polun mukaisesti on viittaus aikaisemmasta portaalista seuraavaan portaaliin. Tällä listalla voidaan selvittää, mitä sektoreita tullaan tarvitsemaan Flow Field -polunetsintäalgoritmissa. Jos aloituspaikkoja on vain yksi, lista päättyy tähän. Kuitenkin aloituspaikkoja voi olla useammassa sektorissa, jolloin myös näille sektoreille täytyy suorittaa A*-algoritmin polunetsintä. Uudet A*-algoritmin suoritukset yhdistetään linketyssä listassa aikaisempaan suoritukseen lisäämällä uuden suorituksen uudet viittaukset listassa olleisiin viittauksiin. Kun polut kaikista aloituspaikoista on löydetty, lista on valmis ja tarvittavat sektorit ovat selvillä, jolloin jokaisen tarvittavan sektorin kohdalla lähetetään Flow Field -pyyntö. [6]

3.2.5 Integraattori

Flow Field -pyynnöt suoritetaan integraattorilla, joka on erityinen luokka Flow Field -polkujen muodostamiseen. Integraattori luo jokaiselle sektorille oman Flow Field -virtauksen. Sektorissa portaalit toimivat sekä aloituspaikkoina että määränpäinä, ja aikaisemman linkitetyn listan perusteella portaali määritellään joko aloituspaikaksi tai määränpääksi. Erityistapaus on, jos sektorissa sijaitsee polunetsintäpyynnön varsinainen määränpää, jolloin se asetetaan sektorin määränpääksi portaalin sijaan. Integraattori toimii siten, että se lähettää annetusta määränpäästä aaltoja, jotka leveyshaun tapaisesti tutkivat jokaisen kuljettavan ruudun ja asettavat niille arvoja. Aalto pysähtyy, kun se törmää esteeseen tai sektorin reunaan. Integraattorin tarkoituksena on luoda kaksi kenttää: ensin integraatiokenttä ja lopuksi Flow Field -kenttä. [6]

3.2.6 Integraatiokenttä

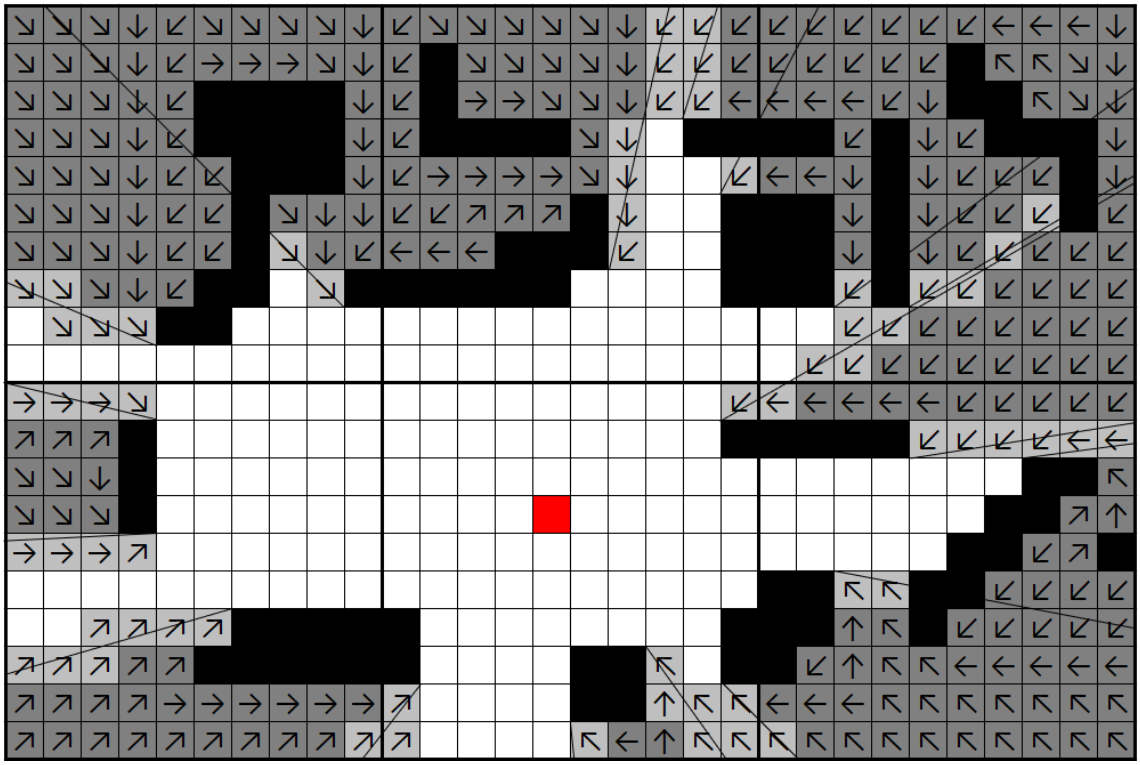
Integraattori aloittaa integraatiokentän luomisella. Integraatiokentän muodostamisen aikana integroidaan heuristisen arvojen lisäksi näkölinja. Integraattori lähettää

kaksi aaltoa, joista ensimmäisen tarkoitus on määrittää näkölinjan omaavat ruudut ja joista toisen tarkoitus on jäljelle jääneisiin ruutuihin integroida heuristiset arvot. Näkölinja määritetään jokaiselle ruudulle, jonka kustannusarvo on yksi. Integraation aikana törmätessä ruutuun, jonka kustannusarvo on suurempi kuin yksi, selvitetään ollaanko saavuttu näkölinjan kulmaan. Kulma voi olla seinä pelikentällä, jonka läpi ei voi kulkea tai maasto, jossa kulkeminen on hitaampaa. Kulma määritellään katsomalla ruudun naapurien kustannusten arvoa. Jos naapuriruudulla on kustannusarvona yksi, mutta sitä vastakkaisella ruudulla ei ole, tiedetään, että ollaan löydetty näkölinjan kulma. Tällöin käynnistyy Bresenhamin algoritmin[22], joka alkaa kulman reunasta piirtämään lineaarista viivaa poispäin määränpäästä. Tämä viiva rajaa kulman takana olevat ruudut sen perusteella, että näkyykö kulman takaa ruudusta määränpää. Ruudut, jotka näkevät määränpään, merkitään näkölinjan omaaviksi ja ruudut, jotka eivät näe, jätetään merkitsemättä. Kun ensimmäinen aalto päättyy, alkaa toinen aalto, joka määrittää merkitsemättä jääneille ruuduille heuristiset arvot. [6]

Flow Field -pyynnöt tulisi käsitellä järjestyksessä, joka muistuttaa aaltoa. Ensimmäinen sektori, jonka Flow Field -pyyntöä käsitellään tulisi olla se sektori, jossa polunetsintäpyynnön määränpää sijaitsee. Tämän jälkeen tulisi käsitellä sektorit, jotka ovat määränpääsektorin naapureita ja aina seuraavia naapureita, kunnes kaikki sektorit on käyty läpi. Tätä järjestystä käyttäen integraattori pystyy uudessa sektorissa integroidessaan lukemaan aikaisemman sektorin integraatiokentän arvoja niiden välisen portaalin kautta ja jatkamaan niitä uudessa sektorissa. Se myös pystyy tuomaan näkölinjan kulman uudelle sektorille lukemalla, mikä portaalin ruutu on osa näkölinjaa ja mikä ei ole. Täten saadaan aikaan integraatiokenttiä, jotka vaikuttavat yhdeltä suurelta integraatiokentältä, mikä taas avustaa parempien Flow Field -kenttien luomista. [6]

3.2.7 Flow Field -kenttä

Integraatiokentän valmistuttua integraattori lähettää uuden aallon, jonka tarkoitus on luoda Flow Field -kenttä. integraattori tarkastelee jokaisen integraatiokentän ruudun, ja mikäli ruudulla on näkölinja, integraattori jättää sen tarkastelematta. Tarkastelussa integraattori katsoo ruudun kahdeksaa ympäröivää naapuria, määrittelee, millä niistä on pienin heuristinen arvo ja asettaa ruutuun vektorinuolen osoittamaan tätä naapuriruutua kohti. Kun aalto on käynyt koko sektorin läpi, Flow Field -kenttä on valmis. [6] Kuvassa 3.3 esitetään valmis polunetsintäpyyntö.



Kuva 3.3: Supreme Commander 2:n polunetsinnän tulos ruudukossa.

Punainen ruutu on maali, valkoiset ruudut ovat näkölinjan omaavia ruutuja ja mustat ruudut esteitä. Viivat ovat Bresenhamin algoritmin piirtämiä näkölinjan kulman rajauksia. Vaaleat harmaat ruudut ovat näkölinjan reunalla kulman taakse jääneitä ruutuja ja harmaat ruudut ovat täysin kulman takana olevia ruutuja. Ruudukon tummennetut viivat tarkoittavat sektoreiden reunoja.

Kaikkien Flow Field -pyyntöjen valmistuttua, polunetsintäpyyntö on valmis, ja yksiköiden on mahdollista kulkea pelikentällä luotuja polkuja pintkin. Polunetsintäpyynnön tulos voidaan tallentaa muistiin ja sille antaa ID. Tämä mahdollistaa vastaavan polunetsintäpyynnön lataamisen suoraan muistista ilman, että tarvitsisi tuottaa koko polunetsintä kokonaan uudestaan. [6]

3.3 Flow Field nykyisten RTS-pelien kehityksessä

Supreme Commander 2:ssa käytettyä Flow Field -polunetsintäalgoritmia on hyödynnetty sen jälkeen tulleissa RTS-peleissä. *Age of Empires IV* on 2021 julkaistu RTS-peli, jonka teemana on keskiaikainen sodankäynti. Frank Cheng *Age of Empires IV:n* Lead Navigation Engineer esitteli peliin liittyviä polunetsintäongelmia ja niiden ratkaisuja GDC:ssä vuonna 2022. [19] *Age of Empires IV:n* polunetsintäalgoritmin kehitykseen liittyi paljon samanlaisia ongelmia kuin *Supreme Commander 2:n* kehitykseen, kuten A*-algoritmilla törmäystapauksien käsittelyssä. Ratkaisu näihin ongelmiin löytyi käyttämällä pelissä Flow Field -polunetsintäalgoritmia, joka muistuttaa hyvin paljon *Supreme Commander 2:n* polunetsintäalgoritmia. *Age of Empires IV:n* polunetsintäalgoritmi käyttää sektoreita, portaaligraafeja ja näkölinjaa polunetsinnän aikana. Selvästi *Supreme Commander 2:n* polunetsintäalgoritmi on inspiroinut *Age of Empires IV:n* polunetsintäalgoritmia, mikä osoittaa Flow Field -algoritmin toimivuuden nykyaikaisena RTS-pelien polunetsintäalgoritmina.

3.4 Vastaukset tutkimuskysymyksiin

Ensimmäinen tutkimuskysymys oli: Kuinka A* ja Flow Field -algoritmit eroavat toisistaan reaaliaikastrategiapelien polunetsintäalgoritmeina? Vastauksena tähän kysymykseen löydettiin, että A* kykenee tuottamaan yhden polun yhdelle yksikölle, kun taas Flow Field kykenee tuottamaan useamman polun useammalle yksikölle. Tämän

vuoksi Flow Field pystyy tehokkaammin etsimään polkuja, kun se käsittelee suuria määriä yksiköitä, ja vähentämään konfliktien syntymistä useampien yksiköiden liikkuessa samaan aikaan pelikentällä. Toinen tutkimuskysymys oli: Kuinka *Supreme Commander 2:ssa* sovelletaan Flow Fieldia tehokkaan polunetsinnän saavuttamiseksi? Vastauksena löydettiin, että *Supreme Commander 2:ssa* Flow Fieldia sovelletaan toimimaan älykkäästi, jotta se tuottaa nopeasti laadukkaita polkuja vähemmällä muistin ja laskutoimituksien tarpeella. Tämä saavutetaan soveltamalla Flow Fieldin rinnalla A*:ia ja erilaisia menetelmiä sektoreita, portaaleita ja näkölinjaa.

4 Yhteenveto

Tämän tutkielman tarkoituksena oli tarkastella A* ja Flow Field algoritmien soveltamista reaaliaikastrategiapeleissä ja erityisesti tarkastella, kuinka hyvin Flow Fieldia sovellettu *Supreme Commander 2:ssa*. Tutkielman tuloksena huomataan, että A*-algoritmi sopii hyvin yksittäisten yksikköjen polunetsinnässä, kun taas Flow Field soveltuu paremmin useampien yksiköiden polunetsinnässä. Tämän voi huomata Kumalan ja Istionon tutkimuksessa [21], jossa Flow Field säännöllisesti johdatti useampia yksikköjä nopeammassa ajassa maaliin kuin A*. Flow Fieldin käytössä huomataan, että voidaan vähentää ongelmien syntymistä, kun useammat yksiköt seuraavat polkuja pelikentällä samanaikaisesti.

Tästä syystä ja syystä, että Flow Field pystyy hakemaan tehokkaasti polkuja useammalle yksikölle, uusissa RTS-peleissä, kuten *Supreme Commander 2* ja *Age of Empires IV*, Flow Fieldia on hyödynnetty paljon tehokkaammaan polunetsinnän mahdollistamiseksi. Tässä Flow Fieldiin perustuvassa polunetsintäalgoritmissa hyödynnetään lisäksi A*-algoritmia, sektoreita, portaaligraafia ja näkölinjaa, älykkään polunetsinnän saavuttamiseksi. Lopputuloksena on polunetsintäalgoritmi, joka tuottaa parhaimmat mahdolliset polut ja samalla pyrkii minimoimaan raskaiden operaatioiden tarvetta.

Tulevaisuuden kannalta RTS-pelien polunetsintäalgoritmeja voitaisiin tutkia lisää, sillä tämän hetkisten tutkimusten määrä on hyvin vähäistä. Varsinkin Flow Field -algoritmin käyttöä RTS-peleissä tulisi tutkia lisää tieteellisesti, koska tutki-

muksen määrä tästä on lähes olematonta. Erityisesti voitaisiin tutkia polunetsintäalgoritmien tehokkuutta RTS-peleissä, kun niille annetaan eri kokoisia yksikkömääriä käsiteltäviksi. Esimerkiksi tutkimuksessa [21] olisi voitu vaihtaa jokaisen pelikentän kohdalla yksikköjen määrää ja tutkia molempien A* ja Flow Fieldin kohdalla, kuinka nopeasti ne pystyvät ohjaamaan kaikki yksiköt maaliin. Tämä olisi voinut antaa tietoa algoritmien kyvyistä käsitellä eri yksikkömääriä ja oltaisiin voitu arvioida molempien algoritmien soveltuvuutta pienien sekä isojen yksikkömäärien polunetsinnässä.

Supreme Commander 2:sta [6] ja *Age of Empires IV:sta* [19] tehdyt raportoinnit niissä toteutetuista polunetsintäalgoritmeista auttavat ymmärtämään, mikä on mahdollista RTS-pelien polunetsinnässä, ja antavat näkemystä, mihin suuntaan polunetsintäalgoritmeja voitaisiin kehittää. Näin ollen uusista tulevista RTS-peleistä, kuten Stormgate, Tempest Rising ja Zerospace, olisi hyödyllistä saadaan tietoa siitä, miten niissä polunetsintä on toteutettu ja mitä uusia tekniikoita on mahdollisesti kehitetty pelin polunetsinnän parantamiseksi.

Lähdeluettelo

[1] D. M. Bourg ja G. Seemann, *AI for game developers*. O'Reilly, heinäkuu 2004, 400 s., ISBN: 0-596-00555-5. url: https://www.academia.edu/8267834/AI_For_Game_Developers_2004_.

[2] K. Khantanapoka ja K. Chinnasarn, "Pathfinding of 2D and 3D game real-time strategy with depth direction A* algorithm for multi-layer", teoksessa *2009 Eighth International Symposium on Natural Language Processing*, Bangkok, Thaimaa: IEEE, 1. joulukuuta 2009, ISBN: 978-1-4244-4138-9. DOI: 10.1109/snlp.2009.5340922.

[3] A. S. Wale, T. S. Saini, A. M. Ali ja V. S. Jagtap, "Survey Heuristic Search for Shortest Path finding in Games", India, joulukuu 2015, ISBN: 978-93-85225-53-6.

[4] S. S. Parth Mehta Hetasha Shah ja S. Verma, "A Review on Algorithms for Pathfinding in Computer Games", teoksessa *International Conference on Innovations in Information Embedded and Communication Systems*, IEEE, maaliskuu 2015.

[5] J. Anhalt. "AI Navigation: It's Not a Solved Problem - Yet", GDC. (2011), url: https://www.gdcvault.com/play/1014514/AI-Navigation-It-s-Not (viitattu 01. 04. 2024).

[6] E. Emerson, ”Crowd pathfinding and steering using flow field tiles”, teoksessa *Game AI Pro 360: Guide to Movement and Pathfinding*, CRC Press, 10. syyskuuta 2019, s. 67–76, ISBN: 9780429055096.

[7] M. Thompson. ”Supreme Commander 2: Chris Taylor speaks to Ars”, Ars Technica. (26. helmikuuta 2010), url: https://arstechnica.com/gaming/2010/02/chris-taylor-interview/ (viitattu 01. 04. 2024).

[8] R. Moss. ”Build, gather, brawl, repeat: The history of real-time strategy games”. (15. syyskuuta 2017), url: https://arstechnica.com/gaming/2017/09/build-gather-brawl-repeat-the-history-of-real-time-strategy-games/ (viitattu 03. 08. 2024).

[9] C. McCole. ”What Makes RTS Games Fun: Why We Play RTS”, Callum "GGTheMachine"McCole. (4. marraskuuta 2019), url: https://callummccole.com/2019/11/04/what-makes-rts-games-fun-why-we-play-rts/ (viitattu 03. 08. 2024).

[10] R. Lara-Cabrera, C. Cotta ja A. J. Fernandez-Leiva, ”A review of computational intelligence in RTS games”, teoksessa *2013 IEEE Symposium on Foundations of Computational Intelligence (FOCI)*, IEEE, huhtikuu 2013. DOI: 10.1109/foci.2013.6602463.

[11] J. Shafer. ”Turn-Based VS Real-Time”, Game Developer. (7. tammikuuta 2013), url: https://www.gamedeveloper.com/design/turn-based-vs-real-time (viitattu 03. 08. 2024).

[12] X. Xu ja K. Zou, ”An Improved Pathfinding Algorithm in RTS Games”, teoksessa *Advanced Research on Computer Science and Information Engineering*, G. Shen ja X. Huang, toim., vol. 153, Zhongshan, Kiina: Springer, Berlin Heidelberg, 22. toukokuuta 2011, s. 1–7, ISBN: 978-3-642-21411-0. DOI: 10.1007/978-3-642-21411-0_1.

[13] E. W. Rivas ja R. Souza, "Map Marker: a Multi-Agent Pathfinder for Cohesive Groups in Real-Time Strategy Games", teoksessa *2021 20th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)*, Gramado, Brazilia: IEEE, 20. joulukuuta 2021, ISBN: 978-1-6654-0189-0. DOI: 10.1109/sbgames54170.2021.00021.

[14] J. Helsing ja A. Bruce, "A Scalability and Performance Evaluation of Precomputed Flow Field Maps for Multi-Agent Pathfinding", kanditaatintutkielma, Blekinge Institute of Technology, Sweden, toukokuu 2022.

[15] M. H. Naveed, "Automated Planning for Pathfinding in Real-Time Strategy Games", tohtorinväitöskirja, University of Huddersfield, Yhdistynyt kuningaskunta, helmikuu 2012.

[16] R. E. Korf, "Depth-first iterative-deepening: An optimal admissible tree search", *Artificial Intelligence*, vol. 27, nro 1, s. 97–109, syyskuu 1985, ISSN: 0004-3702. DOI: 10.1016/0004-3702(85)90084-0.

[17] A. Zafar, K. K. Agrawal ja W. C. Anil Kumar, "Analysis of Multiple Shortest Path Finding Algorithm in Novel Gaming Scenario", teoksessa *Intelligent Communication, Control and Devices*. Springer Singapore, 11. huhtikuuta 2018, s. 1267–1274, ISBN: 978-981-10-5903-2. DOI: 10.1007/978-981-10-5903-2_132.

[18] E. W. Dijkstra, "A note on two problems in connexion with graphs", *Numerische Mathematik*, vol. 1, nro 1, s. 269–271, joulukuu 1959, ISSN: 0945-3245. DOI: 10.1007/bf01386390.

[19] F. Cheng. "Pathing in 'Age of Empires IV': Flow Fields and Steering Behaviors", GDC. (maaliskuu 2022), url: https://www.gdcvault.com/play/1027659/Pathing-in-Age-of-Empires (viitattu 23. 01. 2024).

[20] P. Hart, N. Nilsson ja B. Raphael, ”A Formal Basis for the Heuristic Determination of Minimum Cost Paths”, *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, nro 2, s. 100–107, heinäkuu 1968, ISSN: 0536-1567. DOI: 10.1109/tssc.1968.300136.

[21] G. T. Kumala ja W. Istiono, ”Comparison of Flow Field and A-Star Algorithm for Pathfinding in Tower Defense Game”, *International Journal of Multidisciplinary Research and Analysis*, vol. 5, nro 9, s. 2445–2453, syyskuu 2022, ISSN: 2643-9875. DOI: 10.47191/ijmra/v5-i9-20.

[22] J. E. Bresenham, ”Algorithm for computer control of a digital plotter”, *IBM Systems Journal*, vol. 4, nro 1, s. 25–30, 1965, ISSN: 0018-8670. DOI: 10.1147/sj.41.0025.