3D-Visualisointi

Ohjelmointistudio 2: Projekti

Aleksi Vuorjoki (355111)

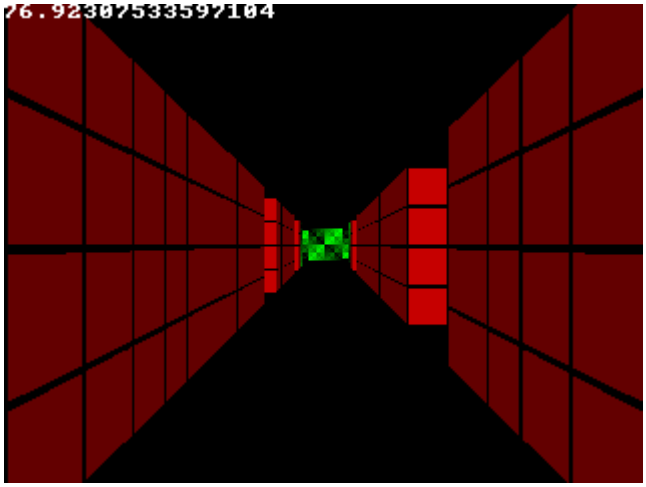
Tietotekniikka 2013

132.2014

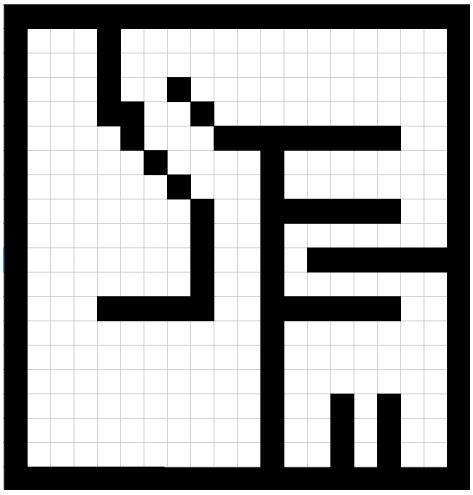
Projektityön dokumentti

Projektin yleiskuvaus

Yksinkertainen 3D-visualisointi- ja mallinnusohjelma. Ohjelma lukee itse suunnitellusta tiedostoformaatista labyrintin tapaisen suljetun ympäristön, renderöi sen ja mahdollistaa siinä liikkumisen. Ympäristö on yksikerroksinen ja siinä on vain seiniä, käytäviä ja huoneita. Seinät ovat pohjois-etelä ja itä-länsi suuntaisia. Seinät voivat olla yksivärisiä tai teksturoituja. Niiden tummuuden tulee riippua etäisyydestä. Ympäristö on tyhjä. Toteutuksessa ei saa käyttää valmiita 3D-kirjastoja.



Kuva 1: Esimerkkinäkymä



Kuva 2: Esimerkkipohjapiirros

Liikkuminen ympäristössä on vapaata mutta seinien läpi ei saa liikkua.

Seinät ovat yksivärisiä. Teksturoitu versio toimii ja renderer -luokka sitä varten on annettu, mutta suorituskyky on erittäin huono.

Tein myös mahdolliseksi muokata kentää siellä liikkuessa.

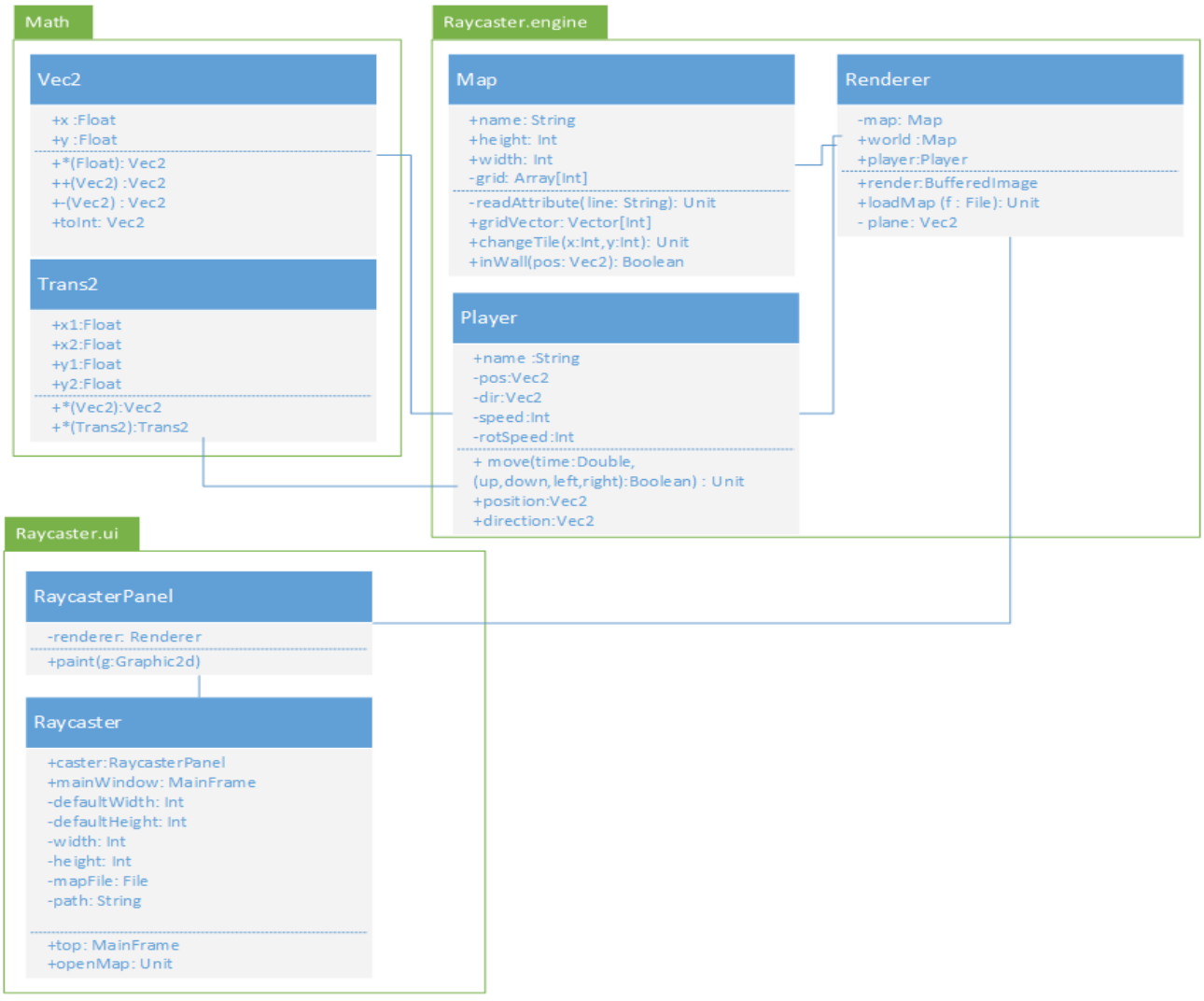
Kyseessä on siis Wolfenstein 3D:n pelimoottorin tapainen ohjelma. Tekniikka, jota käytetään on nimeltään raycasting.

Kuva 3: Wolfenstein 3D (1992), id Software

Käyttöohje

Kun ohjelman käynnistää eclipsessä (Raycaster.scala on käynnistysolio) se alkaa suoraan piirtämään oletuskenttää. Uuden kentän voi ladata Game valikosta löytyvästä tiedostoselaimesta. Mukana tulee useampi kenttä, jotka löytyvät maps kansiosta (nonDigit ja shortRow ovat rikkinäisiä tiedostoja). Kentässä voi liikkuua nuolinäppäimillä: eteen, taakse ja kääntyä oikealle tai vasemmalle. Edessä olevaan seinän voi poistaa tai eteen voi lisätä seinän painamalla E- näppäintä.

Ohjelman rakennesuunnitelma



Kaavio 1: UML-kaavion tapainen

Paketti Math:

Math sisältää ohjelman laskentaa varten hyödyllisiä rakenteita. Alustavasti ohjelmassa tarvitsee kaksiulotteisia vektoreita (Vec2). Näitä ovat muun muassa pelaaja sijainti ja suunta. Vektoreita on mahdollista kertoa luvuilla ja niihin on mahdollista lisätä toisia vektoreita. Lisäksi tarvitaan 2x2 matriseeja (Trans2). Pelaajan kääntyessä suuntavektori (vec2) kerrotaan matriisilla ja sitä varten luokassa on metodi *(Vec2). Matriiseja on myös mahdollista kertoa keskenään, jotta transformaatiota saadaan yhdistettyä.

Paketti Raycaster.Engine

Ohjelman tärkein osio. Pääosassa on Renderer, jolla on Player ja Map. Mapista Renderer saa tiedon mitä pitää piirtää. Mappiin on siis ladattu tiedostosta kartta. Tämä tehdään metodilla readFile, joka ottaa parametrikseen halutun tiedoston. Mapilla on myös mittasuhteet ja nimi. Player-oliolla on nimi, sijainti(vec2),suunta(vec2),nopeus ja kääntönopeus. Nopeuksia käytetään liikkuessa tasaisen liikken saavuttamiseksi. Playerilla on myös metodit liikettä varten. Player oliolta Renderer saa siis tiedon siitä, mistä näkökulmasta karttaa pitää piirtää. Rendererin metodi render piirtää kuvan pelimaailmasta bufferedImageen

Paketti Raycaster.ui

Ohjelma ei suurenpia käyttöliittymiä kaipaan. On vaan pääolio Raycaster, joka on swing-applikaatio ja ikkunassa on yksi RaycasterPanel, jolla on Renderer, jonka tuottamia kuvia se paint metodillaan piirtä ikkunaan. Paneeli huolehtii myös tapahtumankuuntelusta eli kontrolleista joten se kutsuu Rendererin kautta pelaajan liikkumiskomentoa.

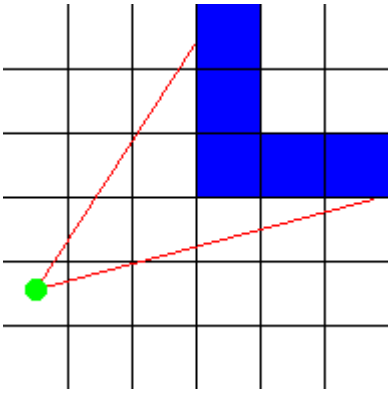
Käyttötapauskuvaus

Käyttäjä käynnistää sovelluksen ja ohjelma lataa koodissa määritellyssä tiedostosta kentän. Ohjelma renderöi avatuneeseen ikkunaan kentästän näkymän, joka riippuu pelaajan sijainnista. Renderöinnistä huolehtii pääsilmukka, joka pyörii koko ajan ohjelman ollessa käynnissä. Pelaaja liikkuu kentässä nuolinäppäimill', jota swingin tapahtumakuuntelija kuuntelee. Pelaaja painaa I-näppäintä nähdääkseen lisätietoja, kuten FPS:n.

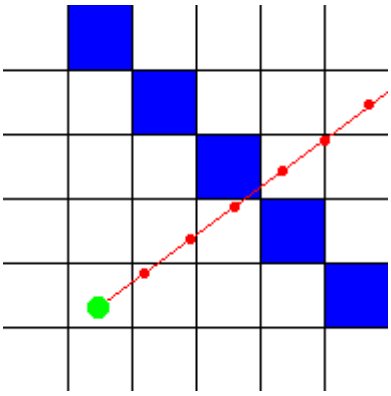
Algoritmit

Pelimaailman yksinkertaisuus – kaikki seinät samankorkuisia ja kuutioita – mahdollistaa sen, että kaikki piirtämistä varten suoritettava laskenta voidaan suorittaa kahdessa ulottuvuudessa normaalin kolmen sijaan. Ideana on jakaa näyttö pikselin levyisiin pystykaistaleisiin, ja ”ampua” säde jokaista kaistaletta kohti. Jokaisen säteen kohdalla lasketaan, kuinka pitkän matkan säde kulkee ennen kun se osuu seinään. Tämän etäisyyden avulla lasketaan kuinka korkeana kukin

kaistale kuuluu ruudulle piirtää. Ihminen kyllä näkee tarkalleen milloin säde osuu seinään, mutta tietokone voi tarkistaa osuman vain tietyin välein (Kuva 2), jolloin on mahdollista, ettei osumaa huomata. Tarkistusaskelta voidaan pienentää, jolloin testauksen tarkkuus paranee, mutta voi silti osuma saatetaan vasta tarkistaa, kun ollaan seinään sisällä(Kuva 3), jolloin etäisyydestä tule liian suuri. Äärimmäiseen tarkkuuteen tarvitaan äärimmäisen pieni tarkistusaskel ja tietenkin äärimmäisen paljon laskutoimituksia. Tämä ei tietenkään käy päinsä. Onneksi, koska kenttä on ruudukko, voimme ratkaista ongelman tarkistamalla osuuko säde seinään aina kun se leikkaa ruudun reunan (Kuva 4). Tällä tavoin osumme aina varmasti seinään ja vielä oikealla etäisyydellä. Tähän käytetään algoritmia, joka perustuu DDA:han (Digital Differential Analysis). DDA:lla laskentaa minkä ruutujen kautta ruudukkoon piirretty suora kulkee. Sitä käytetään esimerkiksi viivojen piirtämiseen tietokoneen näytölle.



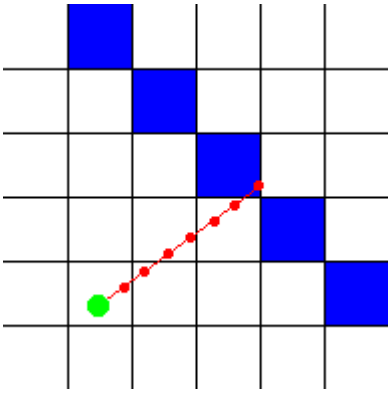
Kuva 1



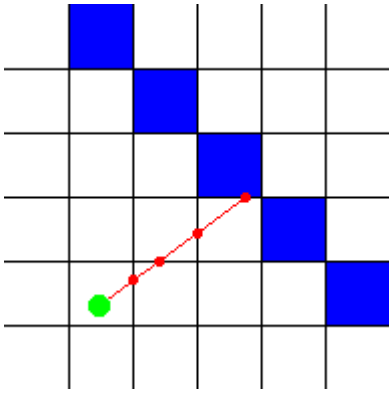
Kuva 2

DDA

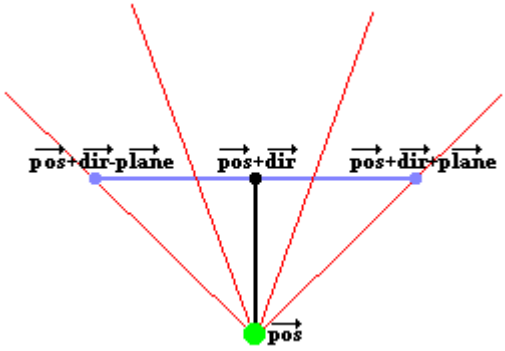
Säteen lähtöpiste on pelaajan sijainti pos. Pelaajalla on suuntavektori dir ja näyttöä kuvaa vektori plane, joka on kohtisuorassa dir-vektoriin ja joka on nolla näytön keskipisteessä. Näin saadaan säiden suuntavektorit laskemalla yhteen dir ja plane, joka kerrotaan kertoimella, joka lasketaan 2*x/w-1, missä x on pystykaistaleen numero ja w on ikkunan leveys (Kuva 5).



Kuva 3

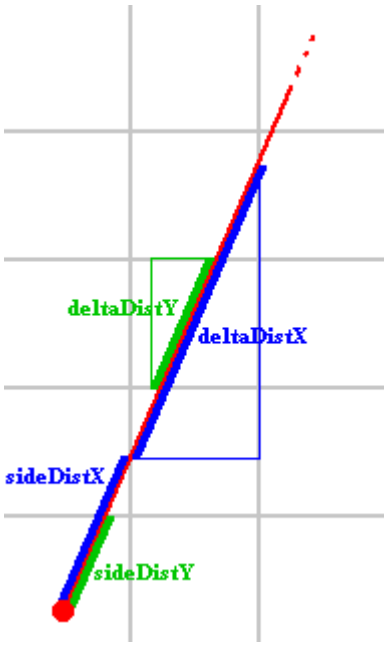


Kuva 4



Kuva 5

Kuvasta 6 näkee DDA:han tarvittavan komponentit. SideDistX ja SideDistY on etäisyys ensimmäiseen ruudun reunaan lähtöpisteestä ja deltaDistX ja deltaDistY ovat etäisyydet, jotka pitää lisätä alkuetäisyyksiin, että päästään seuraavalle reunalle.



Kuva 6

DeltaDistX = sqrt(1+(rayDirY^2/rayDirX^2))

DeltaDistX = sqrt(1+(rayDirX^2/rayDirY^2))

rayDirX ja rayDirY ovat säteen suuntavektorin x- ja y-komponentit

Jos säde lähtee vasemmalle:

sideDistX = (rayPosX - mapX) * deltaDistX

sideDistY = (rayPosY - mapY) * deltaDistY

Jos säde lähtee oikealle tai suoraan:

sideDistX = (mapX + 1.0 - rayPosX) * deltaDistX

sideDistY = (mapY + 1.0 - rayPosY) * deltaDistY

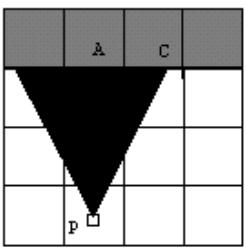
mapX ja mapY on säteen, ja alussa pelaajan, kordinaatit karttaruudukossa

Nyt meillä on tarvittavat tiedot DDA:n suorittamiseen

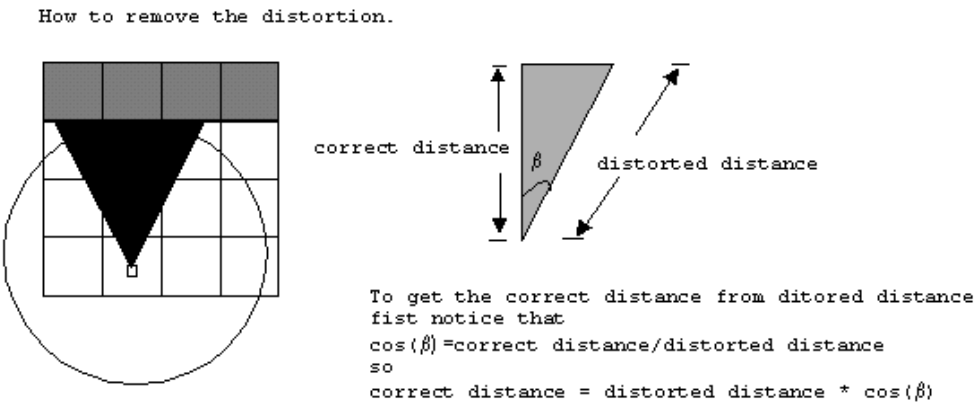
Eli lisätään lyhyenpää sideDist vastaava deltaDist ja päivitetään säteen mapX ja mapY sen mukaan mihinkä suuntaan on liikuttu yksi ruutu. Tätä toistetaan kunnes osutaan seinään.

Kuitenkin, jos kuvan piirtää näin saaduilla etäisyyksillä ei synny täysin oikea kuva. Alla olevissa kuvissa ilmiö ja sen korjaaminen on selitetty.

```
Wall slices that are farther from the center (point A in the figure),
appear shorter.  This is because rays that
are farther from the center of projection have longer distance.
For instance, the ray PC is longer than PA.
(Like in real life, the farther the wall, the smaller the wall appears.
However, the computer screen, unlike the spherical human eyes, is flat.
Therefore we must somehow counter this effect.)
```



```
Result of projection
without removing
the distortion.
```



Tietorakenteet

Kartan tallentamiseen ja käsittelyyn käytän Arrayta, koska kartan koko ei tule muuttumaan ,joten ei ole tarvetta Bufferille. Jokainen alkio kuvaa siis yhtä kartan ruutua. Arrayta käytän Vectorin sijaan koska täytyy ensin luoda kartan kokoinen taulukko ja vasta sen jälkeen lukea siihen kartan tiedot. Lisäksi Array mahdollistaa ympäristön muokkaamisen pelissä. Vectoria käytän kyllä julkisena rajapintana karttaan. Lisäksi toteutan kaksi omaa tietorakennetta : vektorin ja transformaatiomatriisin. Jotka kuvaavat siis ohjelman laskennassa tarvittavia vektoreita ja matriiseja. Ne ovat käytännössä vain Float-tupleja, joille on luotu metodeita. Niille on toteutettu vektorien yhteenlasku ja vektorien ja matriisien välinen kertolasku.

Tiedostot

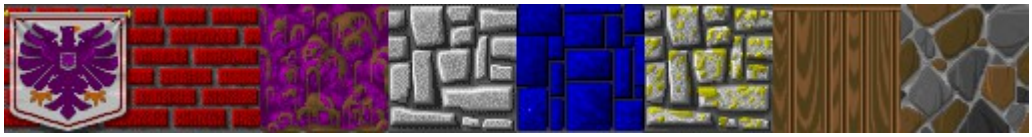
Ohjelma lataa kentän tiedostosta. Tiedostoformaatti on tekstitiedosto, jossa jokainen kentän ruutu kuvataan numerolla: 0 on vapaa ruutu eli siinä voi liikkuaa ja 1 on seinä. Monivärisessä tai teksturoidussa versiossa numerot 1-? kertovat mitä tekstuuria/väriä seinässä kuuluu käyttää. Tiedoston alussa on ilmoitettu sen leveys sekä korkeus kuvan mukaisesti. Tiedosto ovat .world-päätteisiä

RayCasterMapTextured.world

```
1  w:20
2  h:20
3  11111111111111111111
4  10004000000000000001
5  10004000000000000001
6  10002002000000000001
7  10004400200000000001
8  10000300022222222001
9  10000040000100000001
10 10000004000100000001
11 10000000400111111001
12 10000000400100000001
13 10000000400101111111
14 10000000400100000001
15 10005555500111111001
16 10000000000100000001
17 10000000000100000001
18 10000000000100000001
19 10000000000100303001
20 10000000000100303001
21 10000000000100303001
22 11111111111111111111
```

Kuva 6: Monivärinen kenttä

Tekstuurit ohjelma lataa tavallisista kuvatiedostoista esim .png.



Kuva 7: Mahdollisia tekstuureja. Pelistä Wolfenstein 3D

Testaus

Testasin luomiani vektoreita ja matriiseja yksikkötesteillä. Lisäksi yksikkötestasin kartan lukemista tiedostosta Array:hin. Rikkinäisiä tiedostojat lukevat testit eivät varsinaisesti testaa muuta kuin sen, että tuotetaan oikeat virheviestit. Muuten testaaminen oli vain ohjelman käyttämistä. Testaaminen oli siis aivan suunnitelman mukaista

Ohjelman tunnetut puutteet, viat

Ongelma: Teksturoitu versio erittäin hidas

Mahdollinen ratkaisu: BufferedImage getRGB ja setRGB ei ole ehkä nopein tapa

Ongelma: Tekstuurit välillä venynäitä

Mahdollinen ratkaisu: ???

Ongelma: Nurkista voi kävellä läpi, jos sen takana on tyhjää tilaa

Mahdollinen ratkaisu: Pitäisi kirjoitaa tarkistus onko kyseessä nurkka

Puute: Teksturoidussa versiossa seinät kauempana eivät ole tummempia

Hyvää/Huonoa

Hyvää: Vektori ja matriisiluokat ovat selkeästi toteutetut ja dokumentoidut

Huonoa:

- render-metodi varsin suuri, mutta kun koitin sitä pilkko sliceTexture metodin parametri listastakin tuli varsin pitkä
- Olisin voinut kommentoida enemmän

Poikkeamat suunnitelmista ja aikataulu

En toteuttanut hiiriohjausta enkä sivuttais suuntaista liikettä ajan loppumisen vuoksi. Nämä olivat minulla kuitenkin lisäominaisuuksissa ja pysyin muuten aikataulussa hyvin noudattamalla työjärjestystä. Aika-arviot olivat useassa tapauksessa kuten renderöinnissä jopa yläkanttiin.

Arvio lopputuloksesta

Ohjelma toimii niinkuin pitää. Suorituskyky on harmittavan alhainen ja jotain ominaisuuksia joitui karsimaan. Dokumentin teko jäi viime tippaan.

Lähteet

http://lodev.org/cgtutor/raycasting.html

http://www.permadi.com/tutorial/raycast/index.html

Peter Shirley, Michael Ashikhmin, Steve Marschner: Fundamentals of Computer Graphics