Tampereen yliopisto

Arttu Leppäaho

REITINETSINTÄ KOLMIULOTTEISISSA PELIYMPÄRISTÖISSÄ

Kandidaatintyö
Informaatioteknologian ja viestinnän tiedekunta
Toukokuu 2022

TIIVISTELMÄ

Arttu Leppäaho: Reitinetsintä kolmiulotteisissa peliympäristöissä
Kandidaatintutkielma
Tampereen yliopisto
Tietotekniikan tutkinto-ohjelma
Toukokuu 2022

Reitinetsintä on yleinen haaste, joka pitää ratkaista monenlaisissa videopeleissä. Sen ratkaiseminen on usein perusvaatimuksena pelihahmojen toiminnalle, sillä hahmojen täytyy kyetä suunnistamaan luontevasti pelimaailman läpi. Tämän tutkielman tavoitteena on kartoittaa kolmiulotteisten peliympäristöjen reitinetsintään soveltuvia tekniikoita sekä arvioida niiden ongelmia ja mahdollisuuksia.

Tutkielmassa perehdytään ensin yleisimpiin peleissä käytettyihin reitinetsintäalgoritmeihin ja sitten tapoihin, joilla näitä algoritmeja voidaan soveltaa peliympäristöihin. Yleisiksi reitinetsintäalgoritmeiksi osoittautuvat A*-algoritmi, geneettiset algoritmit ja muurahaisyhdyskuntaoptimointi. A*-algoritmi on melko yksinkertainen ja tehokas reitinetsintäalgoritmi, mutta sen toimintakyky heikkenee, jos ympäristössä tapahtuu usein muutoksia. Geneettiset algoritmit ja muurahaisyhdyskuntaoptimointi sen sijaan soveltuvat hyvin myös muuttuviin ympäristöihin.

Havaitaan, että A*-algoritmista on tehty erilaisia muunnoksia, jotka tehostavat algoritmin tiettyjä osia, mutta heikentävät toisia. Tällaisen muunnoksen käytöstä voi olla hyötyä, jos tehostetut ominaisuudet ovat pelille tärkeitä. Heikentyneet ominaisuudet täytyy kuitenkin ottaa myös huomioon ja arvioida, miten muunnos vaikuttaisi kokonaisuutena peliin.

Jotta reitinetsintäalgoritmeja voidaan käyttää pelissä, peliympäristön pohjalta täytyy ensin muodostaa graafi. Tutkielmassa havaitaan, että tämän graafin toteutustavan valinta on tärkeä reitinetsintäjärjestelmän ominaisuuksiin vaikuttava tekijä. Kirjallisuudesta löydetään kolme yleisesti käytettyä graafimallia, joista jokainen soveltuu paremmin tietynlaisiin käyttökohteisiin ja huonommin toisiin. Näistä kolmesta mallista soveltuvimmaksi kolmiulotteisiin ympäristöihin osoittautuu navigaatioverkko.

Navigaatioverkot mallintavat pintoja, joilla pelihahmot voivat kävellä, ja ne mahdollistavat kahta muuta tarkasteltua graafimallia tarkemman ja joustavamman reitinetsinnän. Verkon voi luoda pitkälti automaattisesti peliympäristön pohjalta, mikä säästää aikaa, mutta manuaalinen muokkaaminen on myös mahdollista. Näin pelisuunnittelijat voivat ensin luoda karkean navigaatioverkon automaattisesti ja sitten hienosäätää sitä pelin tarpeisiin.

Tutkielman yhteydessä toteutetaan myös yksinkertainen navigaatioverkkoja hyödyntävä reitinetsintäjärjestelmä, joka havainnollistaa navigaatioverkkojen toimintaa. Järjestelmälle voidaan antaa mielivaltainen kolmiulotteinen malli navigaatioverkoksi, joten reitinetsintää voidaan tarkastella monenlaisissa tilanteissa. Järjestelmää voitaisiin myös helposti laajentaa oikeaksi peliksi, sillä se toteutetaan samoilla työkaluilla, joilla monet pelit on tehty.

Avainsanat: reitinetsintä, navigaatioverkot, pelit, algoritmit

SISÄLLYSLUETTELO

LYHENTEET JA MERKINNÄT

peli	videopeli
pelihahmo	videopelissä esiintyvä simuloitu hahmo
maasto	videopelissä esiintyvä virtuaalinen maasto
A*	A*-reitinetsintäalgoritmi
verkko	navigaatioverkko, reitinetsinnässä käytetty tietorakenne
3D	kolmiulotteinen, engl. three-dimensional
OpenGL	Open Graphics Library -grafiikkarajapinta
OBJ-tiedosto	Wavefront OBJ -3D-mallitiedosto

1. JOHDANTO

Videopeleihin ovat aina kuuluneet olennaisena osana pelihahmot, jotka liikkuvat näytöllä. Jo monet ensimmäiset videopelit, kuten Pac-Man [1] ja Space Invaders [2], sisälsivät yksinkertaisia tietokoneen ohjaamia hahmoja, jotka tekivät pelin kulusta mielenkiintoisempaa ja haastavampaa. Aluksi yksinkertaiset hahmot riittivät, sillä pelien muutkin osat olivat vielä hyvin suoraviivaisia. Kuitenkin laskentatehon parantuessa ja muistin lisääntyessä peleistä haluttiin entistä realistisempia ja monimutkaisempia, joten pelihahmojenkin täytyi kehittyä.

Pelihahmolla halutaan usein kuvata jotain realistisesti käyttäytyvää olentoa, kuten ihmistä tai eläintä. Tästä seuraa, että eräs pelihahmon tärkeimmistä ominaisuuksista on kyky liikkua paikasta toiseen elävän olion tavoin. Pac-Manin haamua [1], Need for Speedin autonkuljettajaa [3] ja Call of Dutyn sotilasta [4] kaikkia yhdistää pelimaailmassa liikkuminen, vaikka hahmot ovat muuten hyvin erilaisia. Monet peli-ideat eivät toimisi lainkaan, jos pelihahmoja ei voitaisi liikutella, sillä esimerkiksi jotkin autopelit [3] ja sotapelit [4] perustuvat suureksi osaksi tietokoneen liikuttamien vastustajien kanssa pelaamiseen.

Pelihahmon realistisessa liikkumisessa on kuitenkin haasteita: myös oikean maailman elollisten olentojen liikkuminen on monimutkainen prosessi, vaikka sitä pidetään usein perusominaisuutena. Jotkin haasteet voidaan onneksi ohittaa, sillä virtuaalisia hahmoja ei tarvitse simuloida täysin reaalimaailman mukaisesti: esimerkiksi hahmon jalkojen liikettä ei tarvitse ottaa huomioon pelin fysiikkasimulaatioissa. Sen sijaan voidaan luoda illuusio kävelystä käyttämällä valmista kävelyanimaatiota ja liu'uttamalla hahmoa maata pitkin sopivalla nopeudella.

Ympäristön navigointi on kuitenkin ongelma, jota ei voida sivuuttaa näin helposti. Jotta pelihahmot kykenisivät siirtymään pelimaailmassa paikasta toiseen, niiden tulee jollain tavalla kyetä löytämään reitti ympäristössä olevien esteiden ohi kohteeseensa. Tähän on kehitetty erilaisia tekniikoita ja järjestelmiä, joiden juuret ovat matemaattisessa graafi- ja reitinetsintäteoriassa.

Tämän tutkielman tarkoituksena on selvittää, millaisia nämä peleissä käytettävät reitinetsintätekniikat ja -järjestelmät ovat, miten niitä voidaan hyödyntää kolmiulotteisissa peliympäristöissä ja millaisia ongelmia niiden käytössä voi esiintyä. Suurta osaa pelien

reitinetsintätekniikoista voidaan käyttää sekä kaksi- että kolmiulotteisissa ympäristöissä, joten tutkielmassa käsitellään aluksi myös yleisempää reitinetsintäteoriaa, joka ei liity pelkästään kolmiulotteisiin ympäristöihin.

Peleihin ei ole yhtä yleispätevää reitinetsintäratkaisua, sillä erilaisissa peleissä hahmot ja ympäristöt toimivat eri lailla ja siten vaativat reitinetsintäjärjestelmiltä eri asioita. Tämän vuoksi erilaiset reitinetsintämahdollisuudet on hyvä tuntea peliä suunniteltaessa ja toteutettaessa, jotta olemassaolevista reitinetsintäjärjestelmistä voidaan valita peliin sopivin vaihtoehto ja tarvittaessa muokata sitä vastaamaan pelin vaatimuksia.

Luvussa 2 perehdytään ensin reitinetsintäalgoritmien toimintaan ja niiden käyttöön peleissä. Sen jälkeen luvussa 3 tutkitaan ja vertaillaan erilaisia tapoja mallintaa peliympäristöä reitinetsinnän käyttöön. Näistä mallinnustavoista soveltuvin kolmiulotteisiin ympäristöihin on navigaatioverkko, jota tutkitaan tarkemmin luvussa 4. Lopuksi luvussa 5 esitellään esimerkkitoteutus yksinkertaisesta navigaatioverkkoa käyttävästä reitinetsintäjärjestelmästä ja luvussa 6 tehdään yhteenveto tutkielman tuloksista.

2. TUTKIMUSMENETELMÄ

Tutkimukseen valittiin mukaan reitinetsintäalgoritmeista, pelien reitinetsinnästä ja navigaatioverkoista kertovia lähteitä. Lähteiden laatua arvioitiin tarkistamalla, onko lähde vertaisarvioitu, onko siihen viitattu muissa lähteissä ja vastaavatko sen tiedot muiden lähteiden tietoja. Lähdehakuja tehtiin Andoriin, SpringerLinkiin ja ACM Digital Libraryyn. Lisäksi monet lähteet löytyivät tutkimalla aiemmin löydettyjen lähteiden lähdeluetteloita ja esimerkiksi muita osia kirja- tai julkaisusarjasta, johon aiemmin löydetty lähde kuului.

Tietokannoista etsittiin lähteitä muun muassa hauilla *pathfinding*, *pathfinding AND game** ja *”navigation mesh”,* joilla löydettiin ensimmäiset lähteet. Lupaavilta vaikuttavia hakutuloksia merkittiin muistiin esimerkiksi selaimen kirjanmerkeillä ja Andorin suosikit-toiminnolla. Hakutuloksista karsittiin pois lähteitä, jotka eivät käsitelleet reitinetsintää joko pelien tai muiden virtuaalisten ympäristöjen näkökulmasta. Lisäksi lähteitä jätettiin pois, jos ne keskittyivät liikaa jonkin tietyn reitinetsintätoteutuksen yksityiskohtiin, eivätkä antaneet tutkielmaan sopivaa yleistä tietoa aiheesta.

Seuraavaksi lähteitä etsittiin lisää tutkimalla aluksi löydettyjen lähteiden lähdeluetteloja, joista havaittiin joitain toistuvia lähteitä ja kirjoittajien nimiä. Näitä lähteitä ja kirjoittajia tutkittiin tarkemmin, ja löydettiin lisää tutkimuksessa hyödynnettävää kirjallisuutta. Myös tutkimukseen mukaan otettujen kirja- ja julkaisusarjojen eri osia tutkittiin uusien lähteiden löytämiseksi. Lopuksi, kun tutkielman kirjoittaminen oli jo edistynyt jonkin verran, arvioitiin, mistä aiheista lähteitä vielä puuttuu ja etsittiin loput tutkimuksen tarvitsemat lähteet yhdistelemällä edellä mainittuja tiedonhakutapoja.

3. REITINETSINTÄALGORITMIT

Reitinetsintäalgoritmit etsivät reitin kahden pisteen välille ympäristössä, joka koostuu vapaista poluista ja esteistä. Niillä on monia käyttökohteita, kuten pelit [5][6][7], robotit [5][7], tietoliikenneverkot [8], GPS-järjestelmät [5] ja erilaiset simulaatiot [6][8]. Peleissä reitinetsintäalgoritmeja käytetään yleisesti pelihahmojen liikkumisen suunnitteluun [5][6][9][10]. Reitinetsintäalgoritmit on luotu toimimaan graafeiksi kutsutuissa tietorakenteissa [8][11], joten jos niitä halutaan käyttää peliympäristöissä, ympäristön pohjalta täytyy ensin luoda graafi.

Graafi rakentuu solmuista ja niiden välisistä kaarista. Kaarilla voi olla painokertoimia, joilla voidaan kuvata esimerkiksi kaarten pituuksia. Reitinetsinnässä graafin solmut kuvaavat paikkoja ja kaaret polkuja niiden välillä. Jotkin reitinetsintäalgoritmit ottavat kaarten painokertoimet huomioon reitin valinnassa, ja näin voidaan esimerkiksi suosia lyhyitä reittejä. [8]

Yleiset reitinetsintäalgoritmit voidaan jakaa kahteen kategoriaan: heuristisiin algoritmeihin ja metaheuristisiin algoritmeihin. Tutkielmassa käsitellyistä algoritmeista heuristisia algoritmeja ovat Dijkstran algoritmi sekä A*-algoritmi ja metaheuristisia algoritmeja ovat geneettiset algoritmit sekä muurahaisyhdyskuntaoptimointi. Heuristiset algoritmit ovat yksinkertaisempia ja soveltuvat hyvin useimpiin reitinetsintätilanteisiin, mutta metaheuristiset algoritmit voivat toimia niitä tehokkaammin ja soveltuvat myös joihinkin tilanteisiin, joissa heuristiset algoritmit voivat pettää. [5]

3.1 Dijkstran algoritmi

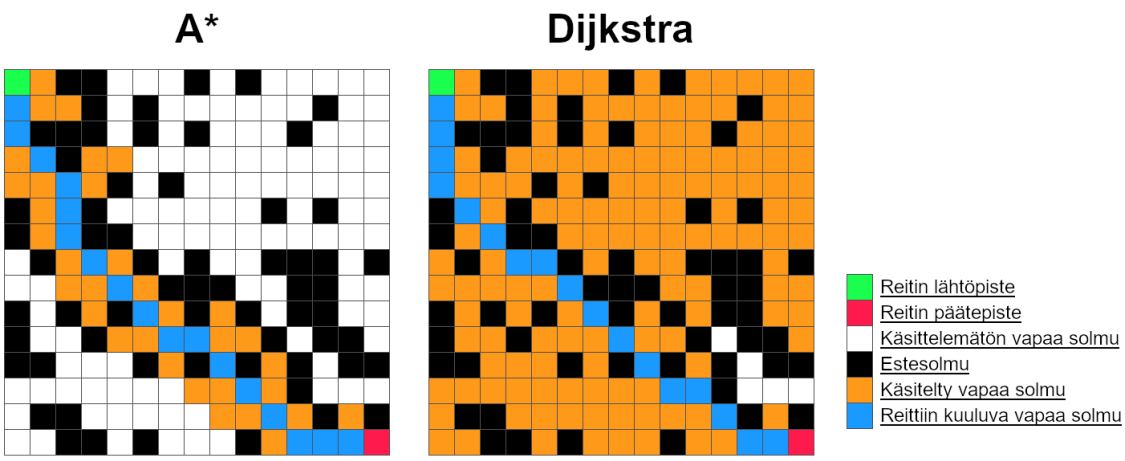
Dijkstran algoritmi on yksinkertainen reitinetsintäalgoritmi, jonka keksi Edsger W. Dijkstra vuonna 1959 [11]. Se löytää reitin levittäytymällä graafissa aloitussolmusta kaaria pitkin ympäröiviin solmuihin ja niistä edelleen eteenpäin, kunnes kohdesolmu löytyy. Käsitellyille solmuille merkitään paras aloitussolmua päin vievä kaari, jolloin kohdesolmun löytyessä siitä voidaan helposti kerätä lyhin reitti aloitussolmuun. [11]

Dijkstran algoritmia käytetään edelleen [6][9], mutta peleissä muut, uudemmat algoritmit ovat usein sitä tehokkaampia [5][6]. Yksi tällainen tehokkaampi algoritmi on A*-algoritmi, jota käsitellään luvussa 3.2.

3.2 A*-algoritmi

A*-algoritmi on tehokkuutensa ja yksinkertaisuutensa vuoksi yksi käytetyimmistä reitinetsintäalgoritmeista peleissä [5]. Se perustuu Dijkstran algoritmiin, jota se tehostaa heuristiikkafunktion avulla [6][8]. Tämä heuristiikkafunktio arvioi solmukohtaisesti minimimatkaa, jonka algoritmin tulee vähintään kulkea päästäkseen tämän solmun läpi kulkevaa reittiä pitkin kohteeseen. Jotta minimimatka saataisiin määritettyä helposti, se lasketaan usein solujen välille muodostettujen suorien viivojen pituuksista. [6][8] Kuitenkin jos tarkempi minimimatkan määrittäminen on mahdollista, se voi tehostaa algoritmin toimintaa lisää [6].

Kun minimimatka-arviot solmuille tiedetään, A*-algoritmi voi priorisoida reitinetsinnän aikana solmuja, joiden läpi tämä minimimatka olisi lyhin. Tämän priorisoinnin ansiosta A* usein löytää reitin Dijkstran algoritmia nopeammin [8]. A*-algoritmin ja Dijkstran algoritmin toimintaa voidaan vertailla kuvan 1 avulla.



Kuva 1. *A*-algoritmin ja Dijkstran algoritmin käsittelemät solmut saman reitinetsintätehtävän jälkeen, perustuu lähteeseen [12].*

Kuvan 1 kaksi ruudukkoa kuvaavat ympäristöä, jossa halutaan löytää reitti vasemman yläkulman vihreästä ruudusta oikean alakulman punaiseen ruutuun. Esteruudut, joiden läpi ei voida kulkea, on merkitty mustalla. Loput ruudut ovat vapaita ruutuja, joiden läpi kulkeminen on mahdollista. A*-algoritmin reitinetsintäprosessi on kuvattu vasemmanpuoleisessa ruudukossa ja Dijkstran algoritmin prosessi oikeanpuoleisessa ruudukossa. Lopullinen löydetty reitti on merkitty sinisellä ja muut algoritmien käsittelemät ruudut oranssilla. Kuvasta 1 voidaan havaita, että molemmat algoritmit ovat löytäneet lyhimmän mahdollisen reitin: vaikka reitit ovat hieman eri muotoisia, ne ovat molemmat 19 ruudun pituisia. Oranssien ruutujen eli käsiteltyjen ruutujen määrä kuitenkin eroaa merkittävästi, ja tästä nähdään A*:n etu.

Kun A*-algoritmin minimimatkojen laskemiseen käytetään suoria viivoja aiemmin mainitulla tavalla, A* yrittää ensin löytää reitin kulkemalla suoraan kohdetta päin (kuva 1). Tämä strategia toimii hyvin, ellei ympäristö ole erityisen sokkeloinen, sillä A* voi näin ylittää avoimet alueet nopeasti. Kohteen suuntaa priorisoimaton Dijkstran algoritmi voi jäädä tutkimaan muihin suuntiin lähteviä reittejä turhaan, vaikka kohteeseen pääsisi helposti vain kulkemalla suoraan sitä kohti. Tämä voidaan havaita kuvasta 1, jossa A*-algoritmi on käsitellyt melko suoran reitin etsinnässä selvästi vähemmän solmuja kuin Dijkstran algoritmi.

3.3 A*-algoritmin muunnokset

A*-algoritmia on yritetty tehostaa lisää eri tavoin, ja siitä on luotu erilaisia muunnoksia pelien käyttöön [5][6][9][10]. Esimerkiksi Rabinin ja Sturtevantin [9] maalinrajausmenetelmä laskee graafin jokaisen solmun jokaiselle kaarelle suorakaiteen, joka rajaa kaikki muut solmut, joihin tätä kaarta pitkin pääsee nopeiten. Menetelmä nopeuttaa A*-algoritmin toimintaa merkittävästi, mutta rajaavat suorakaiteet vaativat ylimääräistä muistia ja niiden laskemiseen voi kulua jopa useita tunteja [9].

Suorakaiteiden laskemiseen kuluvasta ajasta syntyvä haitta voidaan välttää, jos suorakaiteet lasketaan etukäteen ja ladataan pelin aikana valmiista tiedostosta [9]. Tämä kuitenkin edellyttää, ettei peliympäristö muutu merkittävästi pelin aikana, sillä muutosten jälkeen aiemmin lasketut suorakaiteet eivät välttämättä enää vastaa nykyistä ympäristöä. Menetelmä ei siis sovellu muuttuviin ympäristöihin, sillä suorakaiteita ei ehditä laskemaan uudestaan pelin aikana kohtuullisessa ajassa [9]. Tästä voidaan havaita, että A*:n muunnokset saattavat toimia hyvin tietynlaisissa peleissä, mutta huonosti toisissa. Tämä on luultavasti yksi syy, miksi A*:n muunnokset eivät ole syrjäyttäneet alkuperäistä A*-algoritmia.

3.4 Metaheuristiset algoritmit

A* soveltuu hyvin muuttumattomiin ympäristöihin, mutta muuttuvat ympäristöt voivat tehdä siitä liian hitaan tai tehottoman [5][6]. Esimerkiksi jos ympäristö muuttuu reitinetsinnän aikana, A*-algoritmi voidaan joutua aloittamaan alusta [6]. Reaaliaikaisiin muutoksiin soveltuvampia algoritmeja ovat geneettiset algoritmit ja muurahaisyhdyskuntaoptimointi, jotka perustuvat luonnossa esiintyviin biologisiin järjestelmiin [5]. Nämä algoritmit kuuluvat metaheuristisiin algoritmeihin.

Geneettiset algoritmit perustuvat evoluutioon, ja niitä voidaan käyttää reitinetsinnän lisäksi myös monenlaisiin muihin optimointi- ja suunnitteluongelmiin [5][13]. Ne simuloivat geenien periytymistä sukupolvelta toiselle ja siten hyödyntävät luonnonvalintaa optimaalisen ratkaisun löytämiseksi [13]. Kun optimaaliseksi ratkaisuksi määritellään lyhin reitti kahden pisteen välillä, geneettiset algoritmit voivat ratkaista myös reitinetsintäongelmia [5].

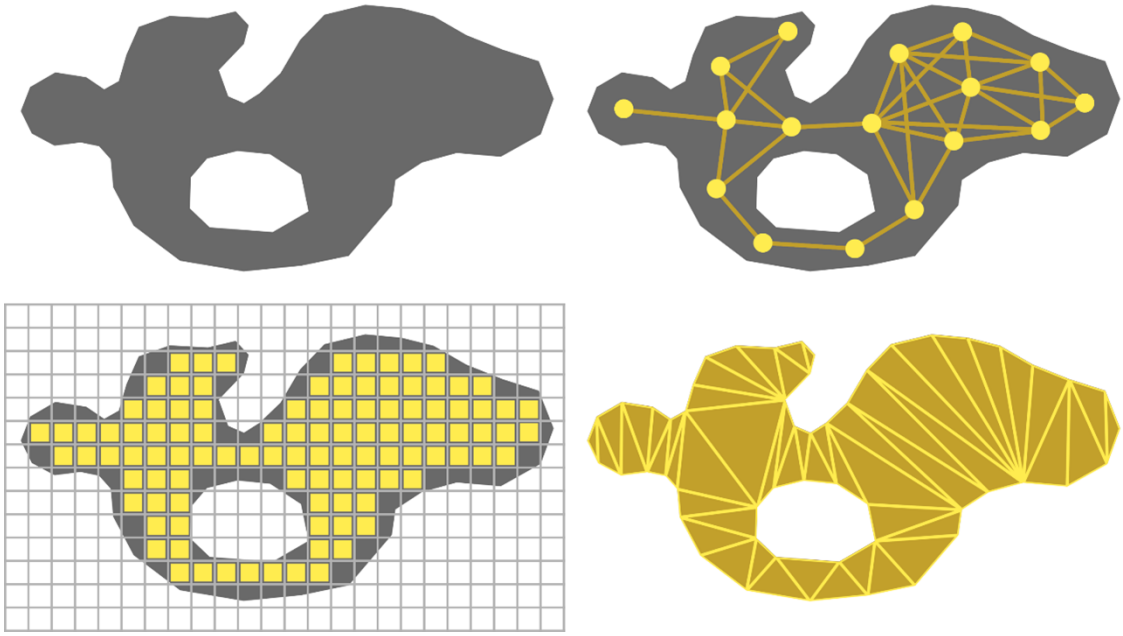
Muurahaisyhdyskuntaoptimointi taas perustuu muurahaisten kykyyn viestiä toisilleen feromoneilla. Oikeat muurahaiset merkitsevät käyttämänsä hyvät polut feromoneilla, jolloin muut muurahaiset voivat myöhemmin haistaa feromonit ja seurata itsekin samaa polkua. Kun nämä toisetkin muurahaiset merkitsevät polun, sen haju vahvistuu ja polku houkuttelee entistä useampaa muurahaista. Lopulta kaikki muurahaiset kerääntyvät parhaille poluille, sillä niiden varrella on eniten feromoneja. [13] Tätä samaa periaatetta voidaan hyödyntää myös virtuaalisissa ympäristöissä, jolloin simuloidut näkymättömät muurahaiset merkitsevät parhaan reitin pisteiden välille [5].

Geneettiset algoritmit ja muurahaisyhdyskuntaoptimointi käyttävät iterointia ratkaisun löytämiseen. Ne eivät välttämättä löydä lyhintä reittiä heti, vaan ne saattavat ensin löytää pidemmän reitin kohteeseen ja sitten optimoida sitä lyhyemmäksi iteroimalla lisää. Iterointi voidaan keskeyttää esimerkiksi aikarajan tai tietyn iterointikierrosmäärän jälkeen: näin käyttökelpoinen reitti voidaan löytää nopeammin, jos reitin ei tarvitse olla optimaalinen. [13]

Rafiqin ja muiden [5] tutkimuksen mukaan näitä kahta metaheuristista algoritmia käytettiin vuonna 2018 pelien reitinetsinnässä vähemmän kuin A*-algoritmia, mutta metaheurististen algoritmien käyttö on lisääntynyt viime vuosina [5]. On siis todennäköistä, että niillä on tulevaisuudessa entistä merkittävämpi asema pelien reitinetsinnässä.

4. PELIYMPÄRISTÖN MALLINTAMINEN REITINETSINTÄÄ VARTEN

Jotta reitinetsintäalgoritmeja voidaan hyödyntää peliympäristössä, ympäristöä täytyy mallintaa jonkinlaisena graafina. Tällöin peliympäristöstä säilytetään pelin muistissa kahta versiota: alkuperäistä pelaajalle näytettävää versiota ja reitinetsinnän käyttämää graafikuvausta. Graafikuvaus voidaan muodostaa eri tavoin, kuten reittipistegraafina (engl. waypoint graph), säännöllisenä ruudukkona tai navigaatioverkkona [14][15]. Nämä tavat on esitetty kuvassa 2, jossa alkuperäinen pelimaasto on väritetty harmaalla ja graafikuvaukset keltaisella.



Kuva 2. *Peliympäristö ja sen kuvaus reittipistegraafilla, säännöllisellä ruudukolla ja navigaatioverkolla, perustuu lähteeseen [14].*

Kuvassa 2 alkuperäinen pelimaasto on esitetty ylhäällä vasemmalla, reittipistegraafi ylhäällä oikealla, säännöllinen ruudukko alhaalla vasemmalla ja navigaatioverkko alhaalla oikealla. Harmaa pelimaasto on kuvattu ilmakuvana ylhäältä päin ja hahmojen oletetaan pystyvän kävelemään sen pinnalla vapaasti. Keltaisella väritetyt osat kuvaavat graafin peittämää aluetta, jossa reitinetsintä tapahtuu. Reittipistegraafissa pallot esittävät solmuja ja viivat kaaria. Säännöllisessä ruudukossa keltaiset ruudut esittävät solmuja ja vierekkäisten solmuruutujen välillä on aina kaari. Navigaatioverkossa kolmiot esittävät solmuja ja saman sivun jakavien kolmioiden välillä on aina kaari.

Nämä kolme graafimallia ovat peleissä yleisesti käytettyjä [14][15]. Parhaan mallin valinta riippuu pelin tyypistä ja vaatimuksista, sillä kaikilla kolmella mallilla on omat vahvuutensa ja heikkoutensa. Tietyt ominaisuudet voi olla helppo toteuttaa tietyllä mallilla, mutta sama malli voi tehdä jostain toisesta ominaisuudesta haastavan. Lisäksi eri operaatioiden tehokkuudet vaihtelevat mallien välillä, joten mallin valinnassa kannattaa huomioida myös pelin tarvitsemien reitinetsintäoperaatioiden luonne ja määrä. [14][15]

Sturtevant mainitsee julkaisussaan [14] kuusi ominaisuutta, joita on hyvä tarkastella graafimallien vertailussa. Koska kaksi näistä ominaisuuksista liittyvät samaan aihepiiriin, ne voidaan tiivistää viiteen ominaisuuteen, jotka ovat reitinetsinnän tehokkuus, muistinkäyttö, paikallistamisen helppous, reitin laatu ja dynaamisuus. Muistinkäyttö tarkoittaa mallin käyttämän muistin määrää. Paikallistaminen tarkoittaa peliympäristön sijainnin muuttamista navigaatioavaruuden sijainniksi: esimerkiksi lähimmän navigaatiosolmun löytämistä jostain pelimaailman pisteestä. Reitin laadun arviointi riippuu pelin vaatimuksista, mutta laadukas reitti voi olla esimerkiksi mahdollisimman lyhyt ja sulavan muotoinen. Dynaamisuus taas tarkoittaa mallin muokkaamisen tehokkuutta pelin aikana. [14] Jotkut mallit esimerkiksi vaativat paljon ennalta laskettua dataa, joten niitä ei ole mahdollista muokata nopeasti [9].

4.1 Ympäristön mallintaminen reittipistegraafilla

Reittipistegraafi koostuu reittipisteiksi kutsutuista solmuista ja niiden välillä kulkevista kaarista. Reittipisteet kuvaavat ympäristön pisteitä, joiden kohdalla hahmot voivat olla ja kaaret kuvaavat pisteiden välisiä polkuja. Reittipistegraafi on melko helppo toteuttaa, ja reitinetsintäalgoritmit toimivat sen avulla tehokkaasti, sillä solmuja on vähemmän kuin muissa graafimalleissa. Reittipisteiden pieni määrä myös pitää muistinkäytön pienenä. [14]

Näin yksinkertainen ympäristön kuvaus kuitenkin harvoin riittää. Jos ympäristö on pelkistetty vain pisteiksi ja kaariksi, ympäristössä kulkevat hahmot voivat liikkua turvallisesti vain kaaria pitkin [14][15], sillä hahmot eivät tiedä, mitä kaarien ulkopuolella on. Jos ne oikaisisivat joskus kaarten ulkopuolelle, ne saattaisivat törmätä johonkin tai pudota kuoppaan [14]. Kaarten seuraaminen voi olla merkittävä rajoite hahmojen liikkumiselle, mikä voidaan havaita kuvasta 2: suuri osa harmaasta peliympäristöstä on reittipistegraafin solmujen ja kaarten ulkopuolella ja siten poissa pelihahmojen käytöstä.

Lisäksi kaaria pitkin kulkevat hahmot saattavat näyttää epärealistisilta ja niiden käytös voi olla liian joustamatonta tai yksinkertaista pelin tarpeisiin. Erilaiset pelitilanteet voivat

aiheuttaa hahmoille ongelmia: esimerkiksi jos kaksi hahmoa kulkevat samaa kaarta pitkin eri suuntiin, hahmot eivät kykene ohittamaan toisiaan kaaren keskellä poistumatta kaarelta. Jos taas jokin työntää hahmon pois kaarelta, hahmolla voi olla vaikeuksia löytää uusi kaari, jota seurata [14]. Reittipistegraafi joudutaan myös usein rakentamaan manuaalisesti, mihin voi kulua aikaa [14][15].

Muut graafimallit ratkaisevat nämä ongelmat muun muassa mahdollistamalla joustavamman liikkeen solmujen ympärillä, ja reittipistegraafien käyttö onkin vähentynyt [14][15]. Reittipistegraafit ovat siitä huolimatta toimiva reitinetsintätapa, ja ne voivat soveltua hyvin esimerkiksi peleihin, joissa hahmojen ei tarvitse liikkua monimutkaisesti.

4.2 Ympäristön mallintaminen säännöllisellä ruudukolla

Reittipistegraafia joustavampi tapa mallintaa peliympäristöä on muodostaa peliympäristöstä säännöllinen kaksiulotteinen ruudukko [15]. Tällainen ruudukko on graafi, jonka solmut kuvaavat ruutuja ja kaaret viereisten ruutujen välisiä yhteyksiä. Ruudut merkataan peliympäristön esteiden mukaan siten, että jos ruudun alueella on yksikin este, koko ruutu merkataan esteeksi [14]. Näin ruudut, joita ei ole merkattu esteiksi ovat varmasti esteettömiä, ja pelihahmot voivat liikkua niiden sisällä vapaasti. Ruudukko myös peittää koko peliympäristön, joten hahmot voivat navigoida ympäristössä vapaammin eikä niiden tarvitse seurata ennalta määriteltyjä viivamaisia kaaria.

Säännölliset ruudukot toimivat hyvin reitinetsinnän apuna peleissä, joissa koko pelimaailma rakentuu muutenkin ruuduista tai palikoista. Lisäksi paikallistaminen ruudukon sisällä ja ruudukon muokkaaminen onnistuvat tehokkaasti [14], joten säännöllinen ruudukko voi olla hyvä valinta muuttuviin ympäristöihin.

Monimutkaiset ja kolmiulotteiset ympäristöt kuitenkin aiheuttavat säännöllisille ruudukoille ongelmia. Kuvasta 2 voidaan havaita, että ruudukko ei seuraa tarkasti harmaan peliympäristön muotoja, joten pelihahmot eivät välttämättä voi navigoida tarkasti ympäristön muotojen mukaan. Lisäksi jos ruutuja on paljon, ne voivat viedä paljon muistia [15]. Ruutujen määrää on hankala vähentää, sillä ruudukon täytyy toimiakseen peittää koko reitinetsintäympäristö ja mitä suurempia ruuduista tehdään, sitä epätarkemmin ne mallintavat ympäristön muotoja.

Kolmiulotteiset ympäristöt taas ovat ongelmallisia, sillä jos ruudukko on kaksiulotteinen, sillä ei voida mallintaa pystysuunnassa päällekkäisiä alueita. Tämä johtuu siitä, että päällekkäiset alueet osuisivat ruudukossa samaan kohtaan, jolloin ruudukko voi ottaa huomioon vain toisen alueen.

Ruudukko voitaisiin toteuttaa myös kolmiulotteisena päällekkäisten alueiden mallintamiseksi, jolloin ruudut olisivat kuutioita. Se kuitenkin kasvattaisi ruutujen määrää ja sitä kautta ruudukon muistinkäyttöä moninkertaisesti: kun ruudukkoon lisätään kolmas ulottuvuus, uusi ruutujen määrä on vanha kaksiulotteisten ruutujen määrä kerrottuna ruudukon korkeudella. Lisäksi ruudukon epätarkkuus lisääntyy, sillä kolmiulotteinen ruudukko joutuu pyöristämään ympäristön muotoja myös pystysuunnassa.

4.3 Ympäristön mallintaminen navigaatioverkolla

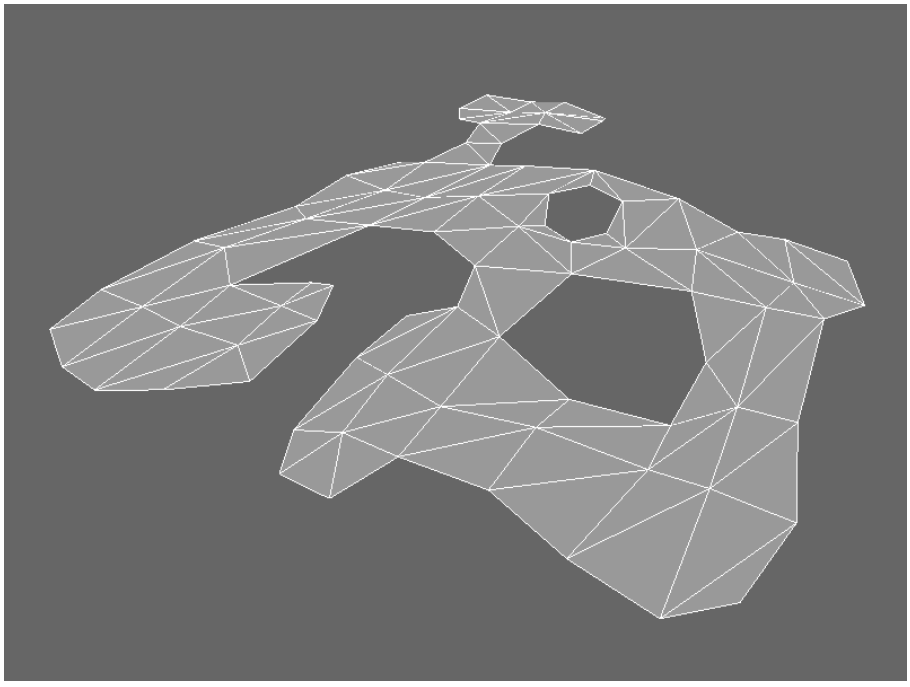
Kolmas yleinen tapa mallintaa peliympäristöä graafina on muodostaa siitä navigaatioverkko. Navigaatioverkot koostuvat yhdistetyistä kaksiulotteisista soluista ruudukoiden tapaan, mutta niissä solujen ei tarvitse olla säännöllisen kokoisia tai nelikulmaisen ruudun muotoisia. Lisäksi solujen ei tarvitse olla geometrisesti samalla tasolla, vaan verkon solut voivat yhdessä muodostaa kolmiulotteisen rakennelman, jossa on useampia kerroksia. [16]

Navigaatioverkko on kahta muuta mainittua graafimallia vaikeampi toteuttaa [14], mutta se ratkaisee monia säännöllisen ruudukon ongelmia: esimerkiksi muistinkäyttöä voidaan vähentää käyttämällä yksinkertaisilla alueilla pientä määrää suuria soluja, jolloin ympäristön peittämiseen tarvitaan vähemmän soluja. Lisäksi päällekkäisten alueiden mallintaminen onnistuu kerrosten avulla ja ympäristön muotoja voidaan mallintaa tarkemmin kuin ruudukolla (kuva 2). Navigaatioverkkoja käsitellään lisää luvussa 5.

5. NAVIGAATIOVERKOT

Navigaatioverkko (engl. navigation mesh) on tietorakenne, jolla voidaan mallintaa kaksi- tai kolmiulotteista ympäristöä reitinetsintää varten. Sen alkuperäisenä kehittäjänä pidetään Greg Snookia [15][17], joka esitteli idean vuonna 2000 julkaisussaan Simplified 3D Movement and Pathfinding Using Navigation Meshes [16]. Navigaatioverkko koostuu litteistä yhdistetyistä soluista, jotka kuvaavat maastoa, jossa pelihahmot voivat liikkua [16].

Navigaatioverkot perustuvat havaintoon, että pelihahmot yleensä liikkuvat vain maata pitkin [16]. Tällöin reitinetsintäjärjestelmien tarvitsee tarkastella vain peliympäristön maastoa ja sen päällä olevia mahdollisia esteitä. Pelimaasto voidaan peittää litteistä soluista koostuvalla navigaatioverkolla niin, että verkko ylettyy maastossa kaikille pinnoille, joiden päällä hahmot voivat liikkua. Kaikenlaiset esteet jätetään verkon ulkopuolelle, jolloin verkon reunat määrittävät käveltävät alueet. Kuvassa 3 on esimerkki navigaatioverkosta.



Kuva 3. *Kolmiulotteinen kolmion muotoisista soluista koostuva navigaatioverkko.*

Navigaatioverkon solut toteutetaan usein monikulmioina [15][16][17][18]. Navigaatioverkko on hyvä rakentaa kolmioista, sillä kolmioita on helppo käsitellä matemaattisesti [17][18]. Tämä johtuu muun muassa siitä, että kolmiot ovat aina konvekseja, eli kolmion kulmien välille piirretyt janat eivät voi koskaan leikata toisiaan [17]. Tällöin kolmion pinnalla olevien kahden pisteen välille voidaan aina muodostaa

suoran viivan muotoinen reitti. Kaikki monikulmiot voidaan myös jakaa kolmioihin, joten kolmion muotoisilla soluilla voidaan mallintaa myös muita monikulmioita. Monikulmioista rakennettuja navigaatioverkkoja käyttäviä pelejä ovat muun muassa Valve Softwaren Source-pelimoottoria käyttävät pelit ja Epic Gamesin Unreal Engine -pelimoottoria käyttävät pelit [15][17].

Kolmioista rakennetut navigaatioverkot ovat suurten ympäristöjen mallintamiseen tehokkaampia kuin säännölliset ruudukot [18]. Navigaatioverkko ratkaisee ruutujen kokoon ja määrään liittyvät ongelmat erikokoisilla soluilla, sillä yksinkertaisilla alueilla voidaan käyttää pientä määrää suuria soluja ja yksityiskohtaisilla alueilla voidaan käyttää suurta määrää pieniä soluja. Näin sekä yksityiskohtaiset että yksinkertaiset alueet otetaan tehokkaasti huomioon reitinetsinnässä. Navigaatioverkko voi esimerkiksi kuvata kaupunkia, jossa on rakennuksia ja katuja. Rakennusten mallintamiseen voidaan varata enemmän soluja kuin katujen mallintamiseen, jolloin reitinetsintä toimii tarkasti yksityiskohtaisten rakennusten sisällä ja nopeasti yksinkertaisia katuja pitkin.

Navigaatioverkon solmuja voidaan yhdistellä erityyppisillä linkeillä, jotka voivat auttaa pelihahmoja kulkemaan ympäristössä monimutkaisemmilla tavoilla kuin pelkästään kävelemällä. Nämä linkit voivat esimerkiksi kertoa pelihahmoille paikoista, joissa voi hypätä esteen yli tai ryömiä esteen ali [15]. Linkkien avulla pelihahmot voidaan saada käyttäytymään luontevammin ja pelisuunnittelijat voivat manuaalisesti hienosäätää pelihahmojen käytöstä erilaisissa paikoissa.

5.1 Navigaatioverkon luominen

Navigaatioverkon luomiseen on monia tapoja, jotka voivat olla manuaalisia tai automaattisia [15][17]. Manuaalinen navigaatioverkon luominen tarkoittaa, että ihminen rakentaa verkon käsin jonkin mallinnustyökalun avulla. Automaattisessa navigaatioverkon luomisessa taas jokin algoritmi tai muu ohjelmoitu järjestelmä luo verkon ilman ihmisen apua esimerkiksi tarkastelemalla pelimaastoa. Automaattisia luomistapoja suositaan, sillä manuaalinen navigaatioverkon luominen voi olla työlästä [17]. Ei ole kuitenkaan olemassa yhtä parasta tapaa, sillä verkolta vaadittavat ominaisuudet riippuvat pelin tyypistä ja peliympäristöstä.

Kolmioista rakennetun navigaatioverkon laatua voidaan arvioida muun muassa kolmioiden määrän, kokojen ja muotojen perusteella. Verkko saattaa olla tehoton tai toimia huonosti, jos kolmioita on liikaa, niiden reunat ovat liian isoja tai niiden muodot ovat liian ohuita. [17] Jotkin automaattiset navigaatioverkon luomistavat voivat tuottaa tällaisia huonolaatuisia verkkoja [17], ja niitä voidaan joutua säätämään manuaalisesti.

Pelimaaston luonne voi myös aiheuttaa ongelmia navigaatioverkon automaattisessa luomisessa [6][16]. Esimerkiksi jotkin päällekkäiset tai toisiaan leikkaavat alueet voivat luoda käyttökelvottomia soluja. Tällaisia alueita voidaan kuitenkin tunnistaa ja rakentaa niihinkin toimivat solut. [6]

Navigaatioverkon automaattisessa rakentamisessa voidaan hyödyntää mediaaliakselia [6][7]. Mediaaliakseli on tietorakenne, joka tallentaa kaikki ympäristön pisteet, joilla on enemmän kuin yksi lähin piste ympäristön reunalla. Virtuaaliympäristössä, jossa on useita kerroksia, voidaan ensin rakentaa jokaiselle kerrokselle kaksiulotteinen mediaaliakseli ja sitten yhdistää erilliset mediaaliakselit yhdeksi kolmiulotteiseksi navigaatioverkoksi. [7] Mediaaliakseli voi kuitenkin olla monimutkainen ja sitä voi olla vaikea käyttää [6].

5.2 Navigaatioverkon käyttö

Koska navigaatioverkko on yhdenlainen graafi, sen solujen välille voidaan etsiä reitti käyttämällä reitinetsintäalgoritmeja. Kun reitin muodostavat solut on löydetty, niiden läpi voidaan muodostaa lopullinen viivamainen reitti esimerkiksi solujen reunojen keskipisteiden läpi. [16] Pelihahmo voi sen jälkeen seurata reittiä päästäkseen kohteeseensa.

Solujen reunojen keskipisteiden läpi kulkeva reitti on kuitenkin harvoin paras mahdollinen, sillä se ei ole välttämättä lyhin reitti ja sen muoto saattaa kiemurrella tarpeettomasti. Parempi reitti voidaan löytää esimerkiksi suppiloalgoritmilla, joka määrittää geometrisesti lyhimmän reitin solujen läpi [6]. On kuitenkin hyvä huomata, että peleissä ei haluta aina käyttää aivan lyhintä reittiä, sillä se voi saada pelihahmon liikkeen näyttämään robottimaiselta ja epärealistiselta. Jalankulkijaa esittävä pelihahmo saattaa esimerkiksi oikoa lyhintä reittiä seuratessaan nurmikon yli, vaikka oikea jalankulkija todennäköisesti pysyisi jalkakäytävillä tai muilla selkeillä poluilla.

Navigaatioverkkojärjestelmillä voidaan ohjata yksittäisten hahmojen lisäksi myös useampia hahmoja samanaikaisesti niin, että hahmot osaavat väistää toisiaan [7][18]. Tämä onnistuu muun muassa siksi, että hahmot voivat liikkua verkon solujen sisällä vapaasti, joten niillä on tilaa väistää. Kuten luvussa 4.1 havaittiin, tämä ei olisi mahdollista reittipistegraafilla.

6. YKSINKERTAISEN NAVIGAATIOVERKKOJÄRJESTELMÄN TOTEUTUS

Tutkielman yhteydessä toteutettiin esimerkkiohjelma, joka havainnollistaa navigaatioverkkopohjaisten reitinetsintäjärjestelmien toimintaa. Ohjelman lähdekoodi on avoimesti saatavilla Githubissa [19], ja repositoriossa on myös valmiiksi käännetty 32-bittinen ohjelmatiedosto Windows-käyttöjärjestelmille. Jos ohjelman haluaa kääntää itse, käytetyt kirjastot tulee ladata erikseen. Lähdekoodin yhteydessä on linkit kirjastojen lataamiseen.

6.1 Valitut työkalut

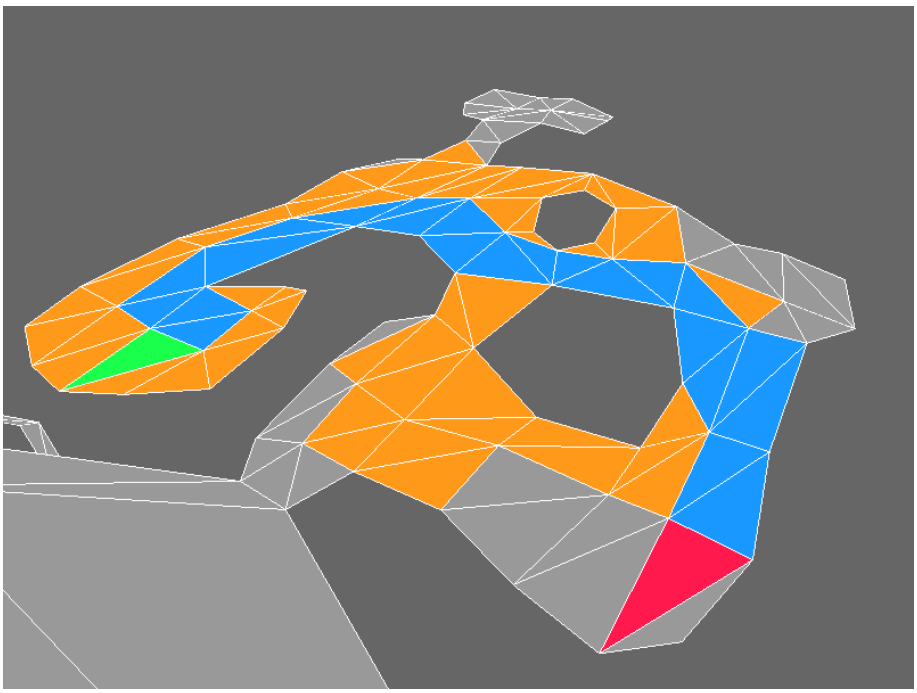
Kaikkien esimerkkiohjelman toteutuksessa käytettyjen työkalujen valinnassa painotettiin yksinkertaisuutta ja yleisyyttä, jotta esimerkkiohjelma pysyisi pienenä, yksinkertaisena ja monikäyttöisenä. Kirjastoiksi valittiin paljon käytettyjä kirjastoja, jotta esimerkkiohjelma toimisi luotettavasti ja käyttäisi samoja työkaluja, joita oikeat pelit käyttävät. Ohjelmointikieleksi valittiin C++ sen yleisyyden vuoksi: C++:aa on käytetty yleisesti pelien matalan tason toteutuksessa jo parin vuosikymmenen ajan, eikä mikään ohjelmointikieli vielä vaikuta olevan syrjäyttämässä sitä [20].

Työkaluksi navigaatioverkon graafiseen esittämiseen valittiin OpenGL-grafiikkarajapinta (Open Graphics Library), sillä se on yhteensopiva monen suoritusalustan kanssa [21] ja melko yksinkertainen. Lisäksi monet pelit käyttävät tai tukevat sitä, joten samaa grafiikkakoodia olisi varmasti mahdollista käyttää myös oikeassa pelissä. Muita yleisiä pelien käyttämiä grafiikkarajapintoja ovat DirectX, Metal ja Vulkan, mutta DirectX ja Metal toimivat vain Microsoftin tai Applen alustoilla [21] ja Vulkan on huomattavasti OpenGL:ää työläämpi käyttää.

OpenGL:n käyttöön valittiin kirjastoiksi GLFW (Graphics Library Framework) ja GLM (OpenGL Mathematics). Lisäksi OpenGL:n vaatimat funktio-osoittimet luotiin käyttämällä Glad-työkalua. Navigaatioverkkojen lataamiseen tiedostoista valittiin Wavefront OBJ-tiedostoformaatti ja tinyobjloader-kirjasto. Navigaatioverkon sisäiseen reitinetsintään valittiin A*-algoritmi.

6.2 Ohjelman käyttö

Esimerkkiohjelma lataa OBJ-tiedostosta 3D-mallin, muodostaa siitä navigaatioverkon ja näyttää sen ikkunassa. Näkymää voi liikuttaa WASD-näppäimillä ja kääntää hiirellä vetämällä. Verkon pinnalta voi klikata kahta solua, jolloin ohjelma määrittää solujen välisen reitin ja piirtää sen näkyviin. Reitinetsinnän jälkeen ohjelma värittää solut samoilla väreillä, joita käytettiin kuvassa 1: reittiin kuuluvat solut väritetään sinisiksi, lähtöpiste vihreäksi, loppupiste punaiseksi ja muut reitinetsintäalgoritmin käsittelemät solut oransseiksi. Kuvassa 4 on esimerkki ohjelman näkymästä.



Kuva 4. *Esimerkkiohjelman näkymä reitinetsintätehtävän jälkeen.*

Kuvasta 4 voidaan nähdä solujen edellä mainitun mukainen väritys reitinetsintätehtävän jälkeen. Käsittelemättömät verkon solut on väritetty vaaleanharmaalla ja tausta tummanharmaalla. Ohjelman lähdekoodin mukana on tutkielman kuvissa käytetyn esimerkkinavigaatioverkon sisältävä OBJ-tiedosto, mutta tiedosto voidaan helposti vaihtaa toiseen ja näin testata myös muunlaisia navigaatioverkkoja.

6.3 Koodin toiminta

Ohjelmakoodi jakautuu kolmeen luokkaan, jotka ovat NavMesh, NavMeshRenderer ja Window. NavMesh toteuttaa navigaatioverkon rakentamisen ja käsittelyn sekä reitinetsintäalgoritmien suorittamisen. NavMeshRenderer huolehtii navigaatioverkon piirtämisestä ja Window ikkunan toiminnasta sekä käyttäjän syötteiden käsittelystä.

NavMesh-luokka rakentaa navigaatioverkon soluja kuvaavista Cell-tietueista, joille merkitään yhteydet viereisiin soluihin. Näin solut muodostavat graafin ja niihin voidaan soveltaa A*-algoritmia. Koska A*:n täytyy solujen yhteyksien lisäksi merkitä soluille paras reitti alkupistettä kohti ja minimiarvio etäisyydestä loppupisteeseen, NavMesh käyttää lisäksi erillisiä PathNode-tietueita näiden tietojen säilyttämiseen. A*:n suorittamisen aikana löydetyille soluille luodaan aina niitä vastaava PathNode, joka yhdistetään soluun osoittimilla. Näin hakukohtaisia tietoja ei tarvitse säilyttää solujen yhteydessä, vaan tiedot voidaan luoda ja poistaa haun sisällä. Ohjelma 1 esittää A*-algoritmin toteutuksen NavMesh-luokassa.

```
std::forward_list<Cell*> NavMesh::AStar(PathfindingTask& task) const
{
  // Create PathNode for start cell
  CreateOrUpdatePathNodeForCell(task.startCell, nullptr, task);

  // Process PathNodes until no new ones are discovered
  while (!task.discovered.empty())
  {
    // Pop the highest priority node from the priority queue
    auto highestPriorityUnprocessedNode = task.discovered.begin();
    PathNode* node = *highestPriorityUnprocessedNode;
    task.discovered.erase(highestPriorityUnprocessedNode);

    // If the node is the goal node, the path has been found
    if (node->cell == task.endCell)
    {
      task.endPathNode = node;
      break;
    }

    // Process each cell connected to the current cell
    for (Cell* connectedCell : node->cell->connectedCells)
    {
      CreateOrUpdatePathNodeForCell(connectedCell, node, task);
    }
  }

  // Pathfinding finished, collect the path if one was found
  return CollectPath(task);
}
```

Ohjelma 1. *Esimerkkiohjelman A*-algoritmin toteuttava funktio.*

Ohjelmassa 1 näkyvä A*:n toteutus etsii reitin parametrina annetun PathfindingTask-tietueen avulla, joka sisältää reitin alku- ja loppusolun sekä prioriteettijonon löydettyjen solujen käsittelyn priorisoimiseen. Riviltä 7 alkava silmukka etsii uusia soluja tutkimalla aiemmin löydettyjen solujen yhteyksiä ja päivittää aiemmin löydettyjen solujen tietoja.

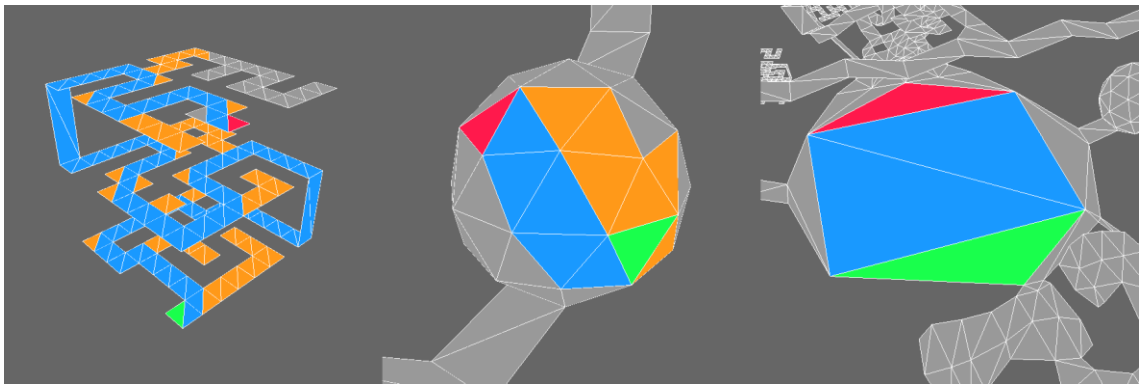
Uudet solut lisätään aina käsiteltäviksi prioriteettijonoon. Solu päivitetään, jos siitä löydetään aiempaa lyhyempi reitti kohti aloitussolua, sillä lyhin reitti halutaan pitää muistissa lopullisen reitin muodostamista ja solujen priorisointia varten. Prioriteettijono on toteutettu std::set-säiliönä, joka järjestää solut lyhimmän mahdollisen reitin pituuden perusteella. Näin A*-algoritmi tutkii ne solut ensin, joiden läpi kulkeva mahdollinen reitti olisi lyhin.

Silmukka katkeaa, kun uusia käsiteltäviä soluja ei ole enää jäljellä tai loppusolu löydetään. Silmukan jälkeen CollectPath-funktio kerää reittiin kuuluvat solut listaan palaamalla viimeisestä PathNodesta kohti alkusolua. Jos reittiä ei löytynyt, CollectPath palauttaa tyhjän listan.

NavMesh on tutkielman tavoitteiden kannalta ainoa olennainen luokka, joten kahta muuta ohjelman käyttämää luokkaa ei kuvailla tässä luvussa tarkemmin. Ohjelman lähdekoodi on kuitenkin laajasti kommentoitua, joten toteutuksen muita yksityiskohtia voi halutessaan tutkia kommenttien avulla.

6.4 Havainnot

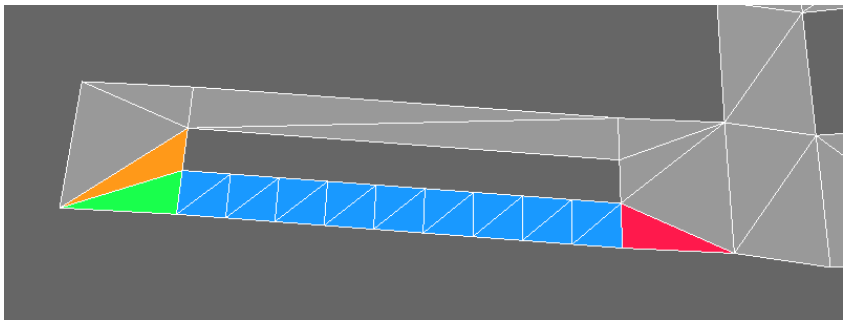
Esimerkkiohjelma havainnollistaa, että navigaatioverkkoa voidaan käyttää reitinetsintään myös kerrostetuissa, epätavallisen muotoisissa ja erikokoisista soluista koostuvissa ympäristöissä. Tämä voidaan nähdä kuvasta 5, joka esittää kolme erilaista reitinetsintätilannetta esimerkkiohjelmassa.



Kuva 5. *Kolme erilaista reitinetsintätehtävän tulosta esimerkkiohjelmassa.*

Kuvassa 5 on reitinetsinnän tulokset vasemmalla kerrostetussa labyrintissä, keskellä pallon muotoisen monikulmion pinnalla ja oikealla suuren, yksinkertaisen aukion päällä. Kuten luvussa 4.2 todettiin, reitinetsintä ei onnistuisi näissä tilanteissa säännöllisen kaksiulotteisen ruudukon avulla, sillä ruudukko ei tue pystysuunnassa päällekkäisiä alueita tai erikokoisia ruutuja.

Esimerkkiohjelman avulla voidaan myös havaita, että A*-algoritmi yrittää löytää nimenomaan pituudeltaan lyhimmän reitin, eikä reittiä, jolla on vähiten soluja. Tämä on tärkeää erikokoisia soluja sisältävässä navigaatioverkossa, sillä lyhin reitti ei välttämättä sisällä pienintä määrää soluja. Kuva 6 havainnollistaa tätä: A*-algoritmi on löytänyt vihreän ja punaisen kolmion välille lyhimmän reitin, vaikka sillä on enemmän soluja kuin viereisellä reitillä.



Kuva 6. *Esimerkkiohjelman löytämä reitti, jolla on useita soluja.*

Tästä nähdään, että erikokoisten solujen yhdisteleminen ei aiheuta A*-algoritmille ongelmia, joten A*:n pitäisi olla toimiva valinta navigaatioverkon reitinetsintäalgoritmiksi. Navigaatioverkko toimii tehokkaammin ja tarkemmin, kun siinä yhdistellään erikokoisia soluja, sillä suuret solut vähentävät peliympäristön peittämiseen tarvittavaa solumäärää ja pienet solut mahdollistavat ympäristön muotojen mallintamisen tarkasti.

Taulukossa 1 on vertailtu esimerkkiohjelman reitinetsintään kulunutta aikaa erikokoisissa navigaatioverkoissa. Ajat ovat keskiarvoja kymmenestä mittauksesta, ja mittaukset on otettu vain A*-algoritmin käyttämästä ajasta, joten niissä ei ole mukana esimerkiksi verkon piirtojärjestelmien käyttämää aikaa tai alku- ja loppusolujen valintaan kulunutta aikaa. Mittaukset tehtiin säännöllisissä navigaatioverkoissa, joissa käsitellyillä soluilla oli aina 3 viereistä solua ja viereisten solujen väliset etäisyydet olivat aina samat. Ohjelman suorittaneessa tietokoneessa oli Windows 10 Home -käyttöjärjestelmä, Intel Core i7-3770 -prosessori ja 8 gigatavua DDR3-RAM-muistia.

Taulukko 1. *Reitinetsintään kuluneita aikoja erilaisissa testitilanteissa.*

Testi	Aika (µs)	Soluja käsitelty	Aikaa per solu (µs)	Soluja koko verkossa
1	92,40	100	0,92	900
2	92,50	100	0,93	10000
3	94,40	100	0,94	90000
4	385,10	500	0,77	900
5	486,60	500	0,97	10000
6	459,90	500	0,92	90000

Taulukon 1 ajat eivät ole tarkkoja, sillä reitinetsinnän kuluttama aika heittelehti paljon, ja ajat on laskettu pienestä määrästä mittauksia. Tarkemmat ajat edellyttäisivät suurempaa määrää mittauksia ja kontrolloidumpaa testiympäristöä, jossa esimerkiksi käyttöjärjestelmän taustaprosessit eivät kuluta ylimääräistä aikaa ja A*-algoritmi suorittaa täsmälleen samat operaatiot samassa järjestyksessä eri navigaatioverkoissa.

Mittauksista voidaan silti tehdä hyödyllisiä havaintoja. Kaikki ajat ovat alle puolen millisekunnin pituisia, joten järjestelmän pitäisi toimia riittävän nopeasti pelin käyttöön. Monet pelit esimerkiksi päivittävät pelimaailmaa 30-60 kertaa sekunnissa, jolloin yhteen päivitykseen on 16-33 millisekuntia aikaa. Reitinetsintäaika mahtuu tähän hyvin, vaikka päivityksen aikana etsittäisiin useampia reittejä ja tehtäisiin myös muita operaatioita.

Navigaatioverkon kokonaissolujen määrä ei näytä vaikuttavan reitinetsinnän nopeuteen, sillä verkon solujen määrän muuttaminen ei muuta taulukon 1 aikojen kokoluokkaa. Reitinetsintä ei siis hidastu, vaikka verkossa olisi paljon soluja. Testi 6 on ollut epätarkkuuksien vuoksi jopa nopeampi kuin testi 5, vaikka testin 6 navigaatioverkossa oli monta kertaa enemmän soluja.

Käsiteltyjen solujen määrän kasvattaminen taas pidentää aikoja selvästi. Lisäksi koska reitinetsinnän käyttämä aika per käsitelty solu pysyy lähes samana kaikissa testeissä, voidaan päätellä, että reitinetsinnän suoritusaika kasvaa testitilanteissa lineaarisesti käsiteltyjen solujen määrän mukana. Suoritusajan asymptoottinen yläraja on tällöin O(n), missä n on käsiteltyjen solujen määrä.

Nämä havainnot vaikuttavat loogisilta, sillä mittauksissa käytetyissä säännöllisissä navigaatioverkoissa kaikki solut ovat samanlaisia, joten niiden käsittelyyn pitäisi kulua suunnilleen saman verran aikaa. Lisäksi A*-algoritmi ei käsittele soluja, joita se ei ole löytänyt, joten verkon kokonaissolumäärän ei pitäisikään vaikuttaa algoritmin toimintaan.

Lineaarisesti kasvavan suoritusajan pitäisi olla melko hyvä, sillä se ei lähde kasvamaan nopeammin käsiteltyjen solujen lisääntyessä. Tämä auttaa erityisesti pitkien reittien löytämistä, sillä niitä etsittäessä saatetaan käsitellä suuri määrä soluja. Suoritusaikaa voitaisiin kuitenkin parantaa esimerkiksi rajaamalla tutkittavien solujen joukkoa [9].

6.5 Laajennusmahdollisuuksien arviointi

Esimerkkiohjelmaa voitaisiin laajentaa moniin suuntiin. Järjestelmän ympärille voitaisiin rakentaa peli, jonka hahmot hyödyntävät reitinetsintää liikkuessaan. Monet erilaiset pelit olisivat mahdollisia, kuten strategiapelit, sotapelit, roolipelit ja autopelit, sillä kaikki nämä pelityypit usein käyttävät liikkuvia pelihahmoja.

Tällä hetkellä esimerkkiohjelman laskemaa reittiä ei voida sellaisenaan vielä käyttää hahmojen ohjaamiseen, vaan ensin täytyy määrittää, miten hahmo halutaan ohjata solujen läpi. Hahmo voitaisiin ohjata kohdepisteeseen esimerkiksi solujen reunojen keskipisteiden läpi [16] tai suppiloalgoritmilla määritettyä lyhintä mahdollista solujen läpi kulkevaa reittiä pitkin [18]. Ohjaustapa tulee päättää pelin muun rakenteen ja vaatimusten perusteella ottamalla huomioon muun muassa hahmojen liikuttamiseen käytetyt järjestelmät.

Navigaatioverkkoon voitaisiin lisätä erilaisia lisäominaisuuksia, joilla pelihahmot saataisiin käyttäytymään monimutkaisemmin ja älykkäämmin. Verkon soluille voitaisiin esimerkiksi merkitä niiden korkeus ja leveys, jolloin hahmot saataisiin välttämään reittejä, joiden läpi ne eivät mahdu [18]. Hahmot voitaisiin myös esimerkiksi laittaa hyppimään esteiden yli merkitsemällä kohdat, joiden yli pääsee hyppäämällä [15].

Lisäksi solujen sijainteja voitaisiin arvioida tarkemmin, sillä tällä hetkellä ohjelma laskee solujen väliset etäisyydet käyttämällä solujen keskipisteitä. Koska hahmojen ei välttämättä tarvitse kulkea solujen keskipisteiden läpi, reitinetsintäjärjestelmä ei tällaisenaan välttämättä arvioi reittien pituuksia täysin todenmukaisesti. Tämä ongelma vaikuttaa erityisesti suuriin soluihin, sillä niiden keskipisteet ovat kaukana solun reunoista.

Esimerkkiohjelma ei rajaa mahdollista käyttökohdetta vain peleihin, joten myös muunlaiset navigaatioverkkopohjaista reitinetsintää tarvitsevat järjestelmät olisivat ohjelman pohjalta mahdollisia. Esimerkiksi jos navigaatioverkon rakentaisi jostain oikean maailman ympäristöstä, ohjelman löytämiä reittejä voisi käyttää robotin liikutteluun tai reittineuvontaan monikerroksisessa rakennuksessa.

7. YHTEENVETO

Tutkielmassa selvisi, että hyviä reitinetsintäalgoritmeja pelien käyttöön ovat A*-algoritmi, geneettiset algoritmit ja muurahaisyhdyskuntaoptimointi. A*-algoritmi on näistä yksinkertaisin ja soveltuu hyvin useimpiin reitinetsintätilanteisiin. Se ei kuitenkaan toimi tehokkaasti muuttuvissa ympäristöissä, ja geneettiset algoritmit ja muurahaisyhdyskuntaoptimointi soveltuvat paremmin niihin.

A*-algoritmista on tehty erilaisia muunnoksia, joihin kannattaa tutustua A*-algoritmia käytettäessä, sillä ne saattavat tehostaa alkuperäistä algoritmia joissain tilanteissa. Muunnosten kohdalla pitää kuitenkin arvioida, soveltuuko muunnos juuri kyseisen pelin tarpeisiin, sillä jotkin muunnokset saattavat esimerkiksi nopeuttaa reitinetsintää, mutta lisätä pelin muistinkäyttöä.

Reitinetsintäalgoritmien käyttämiseksi peliympäristöstä täytyy rakentaa graafi. Kolme yleistä peleissä käytettyä graafimallia ovat reittipistegraafi, säännöllinen ruudukko ja navigaatioverkko. Näistä soveltuvin kolmiulotteisiin ympäristöihin on navigaatioverkko, joka mahdollistaa esimerkiksi monikerroksiset ympäristöt ja yksityiskohtaisen ympäristön muotojen mallintamisen. Navigaatioverkko voidaan rakentaa automaattisesti tai manuaalisesti, ja automaattisesti luotuja verkkoja voidaan myös tarvittaessa hioa paremmiksi manuaalisesti.

Kolmiulotteisten ympäristöjen reitinetsinnässä ongelmia aiheuttaviksi tilanteiksi havaittiin pystysuunnassa päällekkäiset alueet, suuret alueet ja pelihahmojen liikuttaminen joustavasti. Selvisi, että nämä ongelmat pystyy ratkaisemaan käyttämällä navigaatioverkkoa.

Tutkielman yhteydessä toteutetusta esimerkkiohjelmasta voidaan havaita navigaatioverkkojen soveltuvuus kolmiulotteisiin ympäristöihin. Ohjelma havainnollistaa muun muassa verkon erikokoisten solujen käyttöä ja monikerroksisten ympäristöjen reitinetsintää. Koska esimerkkiohjelma on toteutettu yleisillä pelien käyttämillä työkaluilla, se voitaisiin helposti yhdistää erilaisiin peleihin.

LÄHTEET

[1] Namco, Pac-Man, kolikkopeli, 1980.

[2] Taito, Space Invaders, kolikkopeli, 1978.

[3] Electronic Arts, The Need for Speed, 3DO Interactive Multiplayer, 1994.

[4] Infinity Ward, Call of Duty, Microsoft Windows, 2003.

[5] A. Rafiq, T.A.A. Kadir, S. N. Ihsan, Pathfinding Algorithms in Game Development, IOP Conference Series: Materials Science and Engineering, Vol.769, No.1, 2020, s. 1-10.

[6] M. Kallmann, M. Kapadia, Geometric and Discrete Path Planning for Interactive Virtual Worlds, teoksessa Synthesis Lectures on Visual Computing, Vol.23, No.1, Morgan & Claypool Publishers, 2016, s. 1-201.

[7] W. Van Toll, A.F. Cook, R. Geraerts, Navigation Meshes for Realistic Multi-Layered Environments, IEEE International Conference on Intelligent Robots and Systems, 2011, s. 3526–3532.

[8] P. E. Hart, N. J. Nilsson, B. Raphael, A Formal Basis for the Heuristic Determination of Minimum Cost Paths, IEEE Transactions on Systems Science and Cybernetics, Vol.4, No.2, 1968, s.100-107.

[9] S. Rabin, N. Sturtevant, Faster A* with Goal Bounding, teoksessa S. Rabin (toim.) Game AI Pro 3: Collected Wisdom of Game AI Professionals, CRC Press, 2017, s. 275-282.

[10] N. Pelechano, C. Fuentes, Hierarchical Path-Finding for Navigation Meshes (HNA*), Computers & Graphics, Vol.59, No.1, 2016, s. 68-78.

[11] E. W. Dijkstra, A Note on Two Problems in Connexion With Graphs, Numerische Mathematik, Vol.1, No.1, 1959, s. 269-271.

[12] K. Wang, PathFinder: A Java Implementation of A* (A star) and Dijkstra search algorithm for comparison, 2015, päivitetty 12.12.2016. Saatavissa (viitattu 22.2.2022): https://github.com/kevinwang1975/PathFinder

[13] S. N. Sivanandam, S. N. Deepa, Introduction to Genetic Algorithms, Springer-Verlag Berlin Heidelberg, 2008, s. 15-36, 410-423.

[14] N. Sturtevant, Choosing a Search Space Representation, teoksessa S. Rabin (toim.) Game AI Pro: Collected Wisdom of Game AI Professionals, CRC Press, 2013, s. 253-257.

[15] S. Golodetz, Automatic Navigation Mesh Generation in Configuration Space, ACCU, 2013, päivitetty 10.2013. Saatavissa (viitattu 30.1.2022): https://accu.org/journals/overload/21/117/golodetz_1838/

[16] G. Snook, Simplified 3D Movement and Pathfinding Using Navigation Meshes, teoksessa M. DeLoura (toim.) Game Programming Gems, Charles River Media, 2000, s. 288-297.

[17] R. Oliva, N. Pelechano, Automatic Generation of Suboptimal NavMeshes, teoksessa J.M. Allbeck, P. Faloutsos (toim.) Motion in Games, Springer-Verlag Berlin Heidelberg, 2011, s. 328-339.

[18] M. Kallmann, Navigation Queries from Triangular Meshes, teoksessa R. Boulic, Y. Chrysanthou, T. Komura (toim.) Motion in Games, Springer-Verlag Berlin Heidelberg, 2010, s. 230-240.

[19] A. Leppäaho, Navigation Mesh Visualizer, 2022, päivitetty 9.5.2022. Saatavissa (viitattu 9.5.2022): https://github.com/ArttuLeppaaho/NavMeshVisualizer

[20] A. Venigalla, S. Chimalakonda, On the Comprehension of Application Programming Interface Usability in Game Engines, Software, Practice & Experience, Vol.51, No.8, 2021, s. 1728-1744.

[21] C. Ioannidis, A.-M. Boutsi, Multithreaded Rendering for Cross-Platform 3D Visualization Based on Vulkan API, International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences, XLIV-4/W1-2020, 2020, s. 57-62.