Lappeenrannan teknillinen yliopisto
School of Business and Management
Tietotekniikan koulutusohjelma

Kandidaatintyö

Mika Savolainen

REITINHAUN TOTEUTTAMINEN 2D-PELEISSÄ UNITY3D:LLÄ

Työn tarkastaja: Tutkijatohtori Antti Knutas

Työn ohjaaja: Tutkijatohtori Antti Knutas

TIIVISTELMÄ

Lappeenrannan teknillinen yliopisto
School of Business and Management
Tietotekniikan koulutusohjelma

Mika Savolainen

Reitinhaun toteuttaminen 2D-peleissä Unity3D:llä

Kandidaatintyö

2017

34 sivua, 19 kuvaa, 2 taulukkoa, 3 liitettä

Työn tarkastaja: Tutkijatohtori Antti Knutas

Hakusanat: reitinhaku, unity3D, peli, 2D, a-tähti reitinhaku

Keywords: pathfinding, unity3D, game, 2D, astar pathfinding

Unity3D ei tarjoa valmista keinoa toteuttaa reitinhakua 2D-pelinkehityksessä. Reitinhaun hyödyntämiseen täytyy käyttää joko ulkoista liitännäistä tai kehittää se itse. Monet liitännäiset ovat suunniteltuja sekä 2D-, että 3D-peliprojekteille, joten niiden käyttäminen voi olla vaikeaa. Reitinhaku on tärkeä osa pelejä, koska se määrittää, miten yksiköt käyttäytyvät. Työssä toteutettiin 2D:lle suunnattu reitinhakuliitännäinen, joka mahdollistaa reitinhaun 2D-peliprojekteissa. Liitännäinen on helppo lisätä peliprojekteihin, mutta ei ole yhtä tehokas kuin monet nykyiset ratkaisut. Sen jatkokehittämisessä kannattaa keskittyä paremman verkoston luomiseen.

ABSTRACT

Lappeenranta University of Technology
School of Business and Management
Degree Program in Computer Science

Mika Savolainen

Implementing pathfinding for 2D games with Unity3D

Bachelor's Thesis

2017

34 pages, 19 figures, 2 tables, 3 appendices

Examiner: D.Sc. Antti Knutas

Keywords: pathfinding, unity3D, game, 2D, astar pathfinding

Unity3D doesn't offer a solution for using pathfinding in 2D game projects. In 2D games you must use prebuild assets or create one by yourself. Pathfinding is an important part of the games because it defines how units behave. That's why we created pathfinding asset that can be used for adding pathfinding into game projects. Asset is easy to use and can create well-behaving units, but it isn't as powerful as previous solutions. When continuing the development of the asset, you should focus on creating better grids and memory management. For beginners, it is better to use a prebuild asset because it is easier, but an expert would prefer to use their own so they know what everything does. Pathfinding isn't a solved issue and everyone is trying to improve their pathfinding.

ALKUSANAT

Tämä kandidaatintyö tehtiin Lappeenrannan teknillisen yliopiston tietotekniikan koulutusohjelmassa. Haluan kiittää avopuolisoani, joka kannusti minua työnteossa. Kiitos myös työnohjaajalle työnteon opastamisessa ja hyvistä neuvoista.

SISÄLLYSLUETTELO

1 JOHDANTO 3

1.1 TAUSTA 4

1.2 TAVOITTEET JA RAJAUKSET 4

1.3 TYÖN RAKENNE 4

2 KIRJALLISUUSKATSAUS REITINHAUSTA 5

2.1 REITINHAKUALGORITMIT 5

2.1.1 Breadth-first search 6

2.1.2 Dijkstran algoritmi 7

2.1.3 Ahne algoritmi 7

2.1.4 A-algoritmi 8*

2.2 AIKAISEMMAT RATKAISUT 11

2.3 MITÄ ASIOITA EI OLE AIKAISEMMISSA TOTEUTUKSISSA VIELÄ HUOMIOITU? 12

3 LIITÄNNÄISEN TOTEUTUS JA KÄYTTÖ 13

3.1 LIITÄNNÄISEN KÄYTTÄMINEN 13

3.2 LIITÄNNÄISEN OMINAISUUDET 18

4 LIITÄNNÄISTEN VERTAILU JA JATKOKEHITTÄMINEN 20

4.1 ESITTELY A* PATHFINDING PROJECT -LIITÄNNÄISESTÄ 20

4.2 LIITÄNNÄISTEN VERTAILU 20

4.3 KESKUSTELU TESTIEN TULOKSISTA 21

4.4 LIITÄNNÄISEN JATKOKEHITTÄMINEN 23

5 YHTEENVETO 24

LÄHTEET 26

SYMBOLI- JA LYHENNELUETTELO

2D	Kaksiulotteinen
3D	Kolmeulotteinen
A*	A-tähti
KB	Kilotavu
ms	Millisekunti

1 JOHDANTO

Reitinhaku on tärkeä osa pelien tekoälyä, koska se määrittää, miten yksiköt, kuten pelaaja ja viholliset, käyttäytyvät [1]. Reitinhaku vie paljon laskentatehoa ja sitä varten tarvitaan tehokas ratkaisu, jotta reitinetsintä ei suorita paljon laskelmia ja hidasta peliä. Unity3D-pelimoottorin [2] reitinhaku on suunnattu 3D (kolmiulotteisille) peleille, ja se ei tarjoa reitinhakua, joka hyödyntää Unity3D:n 2D (kaksiulotteisia) ominaisuuksia [3]. Reitinhaun käytössä joutuu hyödyntämään jonkun muun kehittämää valmista ratkaisua tai toteuttamaan sen itse. Monet valmiit maksuttomat reitinhakuratkaisut ovat suunnattuja sekä 2D-, että 3D-pelinkehitykseen, ja ne eivät saata sisältää kaikkia toivottuja ominaisuuksia. Ne voivat myös olla vaikeita käyttää ja niiden käytössä voi tulla vastaan ongelmia, joita on vaikea itsenäisesti korjata. Reitinhakujen välillä on isoja eroja ja väärän ratkaisun käyttö voi tuottaa ei toivottuja tuloksia, kuten liian pitkiä reittejä tai turhaa prosessointia [4].

Reitinhakua käytetään reitinetsintään verkostosta pisteen A ja pisteen B välillä. Tätä reittiä käytetään yksiköiden liikuttamiseen pelimaailmassa. Reitinhaku on monimutkainen prosessi ja siksi reitinhakutoteutuksen täytyy olla mahdollisimman tehokas. Reitti etsitään uudestaan, jos pisteiden sijainti vaihtuu, jotta reitti on paras mahdollinen ja varmistetaan, että kuljetaan oikeaan suuntaan. Hyvä reitinhaku saa yksiköt näyttämään luonnolliselta ja ei riko pelikokemusta [5]. Reitinhaussa suositaan A*-reitinhakualgoritmia (A-tähti), mutta on muitakin reitinhakualgoritmeja, jotka sopivat eri tarkoituksiin [1].

Tässä työssä toteutettiin reitinhakuliitännäinen, jota voi käyttää Unity3D:llä pelienkehittämisessä. Reitinhakuliitännäinen on lisäosa, joka on tarkoitettu 2D-pelinkehitykseen. Liitännäinen on helppo lisätä omaan projektiin ja mahdollistaa sovittamisen omaan käyttötarkoitukseen sopivaksi.

1.1 Tausta

Tämän työn aihe on valittu mielenkiinnosta peliohjelmointiin ja halusta luoda jotain kompleksia peleihin liittyen. Tarvitsen reitinhakua omassa peliprojektissani ja tämän työn pohjalta toteutan toimivan ratkaisun omaan projektiini. Olen käyttänyt muiden ohjelmoimia reitinhakuliitännäisiä, mutta ne eivät toimi, kuten haluan. Haluan kehittää yksinkertaisemman ratkaisun, jota voin muokata omaan käyttöön sopivaksi ja tarjota sen myös muille käytettäväksi.

1.2 Tavoitteet ja rajaukset

Työssä tutkitaan reitinhakualgoritmeja ja luodaan oma reitinhakuliitännäinen Unity3D-pelimoottorilla ja käyttämällä C#-ohjelmointikieltä. Liitännäinen on suunniteltu käytettäväksi missä tahansa 2D-peliprojektissa, joka on tehty Unity3D:llä. Reitinhakuliitännäisen on tarkoitus olla helppokäyttöisempi kuin monet muut ratkaisut, koska se on suunnattu vain 2D-peliprojekteille. Liitännäisen on tarkoitus olla tehokas, jotta se ei aiheuta alhaista kuvataajuutta (*engl. frame rate*). Liitännäisellä pitäisi olla helppoa luoda reitinhakua hyödyntäviä yksiköitä, jotka käyttäytyvät luonnollisen näköisesti. Työssä keskitytään oman reitinhakuliitännäisen toteuttamiseen ja ei reitinhaun muihin ongelmiin, kuten reitinhaussa käytettävän ruudukon parantamiseen.

1.3 Työn rakenne

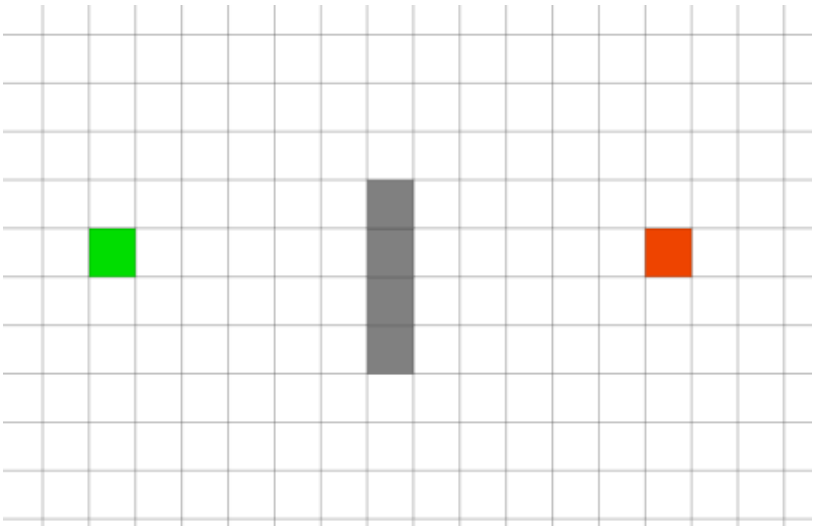
Johdannon jälkeen luvussa kaksi esitellään teoriaa reitinhakualgoritmeista ja niiden toiminnasta. Luvussa kerrotaan, miten A*-reitinhakualgoritmi toimii ja mitä aikaisempia toteutuksia reitinhausta on Unity3D:llä. Kolmannessa luvussa esitellään toteutettu liitännäinen ja miten sitä käytetään. Neljännessä luvussa kerrotaan Aron Granbergin kehittämästä reitinhakuliitännäisestä ja kehitetyn liitännäisen vertailusta siihen. Siinä pohditaan myös, miten kehitettyä liitännäistä kannattaa jatkokehittää saatujen tulosten perusteella. Työn viimeisessä luvussa on yhteenveto käydyistä asioista ja pohditaan myös reitinhaun tulevaisuutta.

2 KIRJALLISUUSKATSAUS REITINHAUSTA

Tässä luvussa esitellään työssä hyödynnetyt reitinhakualgoritmit ja Unity3D:n aikaisempia toteutuksia reitinhausta. Työ vaatii paljon tietoa eri alueilta, mutta tärkein aihealue on reitinhakualgoritmit. Lyhyimmän reitin etsiminen on ongelma, jota yritetään ratkaista monilla aloilla [6]. Peleissä reitinhakua käytetään esimerkiksi pelaajan tai tekoälyllisen yksikön liikuttamisessa. Reitin pystyy etsimään monella eri tapaa, mutta löydetyn reitin pitäisi usein olla mahdollisimman lyhyt ja helppo löytää. Ilman reitinhakua tekoälylliset yksiköt jäävät kiinni esteisiin, kun ne esimerkiksi seuraavat pelaajaa. Reitinhaun avulla ne välttävät esteet ja liikkuvat pelaajaa kohti.

2.1 Reitinhakualgoritmit

Reitinhakualgoritmit etsivät reitin kahden pisteen väliltä. Reitti etsitään kuvan 1. kaltaisesta verkosta (*engl. grid*). Verkosto on joukko pisteitä, joista muodostetaan reitti. Se on tietorakenne, jonka voi esittää eri ulkomuodoissa, kuten ruudukkona tai seikkailupelin maastona [7].



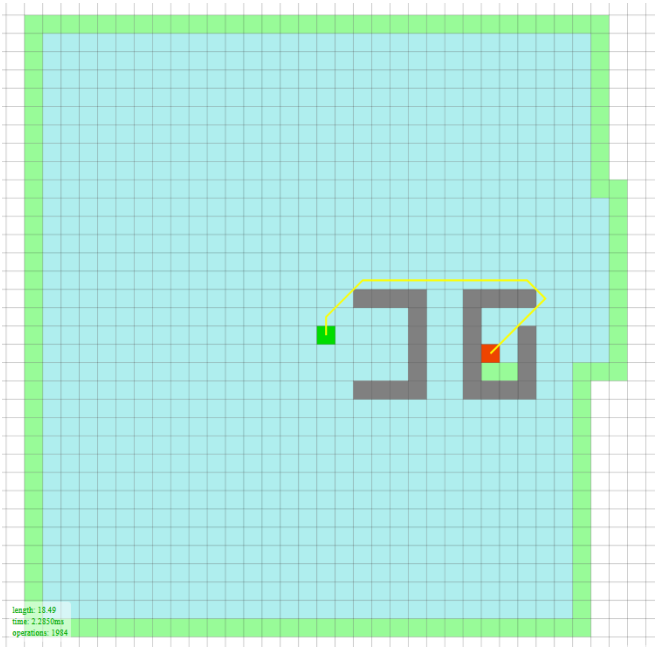
Kuva 1. Esimerkki verkosto, esteistä, alku- ja päätepisteestä.

Algoritmi tuottaa reitin, joka muodostuu verkoston pisteistä. Saadusta reitistä yleensä halutaan mahdollisimman lyhyt. Algoritmin halutaan myös etsivän reitti tehokkaasti ja reitinetsintäprosessin toivotaan olevan mahdollisimman nopea. Reitti on helppo etsiä, jos ruudukko on pieni eikä pisteiden välillä ole esteitä. Mitä enemmän pisteitä ruudukossa on, sitä enemmän vaihtoehtoja algoritmi joutuu tutkimaan [8].

Reitinhakualgoritmeja on monia ja niitä voidaan käyttää eri tilanteissa. A*-reitinhakualgoritmi yleisesti käytetyin, koska se on nopea ja löytää lyhyimmän reitin [1]. Muita reitinhakualgoritmeja voidaan käyttää erikoisissa tilanteissa esimerkiksi, jos on monia päätepisteitä tai halutaan löytää reitti nopeasti, mutta ei välitetä reitin pituudesta. Seuraavaksi esitellään eri reitinhakualgoritmeja ja niiden toimintaa.

2.1.1 Breadth-first search

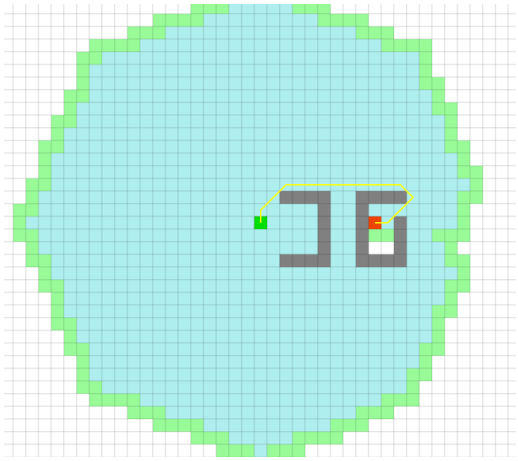
Breadth-first search -algoritmi etsii lyhyimmän reitin tutkimalla koko verkoston. Algoritmi palauttaa aina lyhyimmän reitin kyseiseen pisteeseen, mutta tutkii hyvin paljon turhaa tietoa, kuten kuvasta 2 nähdään. Tästä syystä algoritmi on hidas ja ei yleisesti suosittu ratkaisu [6].



Kuva 2. Reitinhaku Breadth-first search -algoritmilla.

2.1.2 Dijkstran algoritmi

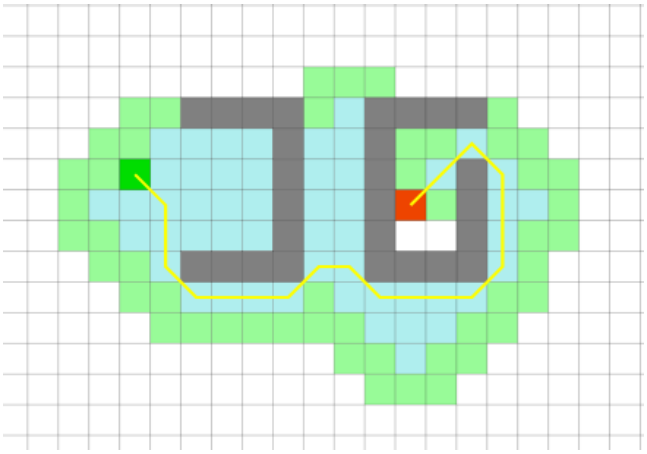
Dijkstran algoritmi toimii lähes samankaltaisesti kuin Breadth-first search -algoritmi, mutta se ottaa huomioon pisteiden välisen etäisyyden. Breadth-first search -algoritmi olettaa, että kaikkien pisteiden välillä liikkuminen on samanarvoista. Dijkstran algoritmin avulla voidaan esimerkiksi huomioida, että viistoliikkeet ovat pidempiä ja näin antaa niille vähemmän arvoa [4]. Dijkstran algoritmi ei ole enää suosittu normaalissa reitinhaussa, mutta sitä voidaan käyttää, jos päätepisteitä on monia ja halutaan muodostaa reitti lähimpään pisteeseen.



Kuva 3. Reitinhaku Dijkstran algoritmilla.

2.1.3 Ahne algoritmi

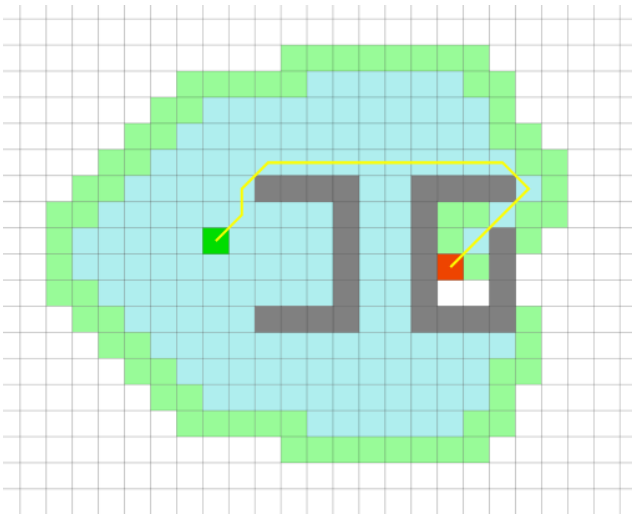
Ahne (*engl. greedy*) algoritmi etsii reitin tutkimalla lupaavinta pistettä eli pistettä, josta on lyhyin matka loppupisteeseen. Algoritmi palauttaa ensimmäisen löydetyn reitin ja siksi se voi olla nopea. Tämä algoritmi ei kuitenkaan ole ideaali, koska se ei usein palauta lyhyintä reittiä [4]. Kuten 4 kuvasta nähdään, reitti on pidempi kuin reitinhaussa Dijkstran algoritmilla.



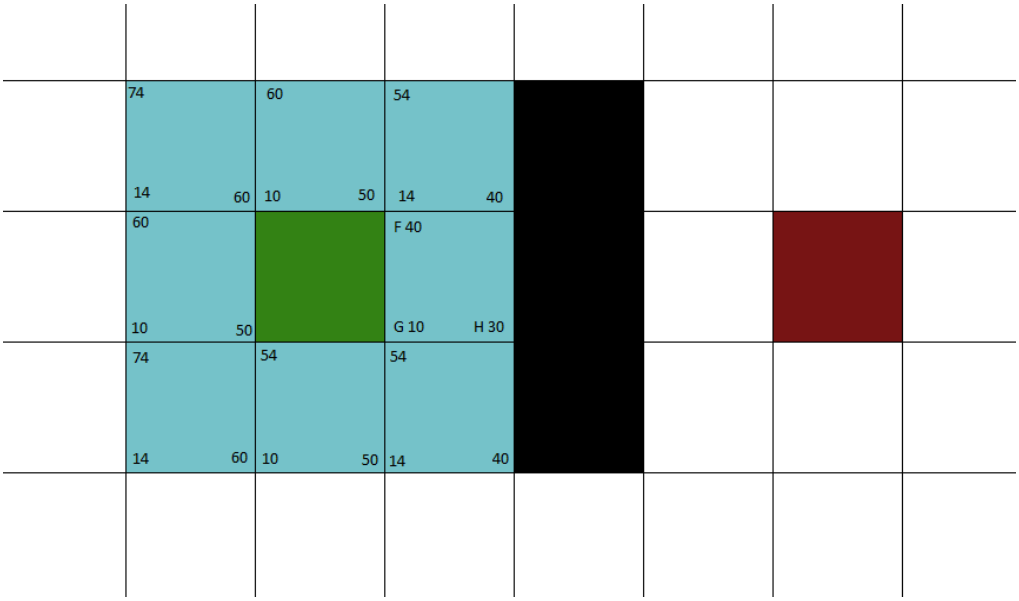
Kuva 4. Reitinhaku ahneella algoritmilla.

2.1.4 A*-algoritmi

A*-algoritmi on edellisten algoritmien yhdistelmä. Se hyödyntää reitinhaussa pisteiden etäisyyttä alkupisteeseen ja loppupisteeseen. Algoritmi tutkii vähemmän pisteitä kuin Dijkstran algoritmi, mutta palauttaa silti lyhyemmän reitin kuin ahne algoritmi [6].



Kuva 5. Reitinhaku A*-algoritmilla.



Kuva 6. A*-algoritmin ensimmäisten arvojen laskenta.

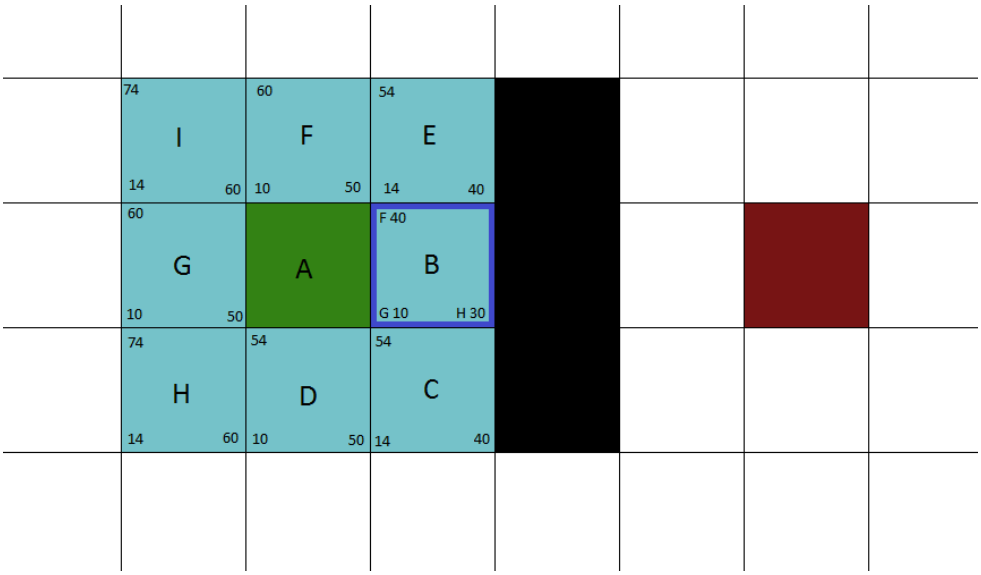
Kuvassa 6 esitellään A*-algoritmin toimintaa. A*-algoritmi aloittaa reitinetsinnän alkupisteestä, joka on piste A ruudukossa. Reitinhaussa hyödynnetään kahta listaa, avointa ja suljettua. Avoin lista on lista mahdollisista pisteistä, jotka saattavat muodostaa reitin. Suljettu lista on lista pisteitä, jotka on tutkittu.

Alkupiste ja sen ympäröivät pisteet lisätään avoimeen listaan. Pisteiden isäntäpisteeksi merkitään alkupiste. Alkupiste poistetaan avoimesta listasta ja lisätään suljettuun listaan. Avoimille pisteille lasketaan kolme arvoa, G-, H- ja F-arvo. G-arvo kertoo, kuinka kaukana piste on lähtöpisteestä. H-arvo kertoo, kuinka kaukana piste on loppupisteestä. F-arvo on G- ja H-arvon summa [8].

Tässä esimerkissä pysty- ja poikittaisliikkeen pituus on 10 ja vinoittaisliikkeen pituus on 14. Kuvasta 7 näemme esimerkin arvojen laskennasta, G-arvo pisteelle B on 10 ja H-arvo on 30. Laskemalla nämä arvot yhteen saamme F-arvon, joka on 40.

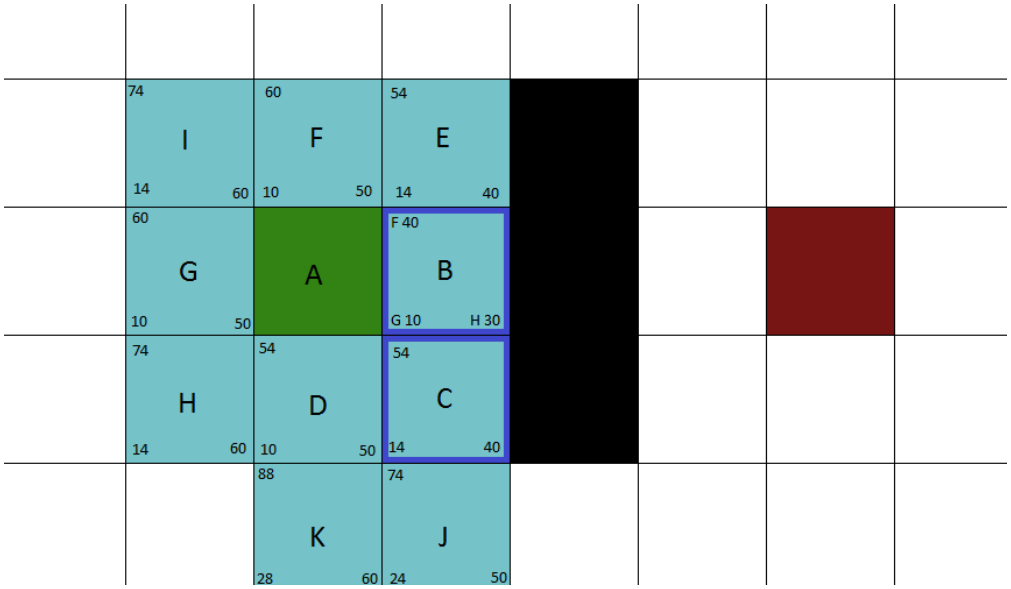
Algoritmi valitsee pisteen, jolla on pienin F-arvo (piste B), poistaa sen avoimesta listasta ja lisää sen suljettuun listaan. Tämän jälkeen käydään läpi sen naapuripisteet, mutta ei huomioida esteitä (mustalla), eikä pisteitä, jotka ovat jo suljetussa listassa (piste A). Piste B:n naapuripisteet ovat jo avoimessa listassa, joten tarkistetaan, onko reitti piste B:n kautta

lyhyempi kuin niiden vanhan isäntäpisteen kautta. Tässä käytetään hyväksi G-arvoa. Pisteen C nykyinen G-arvo on 14 ja uusi arvo kulkemalla pisteen B kautta on 20. Eli uusi reitti ei ole parempi, joten ei tehdä muutoksia. Tämä johtopäätös tehdään myös muille ympäröiville pisteille.



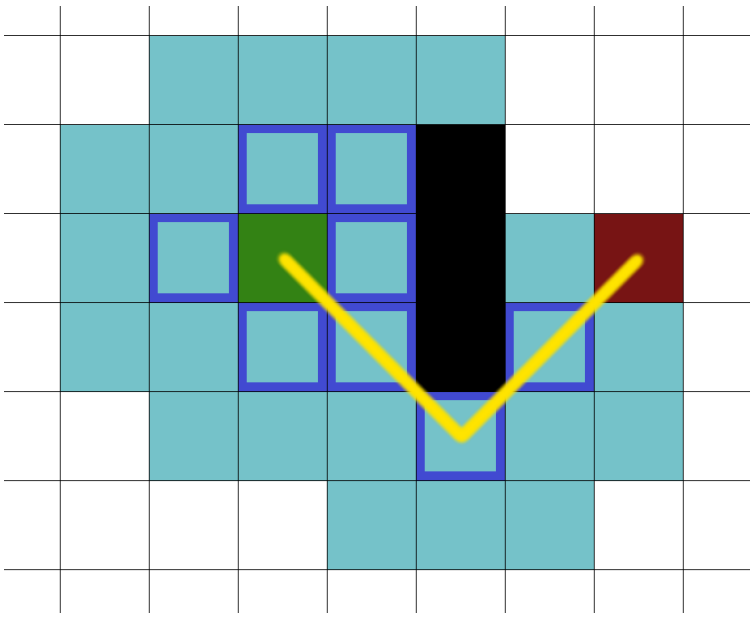
Kuva 7. Piste B lisättiin suljettuun listaan.

Avoimen listan läpi käyntiä jatketaan valitsemalla uusi piste. Valitaan uusi piste satunnaisesti pisteen C ja E väliltä, koska niillä on sama F-arvo. Valitaan piste C ja siirretään se avoimesta listasta suljettuun listaan.



Kuva 8. Piste C lisättiin suljettuun listaan.

Kaksi uutta löydettyä pistettä lisätään avoimeen listaan ja asetetaan niille isäntäpisteeksi piste C. Tätä prosessia toistetaan, kunnes löydetään reitti loppupisteeseen eli saadaan lisättyä se suljettuun listaan. Lopullinen etsintäprosessi on kuvan 9 kaltainen.



Kuva 9. Lopullinen etsintäprosessi. Keltaisella löydetty reitti.

2.2 Aikaisemmat ratkaisut

Reitinhaku on yksi tärkeimmistä osista pelejä, koska monien pelihahmojen käyttäytyminen määräytyy sen mukaan [1]. A*-algoritmi on yleisesti käytetty reitinhakualgoritmi, koska se on tarkka ja nopea [9]. Algoritmin toteutuspohja on kaikilla sama, mutta jokaisella on omat keinonsa optimoida kyseistä algoritmia.

A*-reitinhakualgoritmin pystyy toteuttamaan lähes millä tahansa ohjelmointikielellä. Tässä opintonäytetyössä tutkimme reitinhaun toteuttamista Unity3D-pelimoottorilla. Unity3D:lle tunnettu reitinhakutoteutus 2D-peliprojekteille on Aron Granbergin tekemä reitinhakuliitännäinen [10]. Tästä liitännäisestä on ilmainen versio jaossa, mutta siitä pystyy myös ostamaan paremman version, joka sisältää lisäominaisuuksia. Muita ratkaisumenetelmiä ovat esimerkkisi Poly|Nav – 2D Pathfinding [11].

2.3 Mitä asioita ei ole aikaisemmissa toteutuksissa vielä huomioitu?

Unity3D:ssä on oma sisäänrakennettu reitinhakusysteemi, mutta sitä ei voi hyödyntää 2D-peliprojekteissa. Monet ovet tehneet omia reitinhakuliitännäisiä, mutta ne ovat 3D:lle sekä 2D:lle suunniteltuja, kuten Aron Granbergin tekemä reitinhakuliitännäinen [10]. Mielestäni pystyn luomaan liitännäisen, joka on suunniteltu vain 2D:lle ja sen takia on käyttäjäystävällisempi kuin aikaisemmat ratkaisut. Työssä toteutettu reitinhakuliitännäinen on tarkoitus olla tehokas ja helppo käyttöinen. Liitännäistä pitäisi olla myös helppo jatkokehittää ja sovittaa eri projekteihin sopivaksi.

Reitinhaun toteutustapa Unity3D-pelimoottorilla vaihtelee. Yksinkertainen reitinhaku on helppo toteuttaa Unity3D:lle, mutta hyvin optimoitu ratkaisu vaatii paljon tutkintaa. DigiPen Institute of Technology yliopiston Introductory AI -kurssilla opiskelijoilla oli tehtävänä luoda A*-reitinhakutoteutus. Opiskelijoiden luomien algoritmien erot olivat satakertaiset nopeimman ja hitaimman välillä [4].

Valmiit reitinhaku ratkaisut saattavat toimia hyvin, mutta niiden toimintaa voi olla vaikeaa ymmärtää. 2D:lle että 3D:lle suunnattujen ratkaisujen on tarkoitus toimia molemmissa ympäristöissä ja siksi niiden ohjelmakoodi saattaa olla vaikeaa lukea. Tästä syystä liitännäisten jatkokehittäminen tai ongelmatilanteiden korjaaminen voi olla vaikeaa ilman kehittäjän apua.

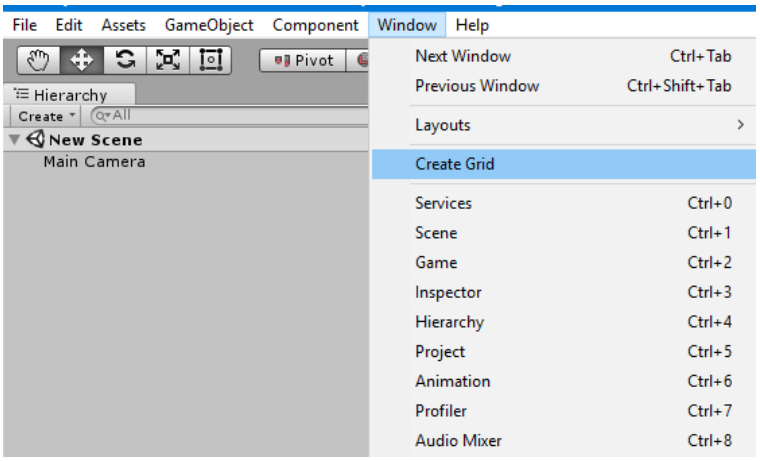
3 LIITÄNNÄISEN TOTEUTUS JA KÄYTTÖ

Työssä toteutettiin reitinhakuliitännäinen, jonka pystyy lisäämään Unity3D-peliprojekteihin. Liitännäisen tarkoitus on helpottaa reitinhaun toteuttamista 2D-peliprojekteissa. Liitännäinen kehitettiin Unity3D-pelimoottorilla ja C#-ohjelmointikielellä. Liitännäisen toteuttamiseen hyödynnettiin Sebastian Laguen toteutusta reitinlaskennasta ja kekolistasta (*engl. heap*) [12]. Reitin hyödyntämistä varten toteutettiin yksinkertainen esteitä välttävä yksikkö, joka liikkuu hiirellä painettavaan suuntaan.

Liitännäistä käyttämällä pystytään yksiköille lisäämään reitinhaku, joka hyödyntää 2D verkostoa ja A*-algoritmia reitinetsinnässä. Verkosto on 2D matriisi, joka koostuu pisteistä. Verkoston pisteet ovat joko käveltäviä tai esteitä, joita ei huomioida reitinhaussa. Pisteille voi lisätä painoarvoa, jolloin reitinhaku suosii tiettyjä pisteitä esim. teitä.

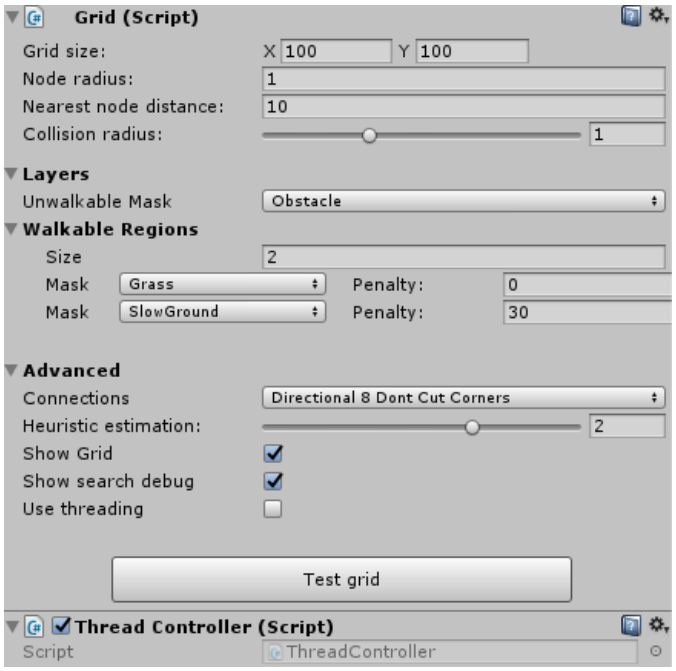
3.1 Liitännäisen käyttäminen

Liitännäistä käyttääkseen avataan Window -> Create Grid kuten kuvassa 10. Tämä luo pelielementin (*engl. game object*), jossa on Grid-komponentti (*engl. component*), sekä Thread Controller -komponentti.



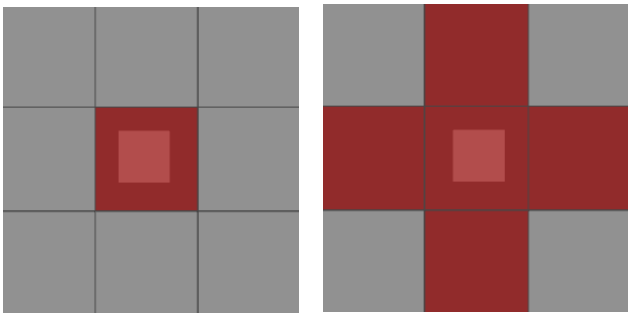
Kuva 10. Grid-pelielementin luominen.

Pelielementti valitaan pelinäkymästä, jotta voidaan muokata verkoston ominaisuuksia. Kuvassa 11 näkyvät kaikki Grid-komponentin ominaisuudet, joita pystytään muuttamaan tarkkailunäkymästä (*engl. inspector*). Grid size -asetus määrittää, miten iso verkosto on ja Node radius -asetus määrittää, miten isoja solmut ovat. Verkostokoolla 100x100 ja solmukoolla yksi luodaan kuvan 14. kaltainen verkosto, jossa on 2500 solmua. Nearest node distance -asetus määrittää, kuinka kaukaa yritetään etsiä seuraava mahdollinen solmu, jos tavoitesolmu on esteen sisällä. Collision radius -asetus määrittää, miten isolta alalta suhteessa solmun kokoon tarkistetaan, onko solmu este. Esteiden tarkistus tehdään toteuttamalla *CircleCast* [13] ja vertaamalla osuttiinko esteeseen. Suuremmalla arvolla kuin yksi tutkitaan esteitä solmun ulkopuolelta ja isompi alue määritetään esteeksi kuten kuvassa 12.



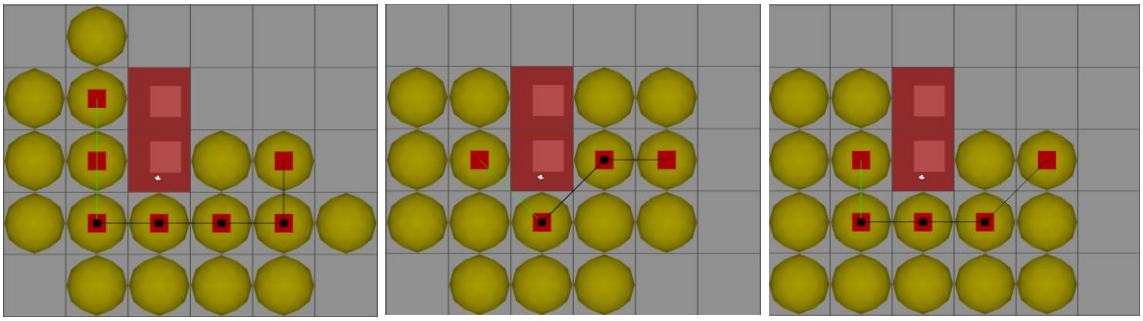
Kuva 11. Grid tarkkailunäkymässä.

Unwalkable mask -asetuksessa valitaan tasoja (*engl. layer*) [14], jotka määrittävät, mitä objekteja pidetään esteinä. Walkable regions -asetuksella voidaan määrittää tasojen rangaistusarvo (*engl. penalty*). Käveltävien tasojen rangaistuksen oletusarvo on nolla ja tätä arvoa lisäämällä saadaan reitinhakijat välttämään esimerkiksi hidasta maastoa.



Kuva 12. Esteiden tarkastus arvolla 1 ja arvolla 2.

Connections-asetuksessa on kolme vaihtoehtoa, ja ne määrittävät, miten solmut ovat kytketty toisiinsa. 4 Directional -asetus ei huomioi viistosuuntia, 8 Directional -asetuksen kanssa voidaan liikkua viistosuuntaan ja 8 Directional don't cut corners -asetuksella ei ylitetä kulmia. Kuvassa 13 nähdään näiden vaikutus. Heuristic estimation -asetus vaihtaa pisteiden heuristista painoarvoa. Arvolla yksi liitännäinen käyttäytyy kuin standardi A* kuten kuvassa 17 ja arvolla 2 se käyttäytyy ahneen algoritmin kaltaisesti kuten kuvassa 18. Show grid -asetus näyttää verkoston pelinäkymässä ja Show search debug -asetus näyttää reitinetsintäprosessin kuvan 13 kaltaisesti. Use threading -asetus käyttää toista ydintä reitinlaskentaan, jolloin tämä ei hidasta pääydintä ja Test grid -painikkeella nähdään luotu verkosto kuvan 14 kaltaisesti pelinäkymässä.

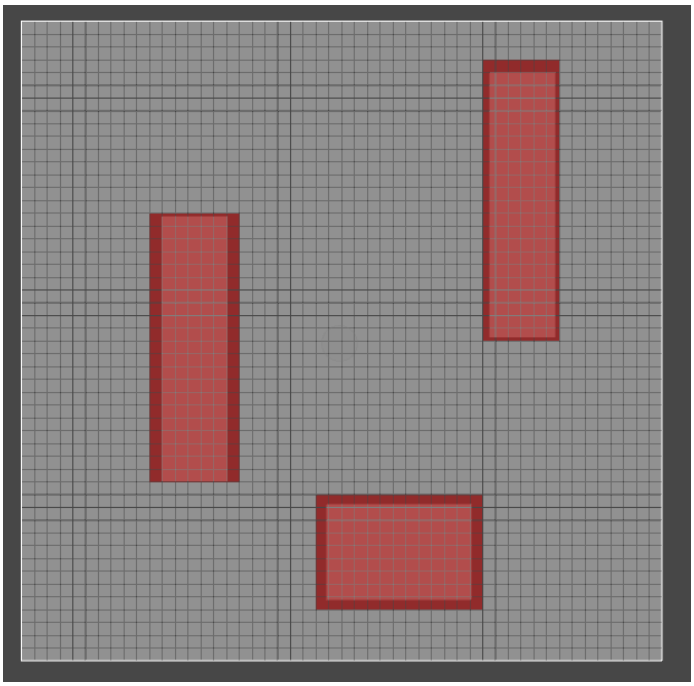


(a) (b) (c)

Kuva 13. Connections-asetuksen vaikutus. (a) 4 Directional (b) 8 Directional (c) 8 Directional Don't Cut Corners.

Reitinhaun oleellinen osa ovat esteet, joiden läpi ei voida kulkea, ja joita ei huomioida reitinhaussa. Ilman esteitä ei ole varsinaisesti mitään syytä etsiä reittiä, koska kohteeseen voi liikkua suoraan. Grid-pelielementissä on määritetty, mitkä tasot ovat esteitä. Esteille pitää

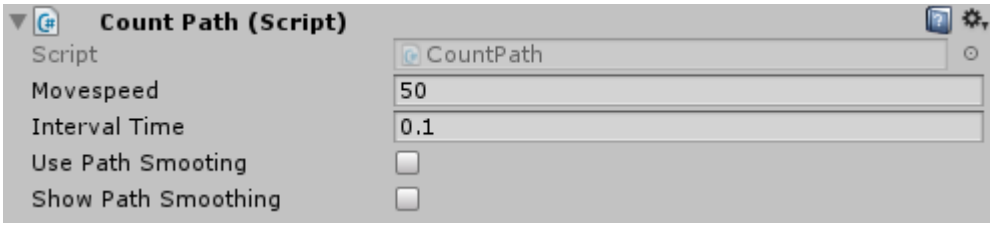
valita tämä sama lajittelutaso, sekä niillä pitää olla jokin Unity3D:n Collider2D -komponentti [14, 15]. Käyttämällä Test grid -painiketta nähdään kuvan 14 esittämällä tavalla punaiset alueet, jotka ovat esteitä.



Kuva 14. Esimerkki luodusta verkostosta.

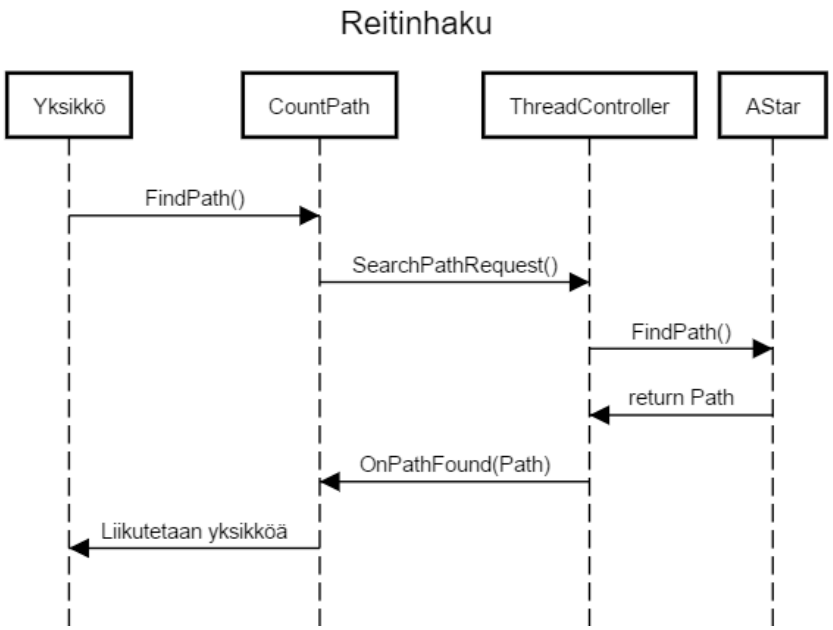
Reitinhaun hyödyntämistä varten yksilölle täytyy lisätä CounthPath-komponentti [16]. Kuvassa 15 nähdään komponentin asetukset. Movespeed-arvo määrittää, miten nopeasti yksikkö liikkuu ja Interval Time -arvo määrittää, kuinka monen sekunnin välein uusi reitti etsitään. Use smoothing -asetus hyödyntää dynaamista esteiden tutkimista ja Show Path Smoothing -asetus näyttää sen prosessin pelinäkymässä.

Liite 3 on komponentin koodi, joka hyödyntää CountPath-komponenttia ja havainnollistaa reitinhaun käyttämistä yksiköllä. Liitteen koodin avulla yksikkö liikkuu sinne, minne hiiren vasemmalla painikkeella painetaan. Tarvitaan muuttuja CountPath, jolle annetaan arvo Start-funktiossa [17]. Tämän jälkeen voidaan kutsua CountPath-luokan FindPath(transform, targetPosition) -funktio, missä transform on reitinhakua käyttävä yksikkö ja targetPosition on päätepisteen koordinaatti.



Kuva 15. CountPath tarkkailunäkymässä.

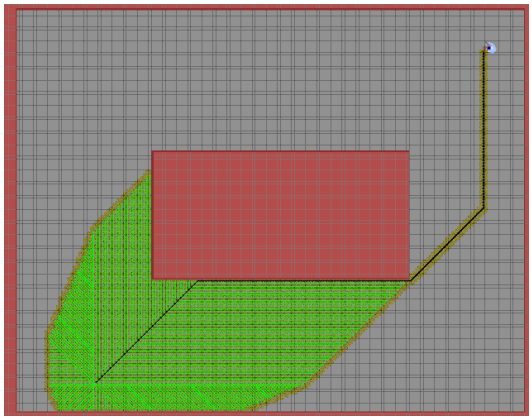
CountPath – komponentti lähettää käskyn reitinetsintään, sekä liikuttaa yksikköä. Kuvassa 16 on kuvattu reitinhakuprosessi sekvenssikaavion avulla. Etsijä lähettää CountPath-komponenttiin FindPath-funktio kutsun, joka kutsuu ThreadController-luokan SearchPathRequest-funktiota. ThreadController-luokka suorittaa AStar-luokan FindPath-funktion muussa kuin pääytimessä, jos käytetään Use threading -asetusta Grid-pelielementissä. ThreadController-luokka etsii lähtö- ja päätesolmun verkosta hyödyntäen lähtö- ja päätekoordinaatteja. Näitä solmuja käytetään AStar-luokassa reitinetsintään, jossa sen FindPath-funktio suorittaa A*-algoritmin mukaisen reitinhaun verkosta käyttämällä annettuja solmuja. Funktio palauttaa koordinaateista koostuvan matriisin, joista muodostuu kuvan 18 kaltainen reitti, jota pitkin CounPath-komponentti liikuttaa yksikköä.



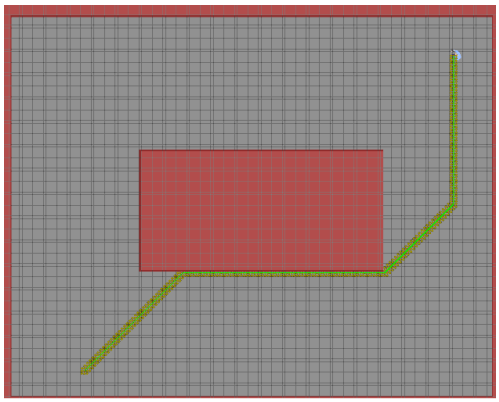
Kuva 16. Sekvenssikaavio reitinhakuprosessista.

3.2 Liitännäisen ominaisuudet

Liitännäisessä pystyy muuttamaan heuristiikan painoa ja määrittämään, haluaako liitännäisen toimivan kuin Dijkstran algoritmi vai ahne algoritmi. Lähtökohtaisesti liitännäinen yliarvioi heuristiikan *(engl. overestimated heuristics)* ja toimii enemmän ahneen algoritmin kaltaisesti. Yliarvioimalla heuristiikkaan liitännäinen laskee vähemmän solmuja ja reitti on usein sama. Kuvassa 17 ja 18 nähdään, miten yliarvioitu heuristiikka vähentää tutkittujen solmujen määrää, mutta palauttaa saman reitin. Heuristiikan arvolla voi määrittää, haluaako lyhyimmän reitin vai nopean algoritmin [4].

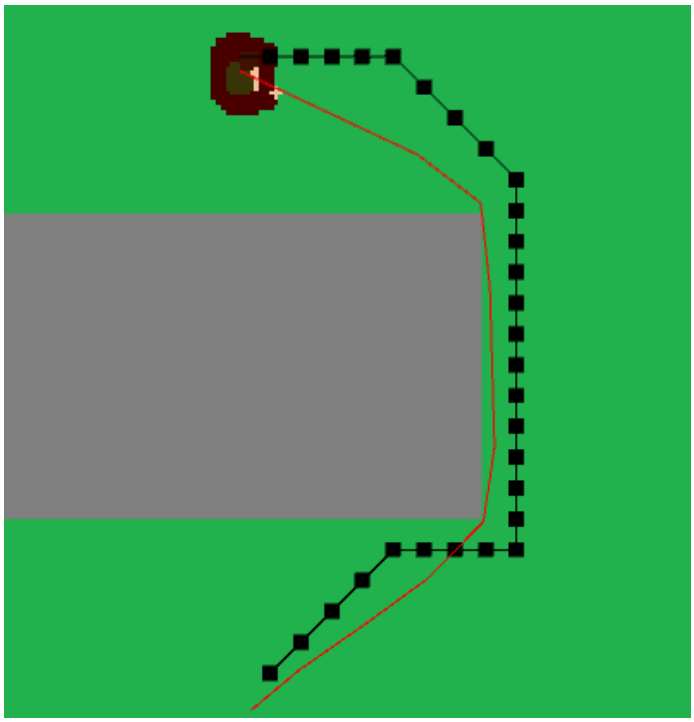


Kuva 17. Normaali heuristiikka. Solmuja laskettu 6003.



Kuva 18. Yliarvioitu heuristiikka. Solmuja laskettu 640.

Liitännäinen on suunniteltu avoimelle pelille, joten yksiköiden pitää pystyä liikkumaan vapaasti. Verkosto mahdollistaa liikkumisen vain kahdeksaan eri suuntaan, jolloin liikkuminen voi vaikuttaa luonnottomalta. Myös liiallinen reitin seuraaminen voi näyttää robottimaiselta [18]. Tästä syystä yksiköillä on mahdollista käyttää dynaamista esteiden tarkastamista, jonka avulla yksiköt voivat liikkua vapaammin ja poiketa niille määrätystä reitistä. Dynaaminen esteiden tarkastus tutkii, onko mahdollista sen hetkisestä sijainnista liikkua seuraavaan pisteeseen. Jos tämä on mahdollista, aloitetaan liikkuminen seuraavaan pisteeseen ja tutkitaan taas seuraavaa pistettä. Tämä saa liikkumisen näyttämään luonnollisemmalta, mutta voi johtaa siihen, että yksikkö ei enää kulje reittiä pitkin. Kuvassa 19 nähdään, miten liikkuminen on paljon pyöreämpää kuin reittiä seuraamalla. Mustalla on merkittynä suunniteltu reitti ja punaisella kuljettu reitti.



Kuva 19. Dynaaminen esteidentarkastusprosessi.

4 LIITÄNNÄISTEN VERTAILU JA JATKOKEHITTÄMINEN

4.1 Esittely A* Pathfinding Project -liitännäisestä

Suosittu ilmainen ratkaisu 2D-reitinhakuun on Aron Granbergin kehittämä A* Pathfinding Project [10]. A* Pathfinding Project on 2D- ja 3D-pelinkehitykseen suunnattu reitinhakuliitännäinen, jota voi käyttää Unity3D:llä peliprojekteissa. Liitännäisellä voi luoda navigointiverkkoja (*engl. navmesh*), ruudukkoverkostoja (*engl. grid graph*) tai pisteverkostoja (*engl. point graph*). Liitännäinen on hyvin dokumentoitu ja sisältää useita demokenttiä (*engl. scene*), joissa näkee, miten liitännäistä voi käyttää.

Liitännäisellä 2D-reitinhaussa käytetään ruudukkoverkostoa, jota pitää pyörittää -90 astetta X- akselin suuntaa. A* Pathfinding Project jakaa lasketun verkoston alueisiin sen mukaan, onko mahdollista luoda reitti alueesta toiseen. Näin liitännäinen ei yritä laskea koko verkostoa, jos reittiä ei ole mahdollista luoda. Verkosto on myös mahdollista ladata etukäteen ja tallentaa se muistiin.

A* Pathfinding Projectista on maksullinen versio, joka tarjoaa lisäominaisuuksia, kuten mahdollisuuden suorittaa reitinhaun useammalla kuin kahdella ytimellä. Maksullinen versio myös tarjoaa mahdollisuuksia lisäoptimointiin, kuten parempia heuristiikkoja.

4.2 Liitännäisten vertailu

Toteutetun liitännäisen ja A* Pathfinding Projectin välillä tehtiin yhdeksän eri testiä. Testeissä käytettiin Unity3D versiota 5.6.1f1, A* Pathfinding Project versiota 4.0.10 ja toteutetun liitännäisen versiota päivämäärältä 15.6.2017. Testeissä vertailtiin tutkittujen solmujen määrää, reitinetsinnän kestoa, reitinhaussa käytetyn muistinmäärää ja löydetyn reitin solmujen määrää. Liitännäisiä vertailtiin kolmessa erilaisessa verkostossa, jossa oli eri määrä esteitä. Esteitä oli ei yhtään, yksi tai kaksi, joiden asettelu nähdään liitteessä 1. Testeissä oli myös eri määrä solmuja, joko 100X100 tai 200X200. Testejä tehtiin myös eri heuristiikka-arvoilla, jotta molemmat liitännäiset käyttäytyisivät kuin Dijkstran

reitinhakualgoritmi ja tutkittujen solmujen määrät olisivat yhtä suuret. Testeissä käytettiin myös Unity3D:n Deep profile -asetusta, jonka avulla pystyttiin tutkimaan tarkemmin ohjelman suorittamista, mutta sen käyttäminen hidasti koko ohjelman suoritusaikaa [19].

Taulukko 1. Testitapaukset.

	Verkoston koko	Solmujen määrä	Käveltäviä solmuja	Esteitä	Heuristiikan arvo
1	100X100	10000	8099	1901	1
2	100X100	10000	6995	3005	1
3	100X100	10000	6509	3491	1
4	200X200	40000	33123	6877	1
5	200X200	40000	28891	11109	1
6	200X200	40000	27098	12902	1
7	100X100	10000	8099	1901	0
8	100X100	10000	6995	3005	0
9	100X100	10000	6509	3491	0

4.3 Keskustelu testien tuloksista

Taulukosta 2 näemme, että A* Pathfinding Project on nopeampi kuin kehitetty liitännäinen lähes joka tilanteessa. Muutamassa testitapauksessa kehitetty liitännäinen laskee vähemmän solmuja, mutta A* Pathfinding Project on silti nopeampi. A* Pathfinding Project vie myös paljon vähemmän muistia, koska se vapauttaa muistin tehokkaasti ja ei luo jatkuvasti uusia muuttujia. Tämä kannattaa huomioida kehitetyn liitännäisen jatkokehittämisessä.

A* Pathfinding Project ei laske heuristiikka-arvolla yksi kuten standardi A* vaan käyttää ahneempaa algoritmia. Tämä nähdään etsintäprosessista yksi, jossa sen olisi pitänyt tutkia lähes yhtä paljon solmuja kuin kehitetty liitännäinen. Tämä teki etsintäprosessien vertailusta vaikeampaa, mutta käyttämällä nolla heuristiikka-arvoa saatiin tutkittujen solmujen määrä samaksi ja pysyttiin vertaaman lähes samankaltaisia tilanteita.

Löydettyjen reittien pituudet ovat lähes samat. Kehitetyn liitännäisen löytämä reitti on usein yhden solmun pienempi, koska se lopettaa reitinhaun, kun loppupiste on avoimessa listassa. A* Pathfinding Project lopettaa reitinetsinnän, kun loppupiste on suljetussa listassa. Tämä

hidastaa reitinhakua vähän, mutta jos solmuilla on eri painoarvoja, reitti on yhä lyhyin mahdollinen.

Taulukko 2. Testitulokset.

	Testitapaus	**Aika (ms)**	**Aika Deep profile -asetuksella (ms)**	**Muistinmäärä (KB)**	**Tutkittujen solmujen määrä**	**Reitinpituus (Solmua)**
Kehitetty liitännäinen	1	2.62	19.08	242	336	70
A* Pathfinding Project	1	4.51	8.01	36.8	71	71
Kehitetty liitännäinen	2	10.22	129.96	1200	1658	99
A* Pathfinding Project	2	4.5	9.01	29.2	100	100
Kehitetty liitännäinen	3	36.56	334.34	2900	3972	131
A* Pathfinding Project	3	30.52	310.26	38.1	5764	132
Kehitetty liitännäinen	4	4.52	34.29	800	661	139
A* Pathfinding Project	4	5.02	17.52	54.8	230	142
Kehitetty liitännäinen	5	56.21	501.14	4600	6003	195
A* Pathfinding Project	5	5.51	27.52	55.9	430	198
Kehitetty liitännäinen	6	163.73	1550.88	11500	15713	260
A* Pathfinding Project	6	122.61	1345.34	57.1	24608	263
Kehitetty liitännäinen	7	59.74	489.27	4600	6315	70
A* Pathfinding Project	7	29.03	295.78	28.6	6288	71
Kehitetty liitännäinen	8	43.59	377.75	3900	5231	99
A* Pathfinding Project	8	25.04	248.25	25.2	5212	100
Kehitetty liitännäinen	9	378.12	333.16	3600	4738	131
A* Pathfinding Project	9	23.03	216.22	25.7	4724	132

A* Pathfinding Project jakaa reitinetsinnän monelle laskentasyklille, jotta yhden laskentasyklin kesto on korkeintaan 15 ms (millisekuntia). Tämä hidastaa reitinhaunlaskentaa, mutta on silti kannattava ratkaisu, koska pelaaja reagoi enemmän alhaiseen kuvataajuuteen kuin myöhäiseen reitinhakuun. Kehitetty liitännäinen laskee reitin yhdellä laskentasyklillä, jolloin tämä kestää 100 ms.

Testeistä huomataan, että kaikissa tilanteissa A* Pathfinding Project liitännäinen on paljon nopeampi. Erot ovat suuret ja vaikka A* Pathfinding Project on suunnattu 2D:lle ja 3D:lle, sen 2D ominaisuudet toimivat todella hyvin.

4.4 Liitännäisen jatkokehittäminen

Liitännäisen jatkokehittämisessä on kannattavaa keskittyä verkoston parantamiseen, koska se on helpoin ja tehokkain tapa nopeuttaa reitinhakua [4]. Parempaan verkostoon voidaan käyttää esim. navigointiverkostoa tai pisteverkostoa. On iso ero, pitääkö reitti etsiä sadan vai tuhannen solmun verkostosta.

Liitännäiseen kannattaa lisätä ominaisuuksia A* Pathfinding Projectista, joiden avulla voidaan välttää alhaista kuvataajuutta. Näitä ovat esimerkiksi reitinhaun jakaminen monelle laskentasyklille ja verkoston jakaminen alueisiin. Liitännäinen voisi myös tarjota reitinetsinnän käytön useammalla kuin yhdellä lisäytimellä kuten A* Pathfinding Projecting maksullinen versio.

Reitinhaussa pitää joskus päivittää verkosto, jos esteissä tapahtuu muutoksia. Verkoston päivittäminen on monimutkaista ja se kannattaa tehdä vain pienille alueille kerrallaan. Koko verkoston uudelleen laskeminen voi viedä aikaa ja pilata pelaajan pelikokemuksen. Verkoston pisteillä kannattaa olla erillinen tieto siitä, voiko se muuttua esteeksi tai esteestä käveltäväksi. Verkostoon merkitään, mitkä ovat staattisia esteitä ja mitkä eivät. Näin tiedetään, mitä pisteitä ei tarvitse uudelleen tarkistaa.

5 YHTEENVETO

Työssä toteutettiin reitinhakuliitännäinen, joka on suunniteltu käytettäväksi Unity3D:llä tehtäviin 2D-peliprojekteihin. Liitännäisen avulla käyttäjä voi luoda verkoston ja lisätä reitinhaun yksiköille. Liitännäinen toteutettiin käyttämällä Unity3D:tä ja C#-ohjelmointikieltä.

Liitännäistä käyttämällä luodaan verkosto ja reitinhakua käyttäville yksiköille lisätään CountPath-komponentti, josta voidaan kutsua reitinhakukäsky. Komponentti etsii reitin ja liikuttaa yksikköä reittiä pitkin. Liitännäisestä haluttiin yksinkertainen ja helppokäyttöinen, ja sen haluttiin luovan luonnollisen näköistä liikettä. Toteutettua liitännäistä vertailtiin Aron Granbergin toteuttamaan A* Pathfinding Projectiin ja pärjäsi vertailussa toivottua huonommin. Kehitetty liitännäinen kuitenkin tarjoaa yksinkertaisen reitinhakuratkaisun, jota pitää vielä jatkokehittää, jos halutaan siitä yhtä tehokas kuin A* Pathfinding Projectista. Liitännäisen jatkokehittämisessä keskitytään verkoston parantamiseen ja parempaan muistinhallintaan.

Unity3D-pelimoottorilla ei ole omaa 2D-peliprojekteille suunnattua reitinhakua ja sellaista ei ole toistaiseksi suunnitteilla [20]. Aloittelijan on kannattavaa käyttää valmiiksi luotuja liitännäisiä, jos haluaa toteuttaa reitinhaun, mutta kokeneempi voi pitää parempana ratkaisuna toteuttaa oman reitinhakuratkaisun, jotta tietää miten se toimii. Unity3D on 3D-pelimoottori lähtökohtaisesti, joten voi mennä aikaa ennen kuin he toteuttavat 2D-reitinhaun. He muuttivat TextMesh Pro:n maksuttomaksi, koska se oli parempi ratkaisu kuin heidän oma käyttöliittymätekstin hallintajärjestelmä [21]. Joten on mahdollista, että Unity3D tarjoaa jonkun toisen reitinhakuliitännäisen yleiseen käyttöön.

Reitinhaku ei ole vielä ratkaistu ongelma. Monet pelinkehittäjät jatkuvasti optimoivat reitinhakutoteutuksiaan, jotta ne olisivat tehokkaampia. Reitinhaun parannuksessa ei ole tapahtunut isoja algoritmisia harppauksia, kuten Dijkstrasta A*: een siirtyminen, mutta monia muita eri keinoja on löydetty. Kaikki optimoinnit eivät ole kaikkiin peleihin sopivia ja kehittäjän täytyy itse keksiä, mitkä toimivat hänen pelissään. Yleisiä hyviä ohjeita on

välttää suurta reitinhakumäärää ja optimoida verkosto. Tarkoituksellisesti jätetty huono reitinhaku voi olla hyödyksi, koska se lisää vaadittua taitoa pelin pelaamiseen [22].

LÄHTEET

[1] Graham, R., McCabe, H., Sheridan, S. (2003) "Pathfinding in Computer Games,"The ITB Journal: Vol. 4: Iss. 2, Article 6

[2] Unity Technologies (2017). Unity3D [verkko]. Saatavilla: <https://unity3d.com/>. [Viitattu 12.4.2017].

[3] Unity Technologies (2017). Navigation and Pathfinding [verkko]. Saatavilla: <https://docs.unity3d.com/Manual/Navigation.html>. [Viitattu 6.9.2017].

[4] Rabin, S., Sturtevant, N. (2013). Pathfinding Architecture Optimizations. Game AI Pro: Collected Wisdom of Game AI Professionals. 1, pp.241-252.

[5] Schnieders B. (2014). AIPath: Movement Planning in Physically Constrained Domains. Master Thesis. Maastricht University, Faculty of Humanities and Sciences, Department of Knowledge Engineering. Maastricht. 85 s.

[6] Zarembo, I., Kodors, S. (2013). Pathfinding Algorithm Efficiency Analysis in 2D Grid. In: Proceedings of the 9th International Scientific and Practical Conference., 2013, Rezekne Higher Education Institution. pp. 46-50.

[7] Patel, A. (2017). Introduction to A*. [verkko]. Red Blob Games. Saatavilla: <http://www.redblobgames.com/pathfinding/a-star/introduction.html/>. [Viitattu 12.4.2017].

[8] Hu, J., Wan, W., Yu, X., (2012). A Pathfinding Algorithm in Real-time Strategy Game based on Unity3D. In: 2012 International Conference on Audio, Language and Image Processing, 2012, Shanghai University. pp. 1159-1162.

[9] Sithu Kyaw, A., Peters, C., Naing Swe, T., (2013). Unity 4.x Game AI Programming. 1st. ed. Birmingham: Packt Publishing Ltd.

[10] Granberg A. (2017). A* Pathfinding Project Home [verkko]. Saatavilla: <http://arongranberg.com/astar/>. [Viitattu 4.3.2017].

[11] Vaggelis G. (2014). Poly|Nav - 2D Pathfinding. [verkko]. AssetStore Unity3D Saatavilla: <https://www.assetstore.unity3d.com/en/#!/content/14718>. [Viitattu 4.3.2017].

[12] Lague S. (2015). Pathfinding 2D. (Software). [Ladattu 23.4.2017]. Saatavilla: <https://github.com/SebLague/Pathfinding-2D>.

[13] Unity Technologies (2017). Physics2D.CircleCast [verkko]. Saatavilla: <https://docs.unity3d.com/ScriptReference/Physics2D.CircleCast.html>. [Viitattu 5.9.2017].

[14] Unity Technologies (2017). Layers [verkko]. Saatavilla: <https://docs.unity3d.com/Manual/Layers.html>. [Viitattu 5.9.2017].

[15] Unity Technologies (2017). Collider2D [verkko]. Saatavilla: <https://docs.unity3d.com/Manual/Collider2D.html>. [Viitattu 6.9.2017].

[16] Unity Technologies (2017). Using Components [verkko]. Saatavilla: <https://docs.unity3d.com/Manual/UsingComponents.html>. [Viitattu 4.9.2017].

[17] Unity Technologies (2017). MonoBehaviour.Start() [verkko]. Saatavilla: <https://docs.unity3d.com/ScriptReference/MonoBehaviour.Start.html>. [Viitattu 4.9.2017].

[18] Booth M. (2009). The AI Systems of Left 4 Dead (pdf) Saatavilla: <www.valvesoftware.com/publications/2009/ai_systems_of_l4d_mike_booth.pdf [Viitattu 24.8.2017].

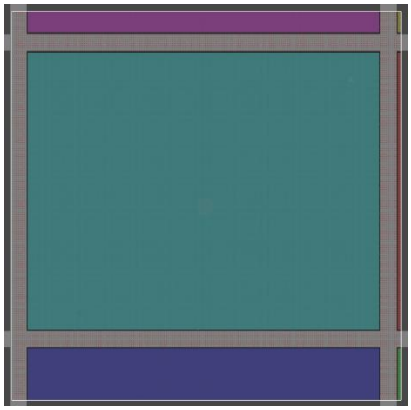
[19] Unity Technologies (2017). Profiler window [verkko]. Saatavilla: <https://docs.unity3d.com/Manual/ProfilerWindow.html>. [Viitattu 8.9.2017].

[20] Unity Technologies (2017). Unity Roadmap [verkko]. Saatavilla: <https://unity3d.com/unity/roadmap>. [Viitattu 24.8.2017].

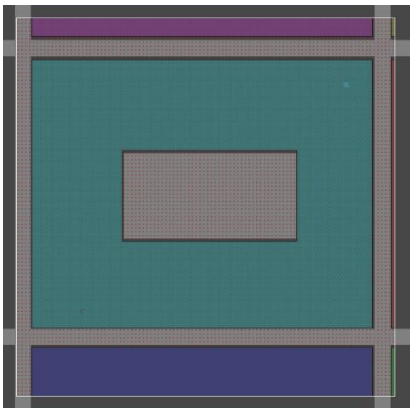
[21] Winter B. (2017). TextMesh Pro Joins Unity. [verkko]. Unity Technologies. Saatavilla: <https://blogs.unity3d.com/2017/03/20/textmesh-pro-joins-unity/>. [Viitattu 24.8.2017].

[22] Anhalt, J., Kring, A., Sturtevant, N. (2011). AI Navigation: It's Not a Solved Problem – Yet, GDC 2011 Saatavilla: <http://www.gdcvault.com/play/1014514/AI-Navigation-It-s-Not> [Viitattu 16.4.2017].

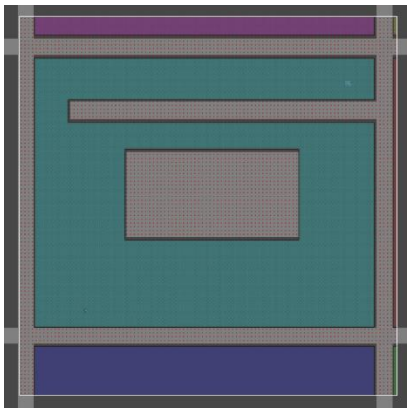
LIITE 1. Testiverkostot



Esteiden asettelu testeissä 1,4,7.



Esteiden asettelu testeissä 2,5,8.



Esteiden asettelu testeissä 3,6,9.

LIITE 2. Reitinhakuliitännäisen GitHub-sivu

https://github.com/wawethewaras/Astar-Pathfinding

LIITE 3. Esimerkki reitinhaun käytöstä

```
using UnityEngine;

[RequireComponent(typeof(CountPath))]
public class Seeker : MonoBehaviour {
    private CountPath counter;

    void Start() {
        counter = GetComponent<CountPath>();
    }

    void Update() {
        if (Input.GetMouseButtonDown(0)) {
            counter.FindPath(transform,
Camera.main.ScreenToWorldPoint(Input.mousePosition));
        }
    }
}
```