Alex Leppäkoski

Polunetsintä pelisovelluksessa

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

1.5.2021

Tiivistelmä

Tekijä:	Alex Leppäkoski
Otsikko:	Polunetsintä pelisovelluksessa
Sivumäärä:	34 sivua
Aika:	1.5.2021
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikka
Ammatillinen pääaine:	Pelisovellukset
Ohjaaja:	Lehtori Miikka Mäki-Uuro

Insinöörityön tarkoituksena oli toteuttaa pelisovellukseen polunetsintä, jotta tietokoneen ohjaamat pelihahmot pystyvät liikkumaan pelikartalla sijainnista toiseen esteiden lomassa. Pelisovellus luotiin käyttämällä C++-ohjelmointikieltä ja OpenGL-kirjastoa. Tavoitteena oli tehdä polunetsinnästä suorituskyvyltään tehokas ratkaisu pelisovellukselle siten, että peli ei hidastu näkyvästi sen suorituksen takia.

Insinöörityön toteutukseen käytettiin A*-algoritmia, joka on pelisovelluksissa yksi käytetyimmistä polunetsintäalgoritmeista. Heuristiikaksi valittiin Manhattanin etäisyys, jota polunetsintäalgoritmi käyttää arvioimaan etäisyyttä kohteeseen. A*-algoritmi määrittelee kuljettavan reitin muun muassa heuristiikan avulla.

Pelikartta pilkottiin pieniin itsenäisiin kokonaisuuksiin polunetsinnän nopeuttamiseksi. Kun käsiteltävä alue on pienempi, suoritusnopeuden enimmäiskesto lyhenee. Suoritusnopeutta pystytään parantamaan entisestään hierarkkisen tietorakenteen avulla.

Insinöörityön lopputuloksena saatiin toteutettua polunetsintä pelisovellukselle. Polunetsintä ohjelmoitiin omaan moduuliinsa. Selvisi, että polunetsintäominaisuus vaatii lisää optimointia, jotta se olisi julkaisuvalmis.

Insinöörityö edisti pelisovelluksen tavoitteita. Insinöörityö mahdollisti pelisovelluksen pelaamista sen nykyisessä tilassa ja mahdollistaa myös tulevia ominaisuuksia, joissa vaaditaan polunetsintää. Insinöörityöraportin avulla voi pohtia polunetsintään tarvittavia ratkaisuja myös muihin pelisovelluksiin ja helpottaa oikeiden ratkaisujen löytämistä.

Avainsanat: polunetsintä, A*

Abstract

Author:	Alex Leppäkoski
Title:	Pathfinding in Game Applications
Number of Pages:	34 pages
Date:	1 May 2021
Degree:	Bachelor of Engineering
Degree Programme:	Information and Communications Technology
Professional Major:	Game Applications
Instructor:	Miikka Mäki-Uuro, Senior Lecturer

The purpose of this final year project was to implement a pathfinding algorithm for a game application. The implementation was meant to enable in-game characters to move between points across the game map. Pathfinding was required due to obstacles that would limit the movement of the characters. The game application is programmed with C++ programming language and uses Open Graphics Library for the graphics of the game. The goal was to create an implementation of the pathfinding in a way that there is no visible lag while it is executed.

The thesis covers various methods for the pathfinding project including search algorithms. A* algorithm was chosen for the implementation of the project. A* algorithm is one of the most used pathfinding algorithms for game applications. This study reflects on implementations of data structures for the game map in various known games.

A pathfinding implementation was made for the game application as the results of this final year project. As the game application itself, the pathfinding project was programmed with C++ programming language. During the project it became clear that in order to launch the pathfinding implementation, the results of this study need to be optimized for quality.

In conclusion, pathfinding was an important milestone for the game application making it possible to play the game with Artificial Intelligence. It will also allow some of the planned features for the game application that have not yet been implemented.

Keywords: Pathfinding, Search, A*

Sisällys

1 Johdanto

Insinöörityön tarkoituksena on toteuttaa polunetsintä pelisovellukseen siten, että pelimaailmassa olevat hahmot kykenevät liikkumaan kaksiulotteisen pelikartan sijainnista toiseen mahdollisimman suorituskykytehokkaasti ja luotettavasti. Polunetsintä on siirtymisen suunnittelua kahden eri pisteen välillä tietorakenteessa. Sen tarkoituksena on ratkaista esimerkiksi sokkeloita tai löytää polulle lyhin reitti.

Työ toteutetaan tekijän omaa vireillä olevaa vapaa-ajan peliprojektia varten. Peliprojekti on toteutettu C/C++-kielillä, minkä lisäksi se tukeutuu OpenGL-kirjastoon. Insinöörityö on tarkoitus toteuttaa sellaiseksi kokonaisuudeksi, jota peliprojekti kokonaisuudessaan pystyy hyödyntämään. Insinöörityötä on tarkoitus soveltaa peliprojektissa niin, että se omalta osaltaan luo uskottavan ja hauskan tekoälyn pelissä oleville hahmoille, joiden parissa pelaaja toimii.

Insinöörityö toteutetaan tutustumalla etsinnän ja polunetsinnän teoriaan. Polunetsinnän toteutustapaa pohditaan internetlähteiden avulla, minkä jälkeen pelisovellukseen implementoidaan polunetsintä.

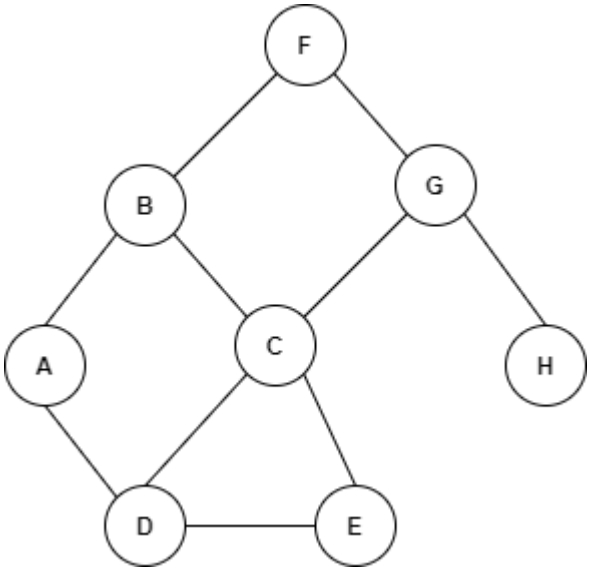
Luvussa 2 tutustutaan yleisesti käytössä oleviin etsinnässä käytettäviin työkaluihin. Luvussa 3 pohditaan erilaisia tapoja ja toteutuksia tietorakenteelle, jossa polunetsintä lopulta suoritetaan. Luvusta 4 löytyvät insinöörityöprojektin spesifikaatiot, suunnittelu ja toteutus.

2 Polunetsintäalgoritmit

2.1 Verkot

Polunetsintää voidaan mallintaa verkon (engl. graph) avulla, joka abstrahoi käsiteltävää tietoa. Verkko [1] koostuu solmuista ja solmupareista eli kaarista (kuva 1), jotka yhdistävät solmuja toisiinsa. Verkko voi olla suuntaamaton tai suunnattu,

jolloin kaarta pitkin voi kulkea vain määriteltyyn suuntaan. Suuntaamattomassa verkossa kaaria pitkin voi kulkea kumpaan tahansa suuntaan.

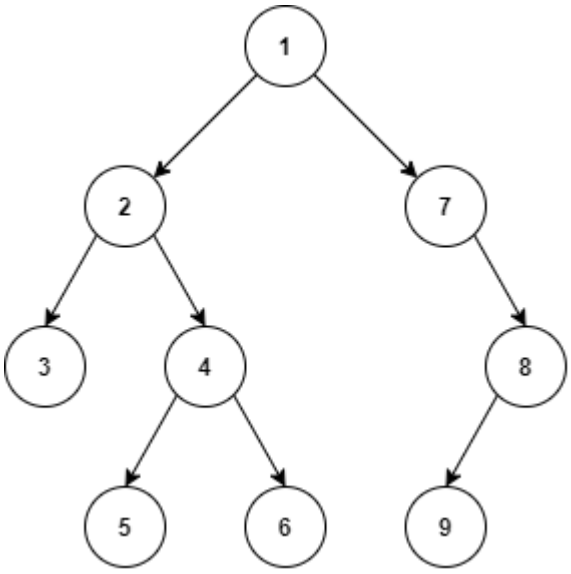


Kuva 1. Verkon mallinnus.

Polku on kokoelma solmupareja, joita seuraamalla päästään lähtösolmusta päätesolmuun verkon sisällä. Kun verkon sisällä käsitellään solmua, tunnetaan sen sisältämä data ja sen kaarien osoittamat solmut, joihin voidaan seuraavaksi siirtyä. Verkkoa läpikäyvää algoritmia suorittaessa voidaan käyttää ehtolausetta, jonka täyttyessä lopetetaan suorittaminen. Tällainen ehtolause yleensä perustuu solmun dataan ja etsinnän tilaan.

Lähtösolmusta tietyn solmun haku verkossa voidaan toteuttaa algoritmisesti. Näistä hakualgoritmeista mainittakoon syvyyssuuntainen haku (engl. depth-first search) ja leveyssuuntainen haku (engl. breadth-first search).

Syvyyssuuntaisessa haussa [2] (kuva 2) nimensä mukaisesti priorisoidaan solmun etsintää syvyyssuunnassa. Solmua etsiessä koetetaan ensisijaisesti ottaa askel aina syvemmälle, kunnes kaaria ei löydy enempää. Tällöin hakua kelataan takaisin ylempään solmuun, kunnes etsintä voi haarautua vielä käsittelemättömästä kaaresta. Kohteeseen törmätessä lopetetaan etsintä ja palautetaan löydetty solmu, muutoin algoritmi käsittelee koko verkon.



Kuva 2. Mallinnus syvyyssuuntaisen haun järjestyksestä puu-tietorakenteessa.

Laajassa ja suuntaamattomassa verkossa syvyyssuuntainen haku voi tuottaa melkoisen sokkelon, ja siksi sitä voidaankin itse asiassa soveltaa esimerkiksi pelikentän generoimiseen, kun kentästä halutaan vaikkapa labyrinttimäinen luolasto.

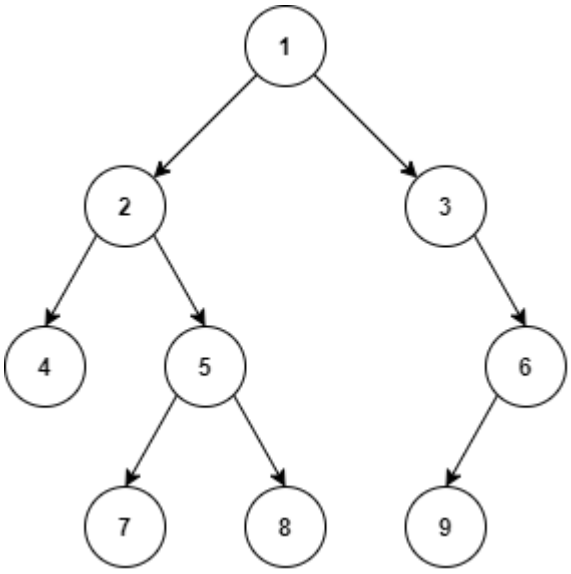
Esimerkkikoodissa 1 on toteutettu syvyyssuuntainen haku. Funktio alkaa siten, että sen osoittimen kautta parametrina saatu solmu merkitään löydetyksi. Jos kyseisen solmun data vastaa haettua dataa, palautetaan tämä solmu. Jos solmu ei vastannut haettua solmua, käydään läpi sen naapurisolmut, joita ei ole vielä merkitty löydetyiksi. Jokaisen tällaisen naapurisolmun kanssa kutsutaan samaa funktiota rekursiivisesti, minkä ansiosta haku toimii syvyyssuunnassa. Jos funktiokutsu palauttaa pointterin, joka ei ole tyhjä, on kohdesolmu löydetty. Jos kohdesolmua ei löydetä ollenkaan, hakufunktio palauttaa tyhjän pointterin.

```
Node* depth_first_search(Data data, Node* node) {
     node->discovered = true;

     if (node->data() == data) {
          return node;
     }

     foreach (Node* child : node->adjacent_nodes()) {
          if (node->discovered) {
               continue;
          }

          Node* result = depth_first_search(data, child);

          if (result != nullptr) {
               return result;
          }
     }

     return nullptr;
}
```

Esimerkkikoodi 1. Rekursiivisesti toteutettu syvyyssuuntainen haku.

Jos verkossa on mahdollista ajautua silmukkaan, kuten verkko-tietorakenteessa usein tapana on, täytyy jo löydetyt solmut korvamerkitä silmukan loputtoman suorittamisen välttämiseksi. Tällöin ei käsitellä enää uudestaan jo käsiteltyjä solmuja.

Leveyssuuntaisessa haussa [3] (kuva 3) sen sijaan asetetaan kaikki käsiteltävän solmun lapsisolmut jonoon, josta käsitellään solmut tulojärjestyksessä ja lisätään myös niiden lapsisolmut saman jonon perälle. Näin tulee käsiteltyä solmut askelpituus kerrallaan syvyyden sijaan.



Kuva 3. Mallinnus leveyssuuntaisen haun järjestyksestä puu-tietorakenteessa.

Esimerkkikoodissa 2 luodaan aluksi jono. Juurisolmu merkitään löydetyksi eli sitä ei enää uudestaan voida ottaa käsittelyyn. Sitten juurisolmu lisätään jonoon ja jonon läpikäynti aloitetaan. Läpikäynnissä otetaan jonon ensimmäinen solmu, joka sitten myös poistetaan jonosta. Jos solmun data vastaa etsittyä dataa, palautetaan kyseinen solmu. Muutoin käydään läpi kyseisen solmun naapurisolmut, jotka ei ole vielä merkitty löydetyiksi. Naapurisolmut merkitään löydetyiksi ja asetetaan jonon perälle. Sitten läpikäynti jatkuu samaan tapaan, kunnes jono on tyhjä. Jos haettua solmua ei löytynyt, funktio palauttaa tyhjän pointterin.

```
Node* breadth_first_search(Data data, Node* root) {
     std::queue<Node*> queue;
     root->discovered = true;
     queue->push_back(root);

     while (!queue.empty()) {
          Node* node = queue->front();
          queue->pop();

          if (cursor->data() == data) {
               return node;
          }

          foreach (Node* child : node->adjacent_nodes()) {
               if (!child->discovered) {
                    child->discovered = true;
                    queue.push_back(child);
               }
          }
     }

     return nullptr;
}
```

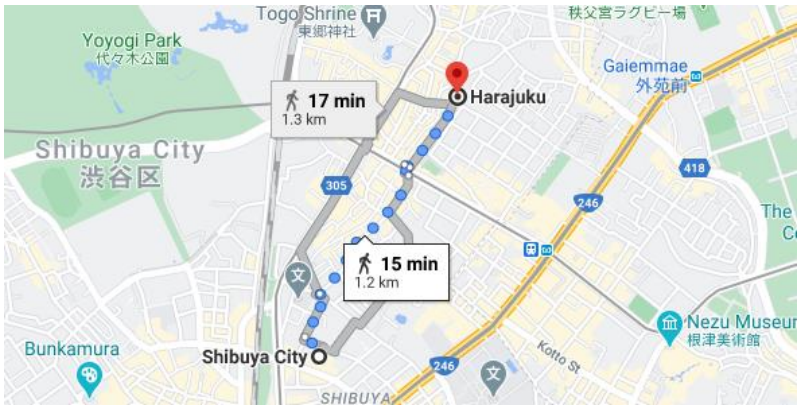
Esimerkkikoodi 2. Esimerkkitoteutus leveyssuuntaisesta hausta.

Kumpikaan hakualgoritmeista ei ole universaalisti nopeampi tai parempi ratkaisu. Jos kuitenkin tiedetään kohteen ilmenevän todennäköisemmin juurisolmun lähettyvillä tai verkon olevan todella syvä, leveyssuuntainen haku on kannattavampaa.

Leveyssuuntainen haku löytää aina sen reitin, johon kuluu vähiten askelia, mutta se ei ota huomioon, ovatko kaaret painotettuja. Painotetut verkot ovat tarpeen, kun joidenkin kaarien käsittely vie enemmän resursseja kuin toisen.

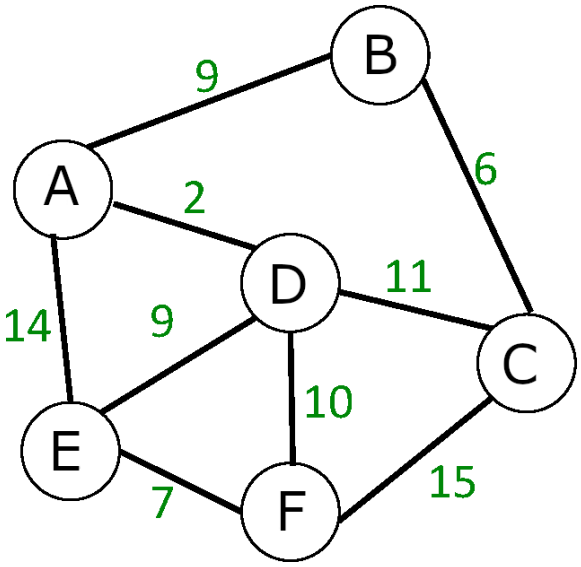
2.2 Dijkstran algoritmi

Dijkstran algoritmi on Edsger W. Dijkstran vuonna 1956 luoma algoritmi, jota käytetään verkon lyhimpien reittien löytämiseksi [4]. Sitä sovelletaan muun muassa etäisyyksien vertailuun tai reittioppaisiin (kuva 4). Sillä ei ole tietoa päätesolmusta etukäteen, eli se löytää lyhimmän reitin jokaiseen solmuun, minkä takia se soveltuu hyvin monikohteiseen polunetsintään.



Kuva 4. Google Maps -verkkosovelluksen reittioppaassa verkko on painotettu matkustusajan perusteella. Google Maps näyttää myös ajallisesti kalliimpia vaihtoehtoisia reittejä [5].

Algoritmissa käytettyä karttaa voidaan mallintaa painotetulla verkolla (kuva 5), jossa kaarilla on jokin arvo, joka kuvastaa esimerkiksi etäisyyttä. Se voi tietenkin myös esittää matkaan kuluvaa aikaa tai käytettävää energiaa seuraavaan solmuun kulkiessa. Dijkstran algoritmin huonona puolena on se, että sen käsittelemää verkkoa ei voi painottaa negatiivisesti.



Kuva 5. Painotettu verkko-tietorakenne, jossa vihreät luvut esittävät kaarien painoja. Kirjaimet ovat solmujen tunnuksia.

Dijkstran algoritmi osaa valita aina lyhimmän reitin olemassa olevista vaihtoehdoista. Algoritmissa solmut pitävät sisällään tiedon siitä, kuinka pitkä matka solmuun on ja onko solmussa vierailtu. Kun lyhin kaari on valittu, lukitaan solmu, jossa vierailtiin kaaren kautta ja lisätään sille solmuun kulunut etäisyys. Tietenkin aloitussolmu on lukittu heti alussa, eli sielläkään ei voi enää vierailla algoritmia käsiteltäessä.

Tässä esimerkissä aloitussolmu on solmu A, eli tällä hetkellä vain sen kaaret kuuluvat vertailtavien listaan. Seuraavassa solmussa vierailtua vertailtaviin reitteihin lisätään uudesta solmusta paljastuneet kaaret. Valittu solmu sitten lukitaan, joten sinne vieviä kaaria ei voi enää valita. Vieraillun solmun kaaria vertaillessa niiden etäisyydeksi tulee ottaa huomioon solmuun kulunut matka.

Kuvan 5 esimerkissä aloitussolmusta A on kolme vaihtoehtoa seuraavaksi vierailtavalle solmulle:

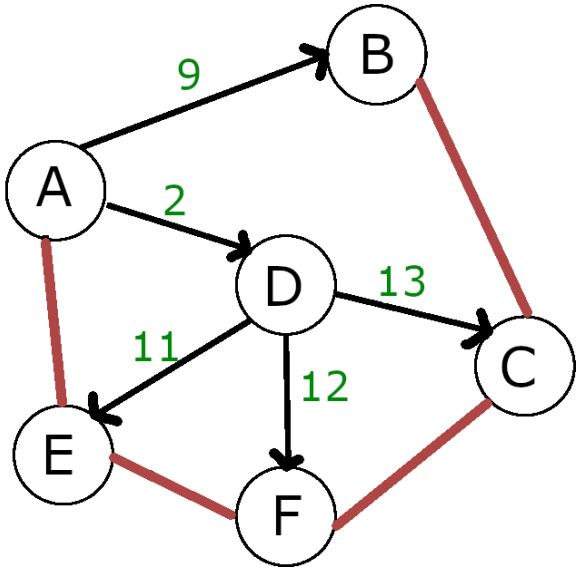
- aloitussolmusta A solmuun B, joiden etäisyys on 9
- aloitussolmusta A solmuun D, joiden etäisyys on 2
- aloitussolmusta A solmuun E, joiden etäisyys on 14.

Huomataan, että lyhin reitti seuraavaan solmuun on aloitussolmusta A solmuun D, joiden etäisyys on 2. Solmu D lukitaan ja otetaan käsiteltäviin kaariin nyt myös D-solmun kaaret. Tämä paljastaa vanhojen kaarien lisäksi uusia vaihtoehtoja:

- aloitussolmusta A solmuun B, joiden etäisyys on 9
- aloitussolmusta A solmuun E, joiden etäisyys on 14
- solmusta D solmuun C, jolloin kokonaisetäisyys on 13
- solmusta D solmuun F, jolloin kokonaisetäisyys on 12
- solmusta D solmuun E, jolloin kokonaisetäisyys on 11.

Tästä huomataan muun muassa, että reitti solmuun E on lyhyempi solmun D kautta, ja mikäli sinne vievä kaari valittaisiin, pitempää reittiä ei koskaan tultaisi käsittelemään. Seuraava askel on silti aloitussolmusta A solmuun B, joka jälleen

kerran avaa uusia vaihtoehtoja. Lopuksi tästä saadaan aikaiseksi tulos, joka on esitetty kuvassa 6.



Kuva 6. Malli lyhimpien etäisyyksien löytämisestä Dijkstran algoritmilla. Punaiset kaaret on suljettu pois, sillä niitä käyttämällä reitti olisi pidempi.

Dijkstran algoritmissa ei suinkaan lopeteta läpikäyntiä, kun kohde on löydetty, vaan sitä jatketaan samaan tapaan, kunnes kaikki solmut ovat vierailtuja. Toisin sanoen, jos Dijkstran algoritmia aikoo soveltaa isossa verkossa, on hyvä katkaista käsittely kohteen löydyttyä, mikäli mikään seuraavista vaihtoehdoista ei ole lyhyempi kuin jo kohteeseen kulunut matka.

Koska Dijkstran algoritmi etsii lyhimmän reitin kaikkiin mahdollisiin solmuihin, algoritmin käyttäminen ajoaikana voi olla hyvinkin hidasta, jos käsiteltävä verkko on suuri. Sellaisessa tilanteessa kannattaa harkita, voiko sen löytämiä arvoja tallentaa etukäteen esimerkiksi tietokantaan tai käyttää suoritustehokkaampaa algoritmia.

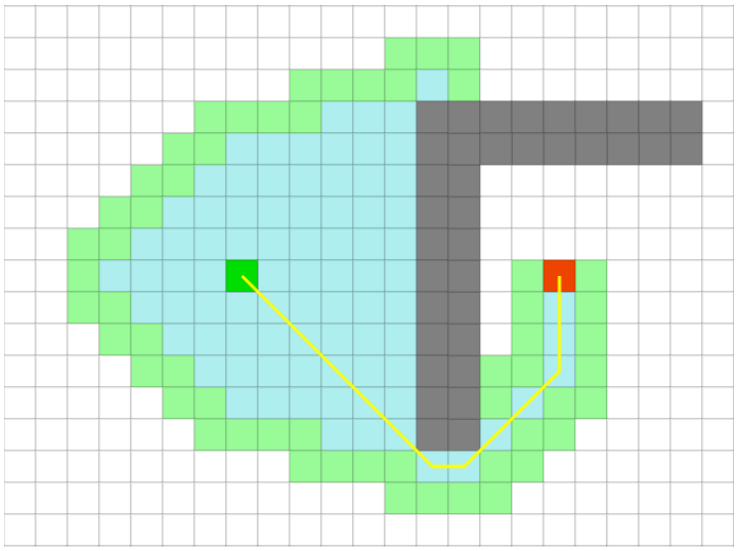
2.3 A*-algoritmi

A*-algoritmi on vuonna 1968 Peter Hartin, Nils Nilssonin ja Bertram Raphaelin julkaisema polunetsintäalgoritmi [6]. Se on yksi parhaita ja käytetyimpiä tekniikoita polunetsinnässä, ja sitä sovelletaan lukemattomissa pelisovelluksissa. Toisin kuin Dijkstran algoritmi, A* on heuristinen algoritmi ja sen tulee tietää kohdesolmusta etukäteen. Sitä voidaan myös soveltaa negatiivisesti painotetussa verkossa, eli solmujen väliset etäisyydet voivat olla positiivisia ja negatiivisia lukuja.

Kuten Dijkstran algoritmissa, A*-algoritmissa pidetään kirjaa vierailluista solmuista. Algoritmissa lasketaan pelkän kuluneen matkan lisäksi myös heuristiikka (kuva 7). Se kuvastaa aloitussolmun ja kohdesolmun etäisyyttä. Käytännössä sen arvo riippuu siitä, mitä heuristiikkaa käytetään. Tämän yhteen lasketun arvon avulla algoritmi valitsee pieniarvoisimman vaihtoehdon. Jos heuristiikkaa ei käytetä, A*-algoritmi käyttäytyy kuin Dijkstran algoritmi. Toisin sanoen Dijkstran algoritmi on A*-algoritmin erikoistapaus [6].

Käytännössä, kuten Dijkstran algoritmissa, allokoidaan seuraavien mahdollisten solmujen etäisyys ja kutsutaan sitä g:ksi. Käsiteltävän solmun etäisyys kohdesolmusta lasketaan valitulla heuristiikalla, ja tätä arvoa kutsutaan h:ksi. Seuraavaksi vertaillaan kaikkien seuraavien mahdollisten solmujen f-arvoa, joka saadaan g- ja h-arvojen summasta.

Heuristiikan tarkoituksena on saada jonkinlainen arvio matkan etäisyydestä, jotta tiedetään, minne kannattaa siirtyä seuraavaksi. Tarkan matkustusetäisyyden laskeminen on raskasta ja vaatii jo itsessään polunetsintää, minkä takia suoritusaikana on paras käyttää etäisyyden arviota. Heuristiikkana voidaan käyttää useita eri vaihtoehtoja, kuten vaikkapa Manhattanin (kuva 8), diagonaalista tai euklidista (kuva 9) etäisyyttä.

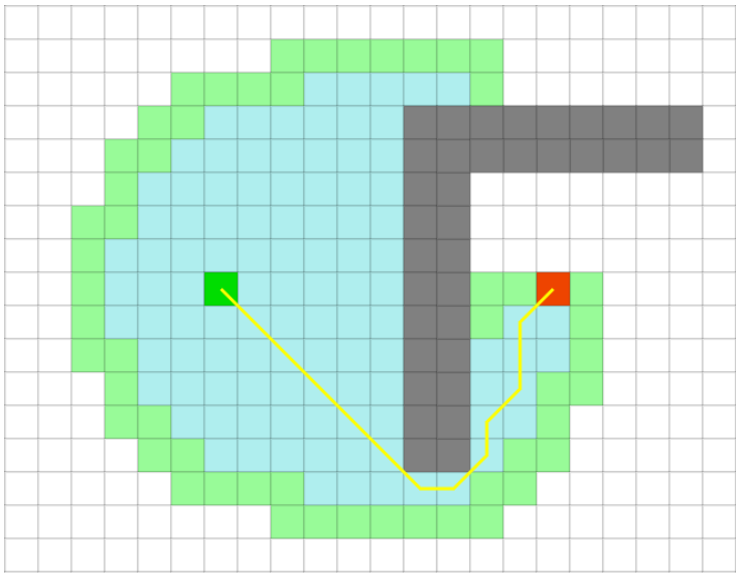


Kuva 7 A*-algoritmin löytämä reitti vihreästä ruudusta punaiseen käyttäen Manhattanin etäisyyttä [7].

Kuvassa 8 nähdään polunetsintäverkkosovelluksen suorittama etsintä. Sinisellä havainnollistetaan käsiteltyjä solmuja ja vaalean vihreällä solmuja, joihin voitaisiin seuraavaksi siirtyä, jos kohde ei olisi jo löytynyt. Tässä esimerkissä käytettiin kaavan 1 havainnollistamaa Manhattanin etäisyyttä.

$$|j_1 - k_1| + |j_2 - k_2| + \cdots + |j_n - k_n| \quad (1)$$

Manhattanin etäisyys [6] on yksinkertaisesti kahden solmun välisen akselin etäisyyden yhteenlaskettu summa. Toisin sanoen kaksiulotteisessa koordinaatistossa olisi mahdollista kulkea vain neljään eri suuntaan.



Kuva 8 A*-algoritmin löytämä reitti vihreästä ruudusta punaiseen käyttäen euklidista etäisyyttä [7].

Kuvista 8 ja 9 huomataan, että heuristiikka ei vaikuta pelkästään itse polunetsinnän suoritukseen, vaan myös sen löytämä reitti on erilainen heuristiikasta riippuen. Kuvassa 9 käytettiin euklidista etäisyyttä. Euklidisessa etäisyydessä [6] käytetään puhtaasti solmujen todellista etäisyyttä linnuntietä kaavan 2 mukaisesti.

$$\sqrt{(j_1 - k_1)^2 + (j_2 - k_2)^2 + \cdots + (j_n - k_n)^2} \qquad (2)$$

Diagonaalisessa etäisyydessä [6] sen sijaan käytetään vain sen akselin erotuksen absoluuttista arvoa, jossa se on suurin.

Dijkstran algoritmia esiteltiin kuvan 5 esimerkillä. Samaisen esimerkin tilanteessa, jos aloitussolmu on A, ovat seuraavina mahdollisina solmuina E, D ja B. A:n etäisyys E:hen on 14, D:hen 2 ja B:hen 9. Nämä ovat kyseisten solmujen g-arvot. Sanotaan, että kohdesolmu on F ja sen etäisyys euklidisesti E:hen on 7, D:hen 10 ja 20. Näistä saadaan kullekin solmulle f-arvo, joista valitaan pienin seuraavaksi solmuksi:

- E-solmun f-arvo on 14 + 7 = 21.

- D-solmun f-arvo on 2 + 10 = 12.
- B-solmun f-arvo on 9 + 20 = 29.

Tästä nähdään, että seuraavaksi siirrytään solmuun D, sillä sen f-arvo on pienin. Tässä tapauksessa Dijkstran algoritmi valitsisi saman solmun seuraavaksi, sillä myös D-solmun g-arvo on pienempi kuin muiden solmujen.

3 Polunetsintä peleissä

3.1 Hierarkkinen tietorakenne

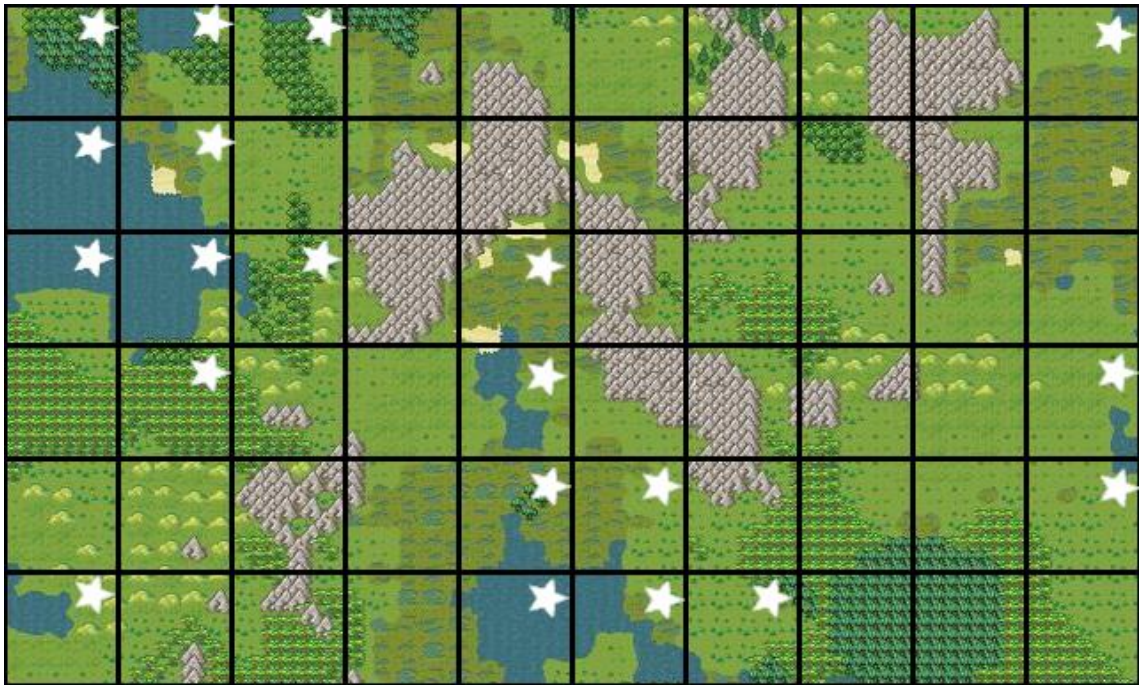
Asioiden etsintä pelikartasta leveyshaun avulla sekä polunetsintä A*-algoritmilla muuttuvat raskaiksi, kun kartan koko kasvaa. Tässä luvussa käsitellään, miten näitä algoritmisia ratkaisuja voitaisiin teoreettisesti soveltaa isommassa kartassa. Tietorakenteiden skaalautuvuutta kannattaa pohtia peleissä, joissa on esimerkiksi avoimen maailman kartta (engl. open world).

Ennen kuin haetaan reittiä jonnekin, pitäisi tietää, minne oikeastaan halutaan. Pelisovelluksessa hahmo voisi esimerkiksi etsiä vettä leveyshaulla ympärillä olevasta maastosta. Tämä ratkaisu jo sellaisenaan voi olla melkoisen nopeaa, jos vettä riittää runsaasti hakupisteestä katsoen. Jos ruokaa ei ole lähelläkään, suorituskyky romahtaa, eikä tämä selvästikään ole tarpeeksi vahva ratkaisu vielä.

Jos oletetaan, että karttaa yleensä käsitellään ruuduittain, kartan vesiruudut voidaan listata pelikartan alustuksessa kerralla ja iteroida vesiruutua valittaessa. Tässä tapauksessa matkan arviointi ja lähimmän tai strategisimman vesiruudun valinta vaatii liikaa suorituskykyä, sillä silloin se iteroi kaikki vesiruudut eli myös ne, jotka ovat liian kaukana lähtöpisteestä. Ratkaisu voisi olla jotain näiden kahden tavan väliltä.

Kun käytetään hierarkkista tietorakennetta, jossa kartta viipaloidaan pienemmiksi tietorakenteiksi tai alueiksi (kuva 10), tiedetään nopeammin, löytyykö alueelta

vettä vai ei. Tällöin karttaa ei käydä läpi leveyshaulla ruutu ruudulta vaan alue alueelta, joten etsinnän määrä vähenee [8].

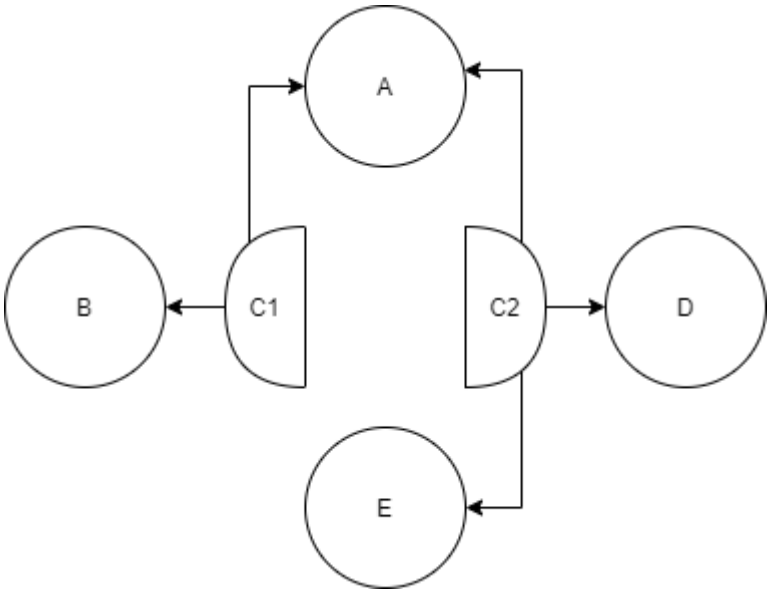


Kuva 9. Visualisointi kartan hierarkkisesta tietorakenteesta. Kartta on pilkottu isompiin paloihin, jotka pitävät kirjaa siitä, onko alueella vettä. Tämä nopeuttaa etsinnän prosessia huomattavasti. Vesiruudut on merkitty valkoisilla tähdillä.

Tällaisen tietorakenteen alue ylläpitää tietoa sisällään olevista asioista sekä tiedon naapurisolmuistaan. Alueen tietoihin voi kuulua sen sisällön lisäksi erilaisia tiloja, esimerkiksi jostakin muuttujasta riippuva vaaratila tai strateginen arvo ja niin edelleen. Sen, minkä tyyppistä tavaraa alueelta löytyy, voi esimerkiksi bittimaskata (engl. bitmask) [9] yhteen muuttujaan, jolloin monivertailu on tehokkaampaa ja voidaan hakea nopeammin aluetta, josta löytyy juuri tietyt asiat yhdellä toiminnolla.

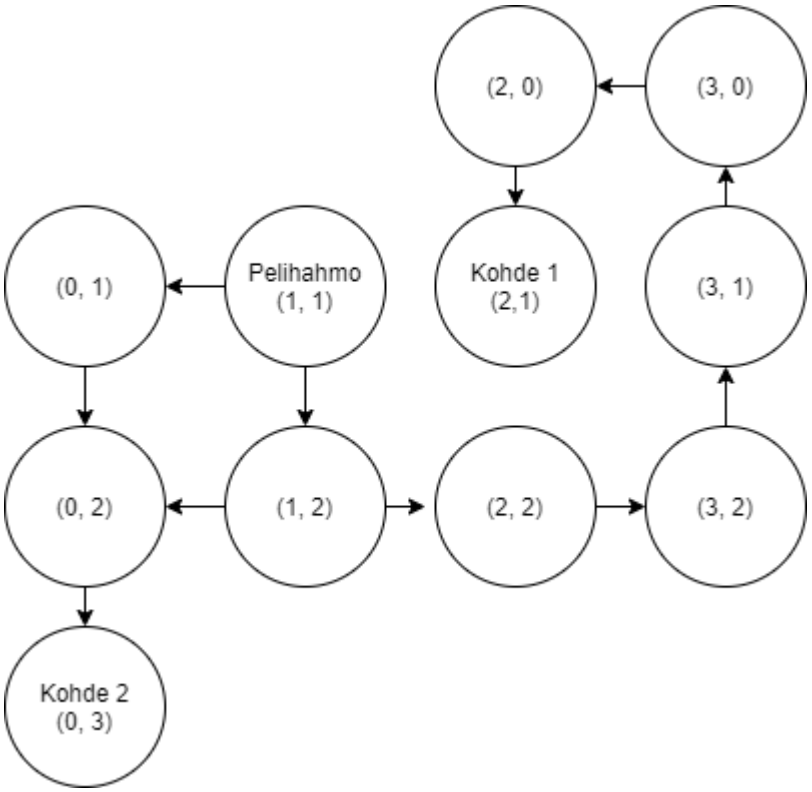
Alue voisi olla linkitetty naapureihinsa kaksisuuntaisesti, mutta voihan olla myös alue, josta pääsee toiseen, mutta ei takaisin. Näiden lisäksi alue sisältää osoittimet myös omiin ruutuihinsa eli alisolmuihin, joista etsityn esineen tai maastotyypin viittauksen voi löytää.

Ennalta määrätyn mallisissa alueissa on ongelma, kun alue leikkaantuu esteiden takia, sillä silloin alueen sisällä ei pysty kulkemaan vapaasti. Sen takia alue on pakko leikata useammaksi alueeksi (kuva 11), jolloin kummallakin uudella alueella on omat naapurisolmut, joihin se on linkitetty. Ne eivät siis ole linkitettyjä keskenään.



Kuva 10. Esimerkki tietorakenteesta, jossa alue on täytynyt jakaa kahteen sen sisällä olevan rajoitteen takia.

Tällainen alueiden muodostuminen esteiden mukaisesti auttaa löytämään halutun kohteen oikean etäisyyden mukaan. Jos kuvitellaan, että kohde olisi seinän takana, voi sinne kulkeminen olla oikeasti huomattavasti pitempi matka kuin kohteeseen, joka on linnuntietä kauempana. Kun kohdetta etsitään alue kerrallaan ja siirrytään leveyshaussa linkitettyihin naapurialueisiin, haku löytää ensin tuon kauempana olevan kohteen (kuva 12), jonne todellisuudessa on vähemmän matkaa.



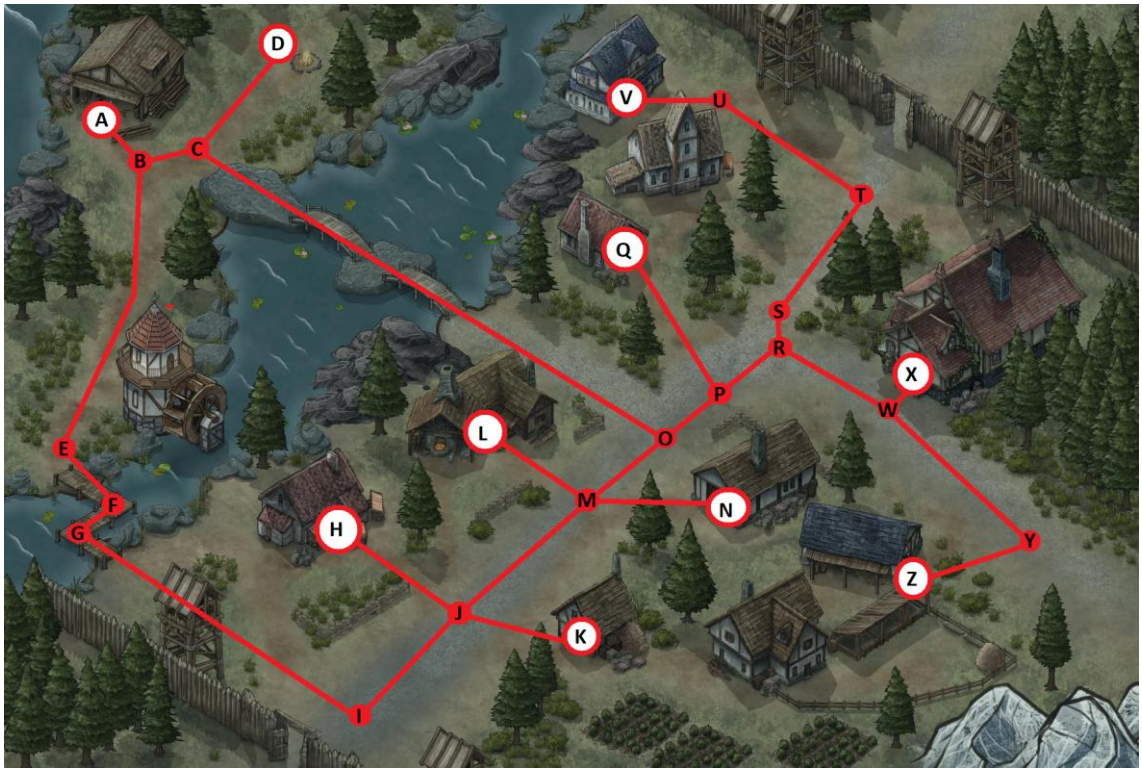
Kuva 11. Pelihahmo löytää kauemman kohteen ensin, sillä sinne on lyhyempi matka, vaikka toinen kohde on koordinaatistossa lähempänä.

Kun tiedetään, minne pelihahmon tulisi mennä, voidaan etsiä sille sopiva reitti. Reitinetsintä voidaan samalla tavalla suorittaa hierarkkisesti, eli voidaan määritellä reitti ensin alueittain, joiden alisolmuilla luodaan lopullinen reitti. Näin saadaan rajattua huomattava määrä ruutuja, joita muutoin olisi saatettu käsitellä suotta. Toinen mielenkiintoinen tapa on käyttää staattiseen karttaan rakennettua välietappien navigaatioverkkoa.

3.2 Navigaatioverkot

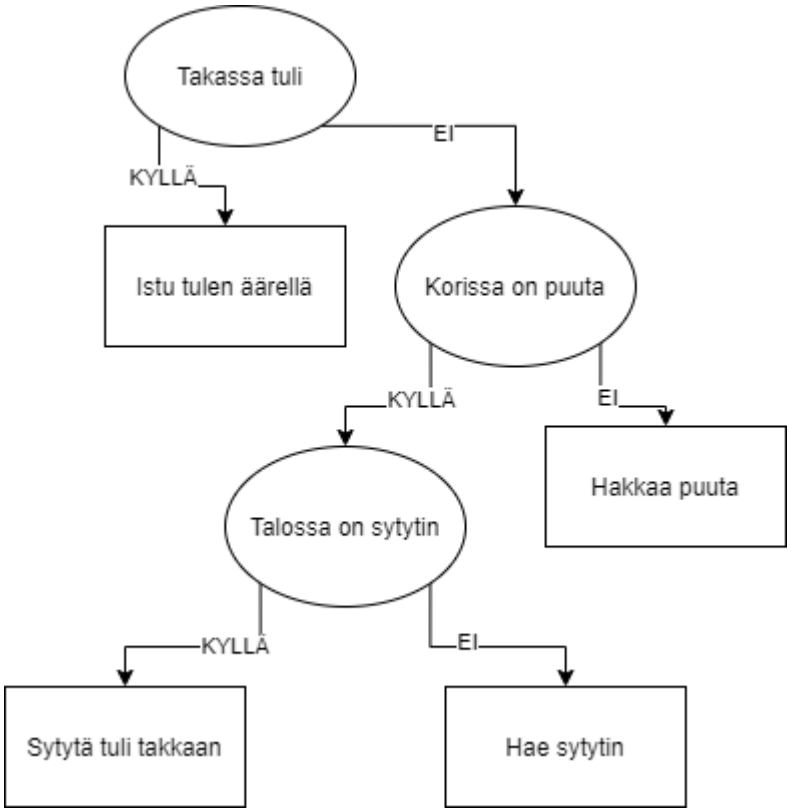
Tekoäly on laaja-alainen tietotekniikan haara, joka koskettaa älylaitteita, jotka pystyvät suoriutumaan tavanomaisesti ihmisälyä vaativista tehtävistä [10]. Tekoälyä käytetään peleissä jatkuvasti simuloimaan muun muassa ihmismäistä käytöstä, jotta saadaan luotua peliin elävän maailman immersiota.

Navigaatioverkkoa (kuva 13) käyttämällä saadaan pelihahmot kävelemään ennalta määriteltyjä reittejä pitkin lähemmäs kohdetta. Hyvänä esimerkkinä toimii Skyrim-pelin suurikokoinen kartta, joka on täynnä erilaisia teitä, polkuja ja mielenkiintoisia sijainteja [11]. Pelissä pelihahmot voivat tätä verkkoa käyttämällä kävellä vaikkapa kaupungista toiseen tietä pitkin. Kaupungin sisällä pelihahmo osaa kävellä ennalta määriteltyjä reittejä pitkin pajalle tekemään töitä, tavernaan istumaan iltaa ja lopulta kotiin nukkumaan.



Kuva 12. Navigaatioverkko, jossa punaiset ympyrät ovat solmuja ja valkotaustaiset ympyrät ovat lehtisolmuja.

Nämä verkon lehtisolmut, kuten pelihahmon talo, ovat sijainteja, joissa pelihahmo voi suorittaa jotain toimenpidettä. Lehtisolmuun päästäkseen ja muissakin solmuissa voi olla erilaisia tarkistuksia kyseiselle pelihahmolle, esimerkiksi onko hahmolla talon avain, osaako hahmo kiivetä ja niin edelleen. Sijainnissa on määritelty useita toimenpiteitä, joissa niissäkin voi olla eri tarkistuksia ja joista voidaan valita suoritettava toimenpide esimerkiksi päätöspuun (kuva 14) avulla.



Kuva 13. Esimerkki pelihahmon päätöspuusta.

Tällaisen navigaatioverkon tulee olla huomattavan kattava, jotta pelihahmojen toiminta ja arjen pyörittäminen olisi luonnollista ja tarpeeksi vakuuttavaa. Skyrim-pelissä hahmot liikkuvat solmujen välillä ympäri karttaa jatkuvasti, jopa silloin kun pelaaja ei ole lähettyvillä. Jos pelaaja ei pysty havaitsemaan jonkin pelihahmon liikettä, on tehokkaampaa vain siirtää hahmo solmusta toiseen ajastimen avulla. Tällöin pelihahmon ei tarvitse oikeasti kävellä kahden pisteen välillä ja säästytään ylimääräiseltä suoritukselta. Näiden kahden pisteen etäisyyden tulisi kuitenkin olla tarpeeksi lyhyt, jos pelaaja saapuukin havainnointietäisyyteen silloin, kun ajastimen mukaan pelihahmon olisi oikeasti tarkoitus olla pisteiden välillä kävelemässä. Siten pelihahmo ei joutuisi aloittamaan kovin pitkää matkaa alusta, mutta tällä kertaa ajastimen sijaan oikeasti kävellen, kun sitä ei voidakaan vain siirtää uuteen pisteeseen, ettei pelaaja näkisi yhtäkkiä tyhjästä ilmaantuvia hahmoja.

3.3 Generoitu ja dynaaminen kartta

Generoidussa kartassa käytettävä tietorakenne luodaan suoritusaikana. Sen takia peleissä, joiden kartat generoidaan, ei yleensä ole yhtä kehittynyttä navigaatioverkkoa generoinnin rajoitteiden takia. Generoitu kartta tuo mukanaan uusia haasteita, kun navigaatioverkkoa ja kartan tietorakennetta ei voi ennalta rakentaa haluamallaan tavalla staattisesti.

Generoidussa kartassa on hyvä ottaa huomioon suljetut alueet, joita saattaa muodostua odottamatta. Jos koetetaan löytää A*-algoritmilla reitti sijaintiin, joka on suljetulla alueella eli sinne ei ole minkäänlaista mahdollista polkua, algoritmi käy läpi kaikki solmut, jonne se vain pääsee. Jotta tällaista performanssin hukuttamista ei tapahtuisi, voidaan antaa suljetun alueen solmuille oma aluetunnus. Tunnuksesta nähdään heti, onko solmuun mahdollista edes päästä, koska nykyisen solmun aluetunnus on eri. Tällaisen tunnuksen voi alustaa ruuduille tulvan omaisella proseduurilla, jolloin se asettaa tunnuksen kaikille saavutettavissa oleville solmuille.

Kun käytetään hierarkkista tietorakennetta kartassa, voidaan nämä eri suljetut alueet määritellä omiksi tietorakenteiksi (kuva 15), jotka eivät ole linkitettyjä toisiinsa. Tätä voisi ajatella niin, että nämä olisivat tavallaan omia karttojaan, vaikka ovatkin samalla kartalla renderöinnin puolesta. Tällöin voidaan verrata kahden solmun välillä niiden tietorakenteiden tunnuksia, eikä tietenkään tarvitse jokaiseen ruutuun sitä siten asettaa.

Kuva 14. Solmut alueessa A ja alueessa B voidaan jakaa omiin tietorakenteisiinsa, kun pelihahmot eivät voi kulkea vuori- ja vesiruutujen lävitse.

Dynaamisessa kartassa, joka voi muuttua suoritusaikana, tulee ottaa huomioon mahdolliset tietorakenteeseen tarvittavat muutokset, kuten uudelleen linkitykset, useamman alueen tietorakenteen yhdistäminen ja myös jakaantuminen.

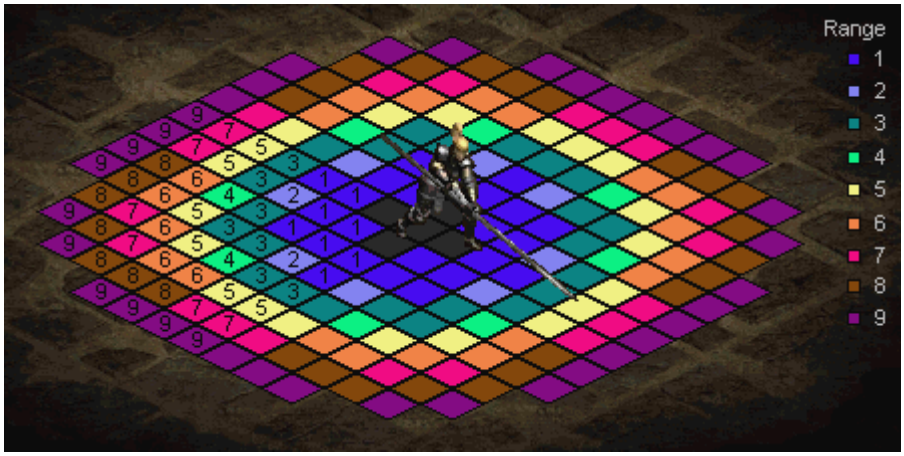
Pelissä, jossa voidaan rakentaa seinä, voidaan estää kulku alueen lävitse. Tämä tarkoittaa sitä, että pitää jakaa alue suoritusaikana. Jos tietorakenteen alueet on linkitetty osoittimina, voidaan kätevästi luoda uusi aluesolmu vanhan lisäksi ja tarkistaa näiden kahden solmun linkitykset ja sisältö. Jos tämän seurauksena muodostuu kaksi suljettua aluetta, ne olisi hyvä jakaa kahteen eri tietorakenteeseen, jotta performanssi säästyy jatkossa niiden välisten solmujen reitinetsinnässä.

Seinän murruttua kaksi tietorakennetta tuodaan takaisin yhteen. Toisin sanoen siihen tietorakenteeseen, jolla on enemmän aluesolmuja, linkitetään pienemmän tietorakenteen alueet. Tämä tapahtuu linkitetyllä järjestelmällä helposti, sillä se ei vaadi muuta kuin näiden kahden kosketukseen joutuneen alueen linkittämistä toisiinsa. Pienemmän tietorakenteen voi poistaa muistista, sillä siinä olleiden alueiden osoittimet ovat nyt tallessa toisessa tietorakenteessa.

Tässä kohtaa voitaisiin ajatella, että eikö ole aika raskasta käydä tietorakenteiden alueita lävitse joka kerta, kun karttaa muutetaan, mutta hyöty on huomattavasti suurempi, sillä pelihahmot saattavat koettaa löytää reittiä useita kertoja sekunnissa. Tämä tuhoaa pelin suorituskyvyn täysin aivan turhaan, jos pelihahmo jää toistuvasti etsimään samaa ruutua, jonne ei voi mitenkään päästä. Toisin sanoen käsittelemällä tietorakennetta muutosten tapahtuessa säästytään muun pelin aikana tapahtuvalta käsittelyltä.

3.4 Ruutupohjaisuus

Esimerkiksi Starcraftissä ja Diablo 2:sessa perinteinen ruutupohjainen liikkuminen on viety astetta pidemmälle, jotta luodaan vaikutelma vapaasta liikkeestä. Tekstuureiden renderöintiin tarkoitettu ruudukko on pilkottu vielä pienemmäksi ruudukoksi (kuva 16). Tällöin voidaan sanoa, että polunetsintäkartan resoluutiota on nostettu [12] ja liikkuminen näyttää vapaammalta. Joka tapauksessa ruutupohjainen liikkuminen rajoittaa liikkumissuuntia, mikä ei itsessään haittaa tekoälyn ohjatessa hahmoja. Tämä tapa on täydellinen Point and Click -peleihin, joissa klikataan sijaintia, jonne pelaajahahmo siirtyy.



Kuva 15. Diablo 2 -pelissä ruudukkoa on vaikea huomata kunnolla ilman sen havainnollistamista, sillä ruudukolla on korkea resoluutio pelin hahmoihin verrattuna [13].

Kun pelaaja ohjaa hahmoaan sauvaohjaimella, pelaaja pystyy syöttämään tiedon pelille haluamastaan suunnasta hyvin vapaasti. Ruutupohjaisessa liikkumisessa pelihahmo napsahtaa lähimpänä olevaan suuntaan esimerkiksi 45 tai 90 asteen välillä, vaikka pelaaja pystyy antamaan suunnan paljon tarkemmin. Jähmeän oloiset kontrollit vaikuttavat siltä, että peli ei reagoi pelaajan syöttämään tietoon oikein ja antaa melko vanhahtavan kuvan pelin toiminnallisuudesta. Esimerkiksi vuonna 1986 julkaistussa [14] The Legend of Zelda -pelissä on vain neljä liikkumissuuntaa, vaikka senaikaisilla ohjaimilla olisi voinut toteuttaa kahdeksan suuntaa. Koska peliprojekti, jonka päälle insinöörityö toteutetaan, on suunniteltu sauvaohjaimella pelattavaksi, sille täytyy löytää responsiivisempi ratkaisu.

3.5 Vapaasti liikkuminen

Polunetsinnän toteuttaminen pelisovellukseen vaikuttaa pintapuolisesti yksinkertaiselta ruudukossa. Ruudukossa on helppo määritellä, mitkä ruudut ovat avoimia tai suljettuja eli mihin ruutuun pääsee ja mihin ei. Tilanne vaikeutuu nopeasti, kun kartalla liikkuminen on vapaata eli kun entiteettien sijainteja ja liikkumista ei ole rajoitettu ruudukon mukaisesti. Lisähaastetta ilmaantuu nopeasti myös, jos kartalla ilmenevät entiteetit voivat olla erimuotoisia keskenään, jolloin joudutaan tarkistamaan eri entiteettien törmäyksiä, jotta voidaan estää toisten entiteettien sisälle liikkuminen. Tämä tuo huomattavan määrän lisää pohdittavaa tehokkuuden kannalta.

Kun pelin kartassa on niin sanotusti rajaamaton liikkuminen, peleissä käytetään ruudukkoa muistuttavaa tietorakennetta taustalla. Tietorakennetta voidaan käyttää datan säilyttämiseen ja polunetsinnän avustamiseen. Vaikka liikkuminen on vapaalla kartalla täysin rajoittamatonta, tietorakenteen ruutupohjainen käsittelytapa auttaa polunetsinnässä.

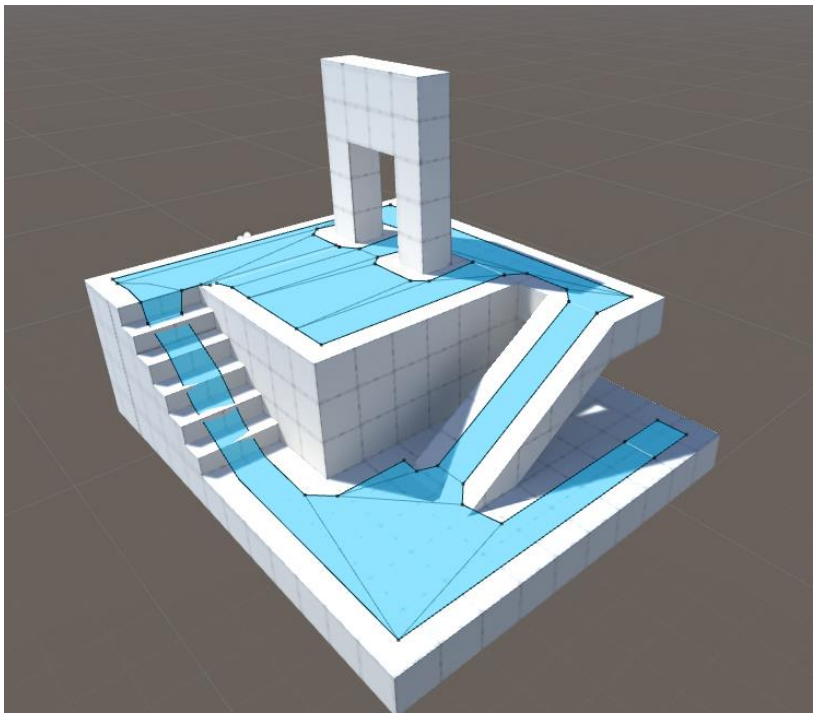
Jos esteitä voi ilmaantua missä tahansa sijainnissa ja niiden muoto on epämääräinen, tulee esiin ongelma, jossa kahden ruudun välinen polku on katkaistu, vaikka itse ruudut vaikuttavat avoimilta liikkumiselle. Koska tällaisessa vapaassa

kartassa ruudut eivät ole yksinkertaisesti suljettuja tai avoimia, liikkuminen vaatii törmäyksen tarkistuksen ympärillä olevien esineiden varalta.

Ruudun avoimuuden määrittäminen vaikeutuu jälleen, kun pitää ottaa huomioon myös polkua kulkevan hahmon muoto. Tämä tarkoittaa sitä, että kyseisen ruudun keskipiste voi olla avoin pienelle hahmolle, joka ei osu ympärillä oleviin esteisiin, mutta isompi hahmo saattaisi osua.

Peleissä jo vitsin omainen stereotypia on, kun hahmo jää jumiin esteeseen ja rupeaa kävelemään paikallaan. Tämä johtuu useasti siitä, että polunetsintäalgoritmi luulee, että hahmo mahtuu kävelemään kohdasta, josta se ei oikeastaan mahdukaan. Tällöin pelihahmo jää kävelemään kohti pistettä, jonne sen on sellaista reittiä pitkin mahdoton päästä, sillä viime kädessä pelin fysiikkamoottori estää hahmon kulkemisen.

Kun hahmo jää jumiin polkua seuratessa, voidaan väkisin pakottaa hahmo esteen läpi tai nykäistä sitä parempaan sijaintiin, josta se pystyy jatkamaan polkua paremmin. Se ei korjaa itse ongelmaa vaan sen seurauksia, minkä takia oletusarvona tämä ratkaisu ei ole hyvä, ja se rikkoo huomattavasti tekoälyn uskottavuutta. Hahmojen jumiutumisen estämiseksi voisi tehdä törmäystarkistuksia polunetsintäalgoritmia käsitellessä tai luomalla kartalle navigaatiomallin (engl. navmesh) (kuva 17).



Kuva 16. Kolmiulotteiseen näkymään on luotu navigaatiomalli, joka on kuvassa sininen.

Navigaatiomalli on tietorakenne, joka koostuu kokoelmasta monikulmioita [15]. Monikulmiot kattavat sen alueen, jossa pelihahmojen on mahdollista kävellä. Näin voidaan nopeasti rajoittaa käsiteltävän alueen kokoa polunetsinnän suhteen, kun pelissä kävelemiseen tarkoitettu alue on staattisesti määritelty.

4 Polunetsinnän toteutus

4.1 Spesifikaatiot

Peli (kuva 18), jolle insinöörityö toteutettiin, on kaksiulotteinen ja sitä toteutetaan pelattavaksi PC:lle ja konsolialustoille. Se käyttää top-down-perspektiiviä eli on niin sanotusti ylhäältä kuvattu. Pelin tärkein mekaniikka on taistella hirviöitä vastaan. Pelissä pystyy muun muassa juoksemaan, syöksymään, lyömään miekalla ja ampumaan projektiileja. Pelihahmoa on tarkoitus ohjata sauvaohjaimella. Lisäksi pelissä on ominaisuudet NPC-hahmojen (Non-Player Character, tekoälyn ohjaama pelihahmo) kanssa dialogin käymistä varten.

Kuva 17. Peli, jolle insinöörityö toteutettiin.

Pelissä on erilaisia esteitä, esineitä ja pelihahmoja. Niillä esiintyy törmäysmuotoja, joita vertailemalla toisiinsa voidaan tarkistaa kahden entiteetin törmäys. Koska peli on kaksiulotteinen, myös törmäysmuodot voidaan yksinkertaistaa kaksiulotteisiksi. Törmäysmuoto (engl. mesh collider) voi olla monikulmio tai ympyrä.

Pelikartalla liikkuminen on vapaata, mutta kartta rakennetaan 16 x 16 -kokoisten ruutujen pohjalta, eli esineet ja ruutujen tekstuurit napsautetaan ruudukon mukaisesti.

Insinöörityössä määriteltiin toteutettavaksi tämän pelin polunetsintäominaisuus NPC-hahmoille. Tarkoituksena oli mahdollistaa muun muassa vihollisten sulava kulkeminen erilaisten esteiden lomassa. Työn toteutuksessa tuli ottaa huomioon kartan vapaus ja esineiden törmäysmuodot (kuva 19), eli polunetsintää ei voisi toteuttaa täysin ruudukkoon luottaen.

Hahmot eivät saa kulkea niille vihamielisten hahmojen tai esineiden päältä, joilla on törmäysmuoto. Hahmot eivät myöskään saa kulkea tiettyjen ruudukon tekstuurien päältä. Polunetsintäominaisuus ei saa vaatia liikaa suorituskykyä tai jumittaa peliä.

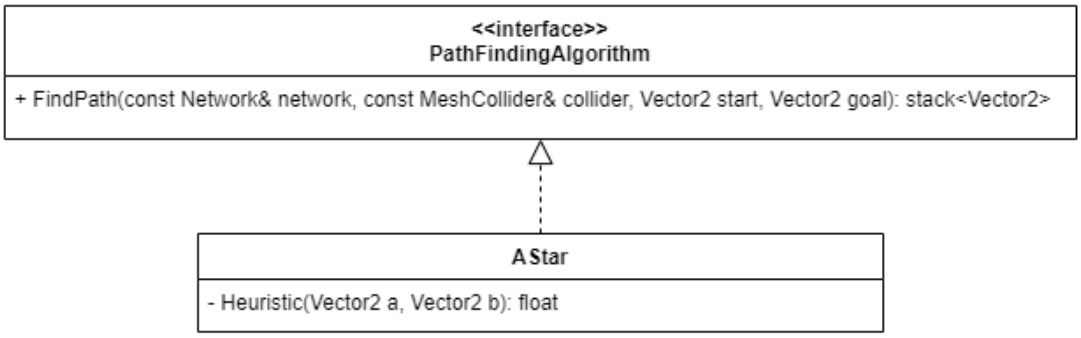
Kuva 18. Visualisointi entiteettien kaksiulotteisista törmäysmuodoista. Kuvassa pelihahmojen törmäysmuoto on ympyrä ja esineillä monikulmio.

Pelihahmoille tarvittiin vähintään kolme eri kokoa törmäysympyröille, pieni, iso ja keskikokoinen. Pientä kokoa käyttävät kaikki tavanomaisen kokoiset hahmot, kuten pelaajahahmo, ihmishahmot ja pienet viholliset. Keskikokoista törmäysympyrää käyttävät hieman isommat hahmot, kuten miniboss-viholliset, jotka voivat esimerkiksi johtaa pienempien vihollisten parvea. Isointa törmäysympyrää käyttävät esimerkiksi kenttien lopuksi vastaan tulevat boss-hahmot.

Peliin oli jo toteutettu törmäystarkastukset erimallisten muotojen välillä, jota fysiikkamoottori käyttää hyödyksi liikkumisen tarkastamiseksi. Fysiikkamoottori siis estää esineiden läpi liikkumisen. Pelissä on jokaisella NPC-hahmolla oma tekoälyluokan ilmentymä, joka pitää sisällään tiedon siitä, mihin sijaintiin hahmo milloinkin haluaa päästä.

4.2 Suunnittelu

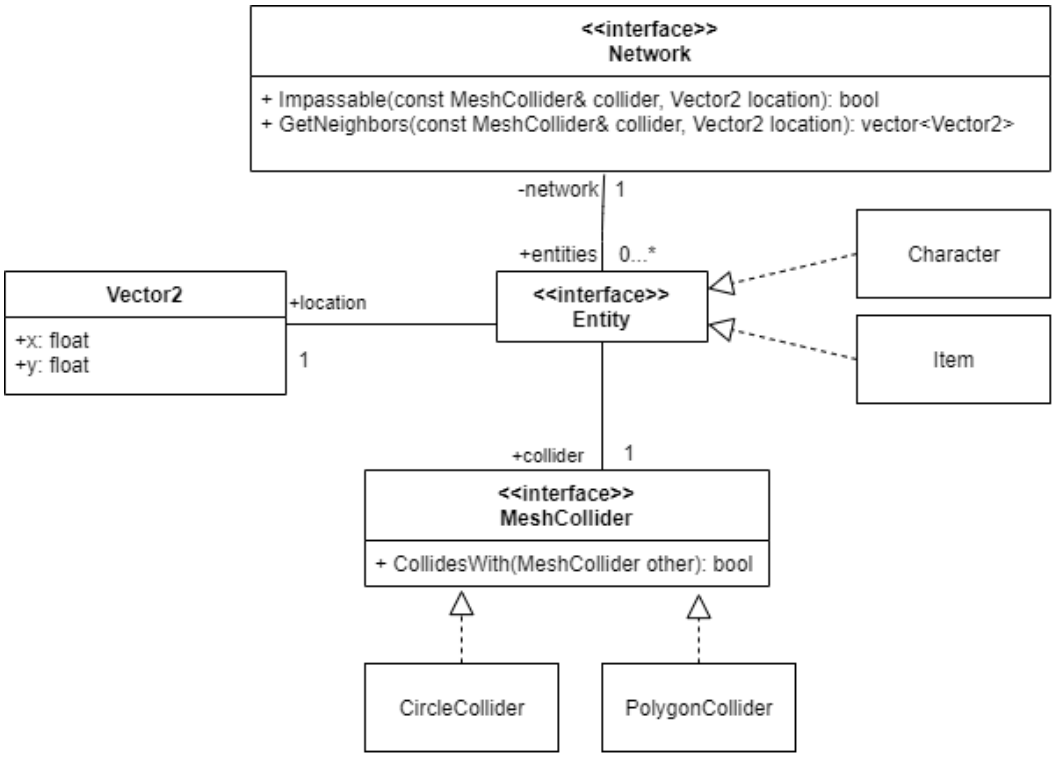
Insinöörityöprojekti päätettiin aloittaa tuottamalla mahdollisimman selkeä ja yksinkertainen rakenne, jonka pohjalta projektia lähdettiin toteuttamaan. Polunetsintäalgoritmeille luotiin rajapinta *PathfindingAlgorithm* (kuva 20). Rajapinnan tuli toteuttaa julkinen *FindPath*-funktio, joka vaatii parametriksi käsiteltävän verkon, törmäysmuodon, jolla polunetsintä halutaan toteuttaa, sekä lähtö- ja kohdesijainti. FindPath-funktio palauttaa reitin eli pinon verkon sijainneista, joita seuraamalla hahmo pääsee kulkemaan kohteeseen nykyisessä maailman tilassa.



Kuva 19. UML-kaavio polunetsintäalgoritmin rajapinnasta ja sen toteutuksesta.

Rajapintaa toteuttamaan valittiin A*-algoritmi, jolle tehty luokka toteuttaa aiemmin suunnitellun rajapinnan. Luokka pitää sisällään yksityisen funktion, joka palauttaa heuristiikan. Tätä metodia tullaan käyttämään apufunktiona polunetsintämetodissa. Tässä projektissa päätettiin, että yksi kovakoodattu heuristiikka on riittävä toteutus, joten heuristiikkaa ei lähdetty abstrahoimaan sen enempää.

Verkolle suunniteltiin oma rajapinta (kuva 21), jonka tulee toteuttaa julkinen tarkistusmetodi solmun tilasta ja viereisten solmujen hakeminen. Tarkistusmetodia käytetään hyödyksi viereisten solmujen hakemisen metodista, joka on myös julkinen. Tämä metodi palauttaa vektorin viereisistä solmuista, jotka ovat avoinna. Kumpikin metodi käyttää apunaan parametrina saatua törmäysmuotoa, jonka avulla saadaan selvitettyä solmun avoimuus. Tämän rajapinnan toteutuksen tulee pitää sisällään vektori entiteeteistä, joita verkossa ilmenee.



Kuva 20. UML-kaavio tietorakenteesta, jossa polunetsintää suoritetaan. Rajapintojen toteutukset ovat esimerkkejä.

Entiteetin rajapinta koostuu sen sijainnista ja törmäysmuodosta. Näin verkon metodit voivat iteroida läpi sieltä löytyvät törmäysmuodot.

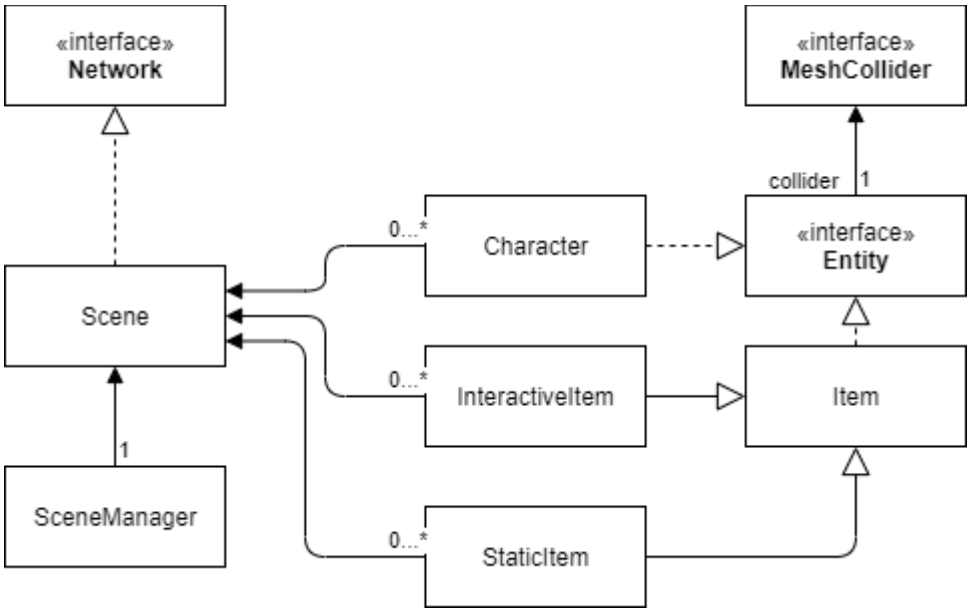
4.3 Toteutus

Insinöörityön projektiosuus toteutettiin siten, että pelissä käytettävät kartat jaettiin omiin näkymiin (engl. scene) itsenäisiksi tietorakenteiksi, jotka pitävät yllä dataa, johon lukeutuvat kartan tekstuurit, staattiset esineet, interaktiiviset esineet ja pelin hahmot. Staattiset esineet ovat esineitä, joita ei tarvita muuhun kuin renderöintiin ja törmäystarkistukseen. Interaktiiviset esineet ovat esineitä, joita iteroidaan myös muihin käyttötarkoituksiin, kuten ovet, teleportit, ansat ja niin edelleen. Staattiset ja interaktiiviset esineet sekä hahmot kaikki perivät Entity-rajapinnan. Toteutusvaiheessa huomattiin, että Entity-rajapintaa ei tarvinnut mainita suunnitelluissa rajapinnoissa tai polunetsintäalgoritmin toteutuksessa. Se, mistä

MeshCollider-rajapinta haetaan algoritmille ja kuinka Network-rajapinta tarkastaa törmäyksiä, jätettiin täysin toteuttavan luokan vastuulle.

Kartan data ladataan pelin käynnistyessä tiedostosta, johon on pakattu kaikki kartan staattinen tieto. Kartan ruutupohjaiset tekstuurit on alustettu kaksiulotteiseen taulukkoon, josta niitä voi hakea käyttämällä ruudukon koordinaatteja. Hahmoja ja interaktiivisia esineitä käsitellessä iteroidaan ylläpidettyä listaa, josta siivilöidään vaikutusalueelle sijoittuvat entiteetit.

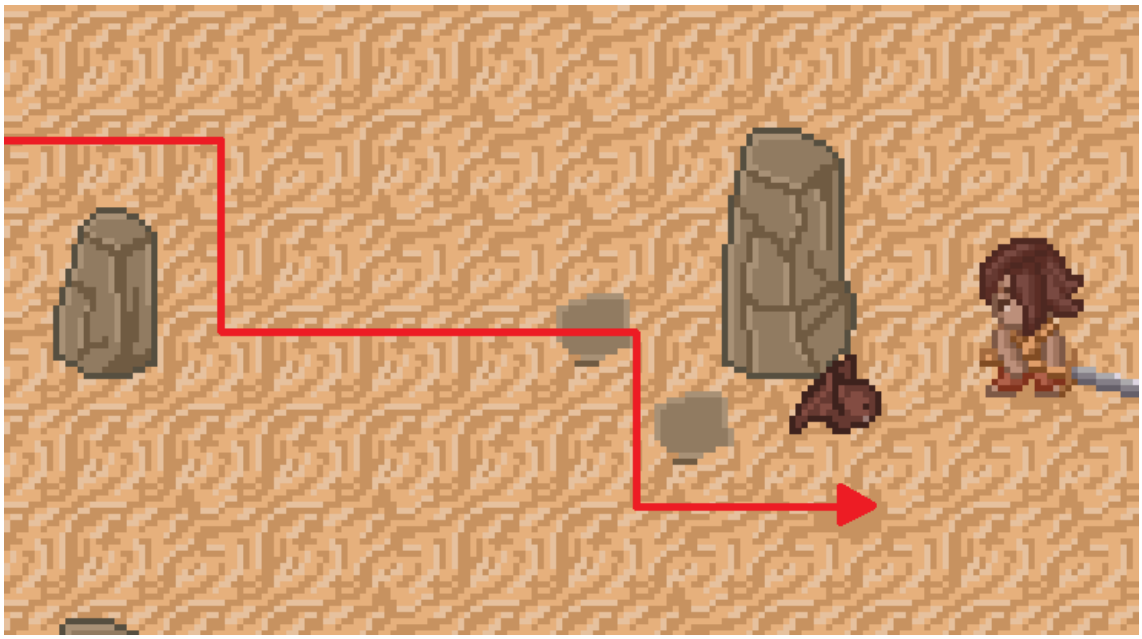
Karttojen tietorakenteeksi toteutettiin yksinkertainen ja helppo malli, sillä pelin kartta ei ole avoin maailma ja sen sai pilkottua siististi osiin. Näkymien käyttämä tietorakenne pysyy nykyisessä käyttötarkoituksessa niin pienenä, että se ei nykyisellään tarvitse suorituskyvyn puolesta optimointia. Näkymiä hallitsee ja päivittää *SceneManager*-luokka, joka tarvittaessa vaihtaa aktiivisen näkymän toiseen näkymään. Kuvassa 22 esitellään toteutuksen rakennetta.



Kuva 21. UML-kaavio toteutuksen rakenteesta.

Näkymien välillä pääsee kulkemaan kartoilla löytyvien interaktiivisten esineiden avulla, ja kun pelaajahahmo astuu esineen päälle, näkymä vaihtuu ja ladataan toinen tietorakenne, jonne pelaaja sijoitetaan.

Tekoälyn ohjaamien hahmojen reitinetsintä toteutettiin A*-polunetsintäalgoritmin avulla, ja se käyttää heuristiikkana Manhattanin etäisyyttä. Reitinetsintä käyttää apunaan ruudukkoa, jossa yksi ruutu on kooltaan 16 x 16 yksikköä. Reitinetsintäalgoritmin hakiessa viereisiä solmuja algoritmi suodattaa ne solmut, joiden keskelle luotu ympyrä leikkaa lähellä olevan esineen törmäysverkon kanssa. Algoritmin toiminnallisuutta esitellään kuvassa 23.



Kuva 22. Visualisointi NPC-hahmon kulkemasta reitistä kohti pelaajahahmoa.

4.4 Suorituskyky ja optimointi

Toteutuksen aikana todettiin, että NPC-hahmojen reittiin ei läheskään aina tarvita polunetsintää, sillä kyseisessä pelissä hyvin usein kuljetulla reitillä ei ole esteitä ja kohteeseen voi kulkea suoraan kohti. Pelihahmojen kulkemista optimointiin tästä syystä niin, että pelihahmo käyttää polunetsintäalgoritmia vain, kun se osuu johonkin esteeseen.

Jos hahmo ei pysty seuraamaan polunetsintäalgoritmin luomaa reittiä ja saapuukin törmäykseen entiteetin, esimerkiksi toisen hahmon, kanssa, hahmo koettaa löytää uuden reitin. Polunetsinnälle on asetettu muutaman sekunnin aikaväli, jotta peli ei jumita, jos hahmo ei löydäkään uutta reittiä.

Toteutusvaiheessa huomattiin, että suoritusaika on välillä melko pitkä, jos polunetsintäalgoritmi ei löydä kohdetta. Tällöin suoritusaika saattaa nousta noin 60 millisekuntiin, sillä algoritmi käy läpi kaikki mahdolliset solmut, joihin sen on mahdollista päästä. Tämä tarkoittaa sitä, että algoritmi tekee todella paljon törmäystarkastuksia kartalla, vaikka kartta onkin pieni. Tämä ongelma ratkaistiin siten, että asetettiin polunetsintäalgoritmille enimmäissyvyys, jonka saavutettua algoritmin suoritus lopetetaan ja palautetaan tyhjä pino eli polkua ei löytynyt.

Toteutetulla ratkaisulla saatiin näytteen perusteella polunetsinnän suoritusajan keskiarvoksi noin 6,35 millisekuntia, keskihajonnaksi noin 8,55 millisekuntia. Virhemarginaali on noin 2,2. Yhden syvyysaskeleen suoritusajan keskiarvoksi saatiin noin 0,26 millisekuntia. Otteen lyhin suoritusaika oli noin 0,43 millisekuntia, kun taas pisin oli noin 28 millisekuntia. Huomioitavaa on, että saman ratkaisun, jossa käytetään Dijkstran algoritmia A*-algoritmin sijaan, näytteen suoritusajan keskiarvo on noin 23,3 millisekuntia.

Suorituskykyä saataisiin vielä nopeammaksi hierarkkisella tietorakenteella sekä navigaatiomallilla, jolloin algoritmin ei tarvitse tehdä yhtä paljoa törmäystarkistuksia esimerkiksi staattisten esineiden kanssa, sillä ne on jo otettu huomioon navigaatiomallissa.

4.5 Loppuanalyysi

Toteutettu polunetsintäalgoritmi toimii tarkoituksenmukaisesti suurimman osan ajasta, mutta erityisen vaikean maaston lomassa pelihahmo saattaa jäädä jumiin, jos polunetsintäalgoritmi aloitetaan solmusta, joka on suljettu. Käytännössä tämä voi olla tilanne, jossa pelihahmo on sijoittunut kahden solmun välille, joista valitaan yksi solmu lähtösolmuksi, mutta tässä solmussa onkin jokin este. Tämän ongelman korjaamiseksi toteutus tarvitsee vielä hiomista tai pelikarttojen suunnittelua polunetsintäalgoritmin rajoitteet mielessä. Ongelma johtuu hyvin pitkälti polunetsintäverkon 16 x 16 -resoluutiosta, jota käytännössä voisi suurentaa, mutta tämä vaikuttaa suorituskykyyn huomattavasti.

Jumittumisongelman ratkaisemiseksi toteutetaan peli myöhemmässä vaiheessa

todennäköisesti navigaatiomallilla, jolloin polunetsintäalgoritmin käyttämä tietorakenne koostuisi vain niistä solmuista ja kaarista, joiden välillä hahmo varmasti pystyy liikkumaan. Koska hahmojen eri koot tarvitsevat oman navigaatiomallinsa, ko'oille on luotu standardit, joissa törmäystarkistukseen käytetyn ympyrän halkaisijat ovat 8, 16 ja suurin 32 yksikköä. Tällöin navigaatioverkon saa rakennettua vähemmän kömpelöksi ja pelihahmo jäisi epätodennäköisemmin jumiin.

5 Yhteenveto

Polku on sellainen solmujen sarja verkkotietorakenteessa, jonka avulla voidaan siirtyä aloitussolmusta kohdesolmuun. Polun luomiseen voidaan käyttää polunetsintäalgoritmeja. Syvyys- ja leveyshaku ovat etsintäalgoritmeja, joita voidaan käyttää kohdesolmujen etsimiseen. Leveyshaku on yleensä parempi vaihtoehto, koska se löytää lähellä olevat kohdesolmut todennäköisesti nopeammin syvässä tietorakenteessa.

Dijkstran algoritmi on polunetsintäalgoritmi, joka löytää lyhimmän reitin kaikkiin solmuihin, kun taas A* käyttää heuristiikkaa päästäkseen kohdesolmuun nopeammin. Työssä huomattiin, että A* on suoritustehokkaampi ratkaisu, sillä sen ei yleensä tarvitse käydä koko tietorakennetta läpi.

Hierarkkinen tietorakenne tekee hausta nopeampaa ja siten mahdollistaa isommatkin pelikartat. Isossa tai monimutkaisessa kartassa navigaatioverkot säästävät suoritusta ja polut ovat suunnitellumpia, sillä se voidaan virittää siten, ettei vastaan tule suurempia esteitä.

Kartan resoluutio vaikuttaa suorituskykyyn huomattavasti, mutta tekee myös törmäystarkistuksista tarkempia.

Insinöörityörojektissa toteutettiin kaksiulotteiseen peliin polunetsintä niin, että luotiin rajapintoja, joiden toteutuksia käytetään polunetsintäalgoritmin toteutuksessa. Rajapintojen toteutukset määrittelevät muun muassa, millaisia tör-

mäysmuodot ovat, kuinka ne toimivat ja osuvatko ne toisiinsa. Siten itse algoritmia ei kiinnosta, miten törmäystarkistuksia tehdään tai millainen peli ylipäätänsä on. Se tekee vain sitä, mitä sen pitää, eli etsii polun.

Ilmeni, että polunetsintä toimii suurimman osan ajasta, mutta on erikoistapauksia, joissa se voi jäädä jumiin. Lisäksi ilmeni, että jos kohde on sijainnissa, jonne hahmo ei voi päästä, polunetsintäalgoritmi käy läpi kaikki mahdolliset solmut, minkä takia suoritusaika kasvaa huomattavasti. Selvisi, että polunetsinnän tehokkuus ja tarkkuus paranisi navigaatiomallin avulla.

Insinöörityössä päästiin tavoitteisiin, mutta selkeästikin erikoistapauksia ja optimointia varten vaaditaan vielä lisää työtä toteutuksen laadun parantamiseksi.

Lähteet

1 Graph Data Structure. Verkkoaineisto. Programiz. <https://www.programiz.com/dsa/graph> Luettu 10.4.2021.

2 Depth First Search or DFS for a graph. 2021. Verkkoaineisto. GeeksforGeeks. <https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/> Päivitetty 18.3.2021. Luettu 10.4.2021.

3 Breadth First Search or BFS for a Graph. 2020. Verkkoaineisto. GeeksforGeeks. <https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/> Päivitetty 4.12.2020. Luettu 10.4.2021.

4 Makrygiannakis, Ioannis. 2020. Dijkstra's Algorithm Java Example. Verkkoaineisto. Java Code Geeks. <https://examples.javacodegeeks.com/dijkstras-algorithm-java-example/> 9.4.2020. Luettu 10.4.2021.

5 Google Maps. 2021. Verkkoaineisto. Google. <https://www.google.com/maps> Luettu 10.4.2021.

6 A* Search Algorithm. 2021. Verkkoaineisto. GeeksforGeeks. <https://www.geeksforgeeks.org/a-search-algorithm/> Päivitetty 1.2.2021. Luettu 10.4.2021.

7 Pathfinding.js. Verkkoaineisto. Github. <https://qiao.github.io/PathFinding.js/visual/> Luettu 10.4.2021.

8 Sylvester, Tynan. 2014. RimWorld Technology – Region System. Verkkoaineisto. Youtube. <https://www.youtube.com/watch?v=RMBQn_sg7DA> 1.6.2014. Luettu 10.4.2021.

9 Ahmed, Shakib. 2019. The Art of Bit Masking. Verkkoaineisto. Bits and Pieces. <https://blog.bitsrc.io/the-art-of-bitmasking-ec58ab1b4c03> 22.4.2019. Luettu 10.4.2021.

10 How does artificial intelligence work? Verkkoaineisto. Builtin. <https://builtin.com/artificial-intelligence> Luettu 10.4.2021.

11 Skyrim. Verkkoaineisto. Bethesda. <https://elderscrolls.bethesda.net/en/skyrim> Luettu 10.4.2021.

12 Wyatt, Patrick. 2013. The StarCraft path-finding hack. Verkkoaineisto. Code of Honor. <https://www.codeofhonor.com/blog/the-starcraft-path-finding-hack> 20.2.2013. Luettu 10.4.2021.

13 Character collision box shape is not like original design. 2020. Verkkoaineisto. Github. <https://github.com/mofr/Diablerie/issues/88#issuecomment-573637861> 13.1.2020. Luettu 10.4.2021.

14 Valkola, Johannes. 2021. Elämää suurempi seikkailu. Verkkoaineisto. Helsingin Sanomat. <https://www.hs.fi/nyt/art-2000007810983.html> Päivitetty 18.2.2021. Luettu 10.4.2021.

15 Navigation Meshes and Pathfinding. 2018. Verkkoaineisto. gamedev.net. <https://www.gamedev.net/tutorials/programming/artificial-intelligence/navigation-meshes-and-pathfinding-r4880/> 27.4.2018. Luettu 10.4.2021.