Pelin algoritmit

Ryhmä no. 2

Tuomas Kyttä H3408

Arttu Heinonen H3425

Joonas Katainen H3248

Joulukuu 2014

Ohjelmistotekniikan koulutusohjelma

Tekniikan ja liikenteen ala

Sisältö

[1 Johdanto 2](#_Toc405392830)

[2 Algoritmin tehokkuus & erityispiirteet 2](#_Toc405392831)

[3 Toteutus 5](#_Toc405392832)

[4 Testiaineiston generointi ja testaus 5](#_Toc405392833)

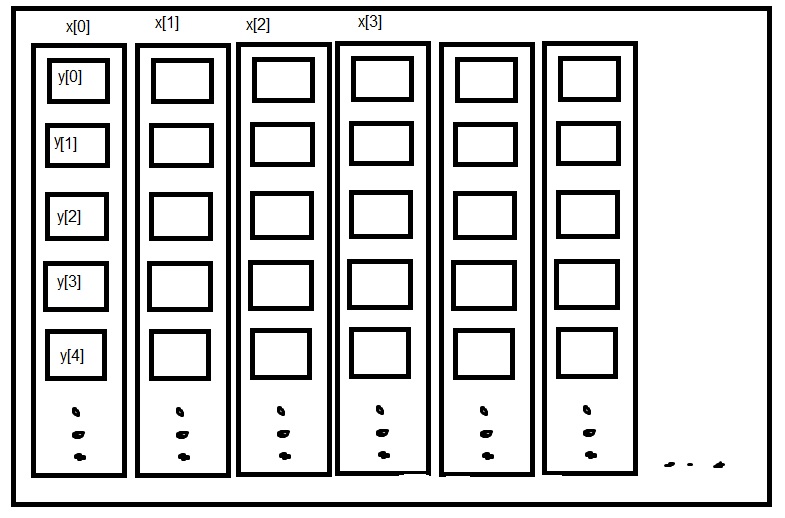
[5 Yhteenveto 6](#_Toc405392834)

# Johdanto

Olio ohjelmointi 2 kurssilla teemme kolmen hengen ryhmässä 2D tasohyppely peliä. Kartan koon kasvaessa peli alkoi kuitenkin hidastua johtuen piirtämisestä ja törmäystarkastelusta. Aloimme pohtia tähän ratkaisuja ja hyväksyimme Kytän esittämän idean kartan paloittelusta eri ruutuihin x ja y akselille. Tällöin tekisimme piirron ja törmäystarkastelut vain pelaajan lähellä oleville ruuduille. Päämääränä oli, ettei kartan koko tai vihollisten lukumäärä vaikuttaisi pelin ajettavuuteen vaan ainoastaan alku käynnistykseen.

Tässä dokumentissa tutkimme kartan latausta tiedostosta latauksen tehokkuutta sekä vihollisten määrän ja ruudunpäivityksen nopeuden suhdetta.

# Algoritmin tehokkuus & erityispiirteet

Alussa kartta parsittaisiin kartta editorissa käyttäjän tekemistä paloista merge tilassa. Käyttäjällä on tällä hetkellä kyky uudelleen rakentaa kartta ja tallentaa se. Tallennettu kartta avataan peli tilassa ja parsitaan kolmiulotteiseksi vektoriksi x- ja y akselin mukaan kuvan 1 tavoin.

Kuva

Pseudo-koodina algoritmi jolla mappi rakennettiin on seuraavanlainen:

Map()

{

kerroin = 0

koko = 100

avaa haluttu teksti tiedosto

kun on rivejä

{

jos rivi on --

{

heitä mapObjects vektori yRivi vektorin jatkoksi

kerroin++

tyhjää mapObjects vektori

jos kerroin on yhtäsuuri kuin koko

{

heitä yRivi vektori xRivi vektorin jatkoksi

muuta kerroin nollaksi

tyhjää yrivi vektori

}

}

muulloin

{

parsi saatu rivi kolmeen osaan: nimi, x koordinaatti ja y koordinaatti

vertaile minkä objectin pointteri luodaan mapObject vektoriin ja anna sille sijainti

}

}

sulje tiedosto

Tässä vaiheessa kartan paloittelun etuja ei havaita edes teoreettisesti. Kartan luominen hidastuu verrattuna siihen että kaikki esineet luotaisiin osoittimina vain yhteen vektoriin. Edut heijastuvat vasta kartan, pelaajan ja vihollisten päivittämisessä sekä piirrossa.

Pelaaja on aina ruudun keskellä, kamera seuraa pelaajaa. Tämä mielessä pseudo-koodi kartan piirtoon on seuraavanlainen

piirrä()

{

x = kameran keskusta().x / ruudun leveydellä

y = keskusta().y / ruudun korkeudella

for (int xx = -1; xx < 2; ++xx)

for (int yy = -1; yy < 2; ++yy)

if (x + xx >= 0 && y + yy >= 0 && x + xx < xRivi.size() && y + yy < xRivi.size())

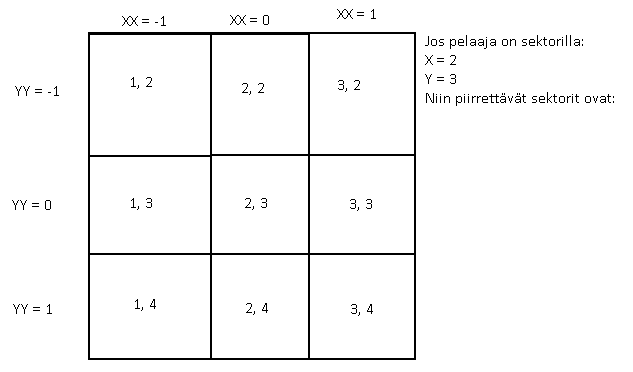
for (int i = 0; i < xRivi[x + xx][y + yy].size(); ++i)

piirrä(xRivi[x + xx][y + yy][i]->shape);

}

Tällöin maksimissaan 9 pelaajaa lähinnä olevaa ruutua piirretään sen sijaan että koko kartta piirrettäisiin kerralla. Tässä joudumme tekemään kaksi laskua ja 4+4+12+ruudussa olevien kappaleiden määrän verran vertailuja, mitkä ovat kevyitä tehdä.

Samanlaista logiikkaa käytettiin pelaajan päivittämisessä, vihollisten piirtämisessä ja päivittämisessä.



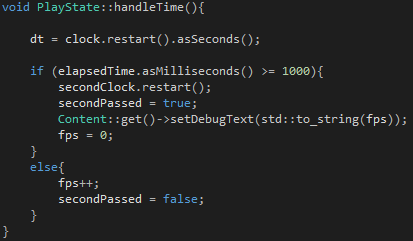
Kuva 2

Yksi ruutu kuvaa yhtä karttapalaa. Yhteensä päivitetään yhdeksää karttapalaa, jotka kuuluvat isompaan kartta kokonaisuuteen.

# Toteutus

Aloitimme testaamaan edellä mainittua kartanluonti algoritmia katsomalla mitä eri kokoja kartanluontiin pystyi laittamaan ja kauan ne veivät aikaa. Ensimmäiseksi selvitimme maksimi arvon, kuinka monta karttapalaa karttaan pystyi asettamaan, ennen kuin kääntäjä lopettaa koodin ajamisen. Minimiarvo on selvästikin 1x1 ruudukko, joka koostuu vain yhdestä kartanpalasta. Testasimme aina neliön muotoisilla kartoilla. Meidän lopetusehtomme oli ohjelman käynnistyminen ja kartan valmistuminen pelikäyttöön. Kartan latautumisnopeutta tutkimme sekunti kellolla, aloittaen kartanlataamisesta ja lopettaen pelitilan käynnistymiseen.

Testasimme tämän jälkeen vihollisten määrän vaikutusta ruudunpäivitykseen. Teoreettisesti vihollisten määrällä ei pitäisi olla vaikutusta ruudunpäivitykseen juuri lainkaan. FPS tulee sanoista Frames Per Second, kertoen kuinka monta ruudunpäivitystä tehdään sekuntissa.



Kuva 3 FPS mittari

# Testiaineiston generointi ja testaus

Kartan generointia peliympäristöön testattiin mittaamalla aikaa manuaalisesti sekuntikellolla ja vihollisten vaikutusta ruudunpäivitykseen mitattiin peliin ohjelmoidulla ruudunpäivitys mittarilla. Seuraavassa taulukossa nähdään mittaustulokset kartanluonnille ja vihollismäärälle:



Testattaessa kartanluontia vihollisia luotiin karttaan 200 kpl. Vihollisia testatessa kartan koko oli 50.

Ohjelman ajo kaatui, jos yritti ajaa suurempaa kuin 544 karttapalan karttaa per sivu. Yhteensä peli pystyy siis käsittelemään 295 936 karttapalaa samaan peliin.

Ohjelma kesti kääntyä vielä 100 000 viholliseen asti, mutta ruudunpäivitykselle jäi aikaa vain päivittyä 3 kertaa sekunnissa. Toki tulokset vaihtelevat tietokoneittain.

# Yhteenveto

Tuloksista voidaan huomata, että vaikka kartan lataukseen menee suhteellisen kauan aikaa, niin silti ohjelma pystyy merkittäviin tuloksiin. Tavallisessa peli tilanteessa todennäköisesti käyttäisimme n. 50–100 karttapalaa per kartta. Kartan koko itsessään ei vaikuttanut latauksen jälkeen pelin ruudun päivitys nopeuteen. Kuten kuvasta voi huomata, latausajan käyrä on melkein lineaarinen kartan kokoon verrattuna. Pientä kasvua on kuitenkin huomattavissa mennessä suurempiin lukuihin.

Vihollisten määrässä suhteessa ruudunpäivitysnopeuteen tuloksista voidaan huomata, että ruudunpäivitysnopeus laskee eksponentiaalisen hitaasti ja lähestyy nollaa, kunnes jumiutuu. Vihollisten määrä aiheutti yllättävän paljon rasitetta pelille, vaikka piirrämme ja aktivoimme (piirto ja törmäystarkastelu) vain näytöllä näkyvät viholliset. Ilmeisesti linkitetyssä listassa olevien vihollisten läpikäynti useita kertoja sekunnissa toi liikaa rasitetta. Käytännön tilanteeseen tämä menetelmä tosin sopii, koska ruudunpäivitys tippui alle 60 vasta 5000 yhtäaikaisen vihollisen jälkeen.