Projektin dokumentti

Miska Kananen (652102, TiK, 1.vk)

23. huhtikuuta 2019

Sisältö

1 Yleiskuvaus

Toteutin ohjelman, jolla voi renderöidä reaaliajassa kolmioista koostuvia maailmaan sijoitettuja kolmiulotteisia malleja. Mallit voivat olla mielivaltaisia, eivätkä ne ole rajoittuneita esimerkiksi pelkästään suorakulmaisiin seiniin. Mallin ympärillä voi liikkua vapaasti ja kameraa voi kääntää mielivaltaiseen suuntaan.

2 Käyttöohje

Kun ohjelma käynnistetään, se lataa mallinnettavan maailman `world`-nimisestä tekstitiedostosta ohjelman juurikansiosta. Jos lataaminen epäonnistuu, tulostuu tieto tästä standarditulosteeseen.

Kameraa voi liikuttaa eteen, taakse ja sivuille `W, A, S, D`-näppäimillä. `Q` ja `Z` liikuttavat kameraa ylös ja alas. Nuolinäppäimet kääntävät kameraa.

3 Ohjelman rakenne

Ohjelma on jaettu komponentteihin siten, että maailman tila, renderöinti ja ruudulle piirto ovat omina erillisinä komponentteinaan.

3.1 Maailman tila

Luokat ***Vector4 ja Matrix4*** kuvaavat laskennassa käytettäviä neliulotteisia vektoreita ja matriiseja ja toteuttavat niille tarvittavat laskutoimitukset.

Luokka ***Vertex*** kuvaa kolmion nurkkapistettä. Kullakin *Vertexillä* on:

- sijainti object spacessa (yksittäisen kappaleen omassa koordinaatistossa)
- väri

Luokka ***Triangle*** kuvaa kolmesta *Vertexistä* koostuvaa kolmiota. *Triangle* on avaruuden kappaleiden rakennuspalikka ja yksinkertaisin renderöitävissä oleva primitiivi. *Trianglella* on:

- kolme *Vertexiä*, eli kolmion nurkkapisteet object spacessa
- materiaali (kertoo miten kolmio piirretään: umpinainen, wireframe)

Luokka ***Model*** mallintaa yksittäistä, kolmioista koostuvaa kappaletta. Kolmiot on aseteltu kappaleen omaan object spaceen. *Modelilla* on:

- yhden tai useamman *Trianglen* muodostama mesh eli käytännössä lista kolmioista, jotka muodostavat yhdessä kappaleen
- metodit, joilla voi asettaa materiaalin ja värin kerralla koko meshille

Luokka ***CollisionBox*** kuvaa kappaleen muotoa törmäystunnistuksen kannalta. *CollisionBox* on koordinaattiakselien suuntainen suorakulmainen särmiö, jonka läpi liikkuminen ei ole sallittua. *CollisionBoxilla* on kaksi nurkkapistettä, jotka määrittävät särmiön muodon. Pisteet ovat kappaleen object spacessa, eli samassa koordinaattijärjestelmässä kuin kappaleen *Modelin* verteksit.

Luokka ***WorldObject*** kuvaa maailmassa olevaa objektia, kuten esimerkiksi seinää. *WorldObjectilla* on

- *Model*, joka määrittää kappaleen muodon ja ulkonäön
- mahdollinen *CollisionBox*, joka kuvaa kappaleen muotoa törmäystunnistuksen kannalta
- metodi, jolla voi kysyä, onko jokin piste kappaleen sisällä

- Transform, eli 4x4-matriisi (world matrix), joka kuvaa kappaleen *Modelin* ja *CollisionBoxin* verteksit object spacesta world spaceen. Matriisin avulla asetetaan kappale avaruuteen haluttuun paikkaan ja asentoon.

Luokka ***World*** kuvaa mallinnettavaa avaruutta ja pitää sisällään kaikki avaruudessa olevat kappaleet. *Worldilla* on

- lista *WorldObjecteista* sopivine Transformeineen, eli maailmassa olevat kappaleet oikeilla paikoillaan
- *Camera*, eli kamera, jonka näkökulmasta maailma renderöidään
- metodi, jolla voi kysyä, onko jokin piste maailman kappaleen sisällä

3.2 Renderöinti

Luokka ***Camera*** määrittää, mistä paikasta maailmaa tarkastellaan ja mihin suuntaan katsotaan. Kamera on aina renderöitäessä oman koordinaatistonsa (view spacen) origossa osoittaen $+z$-suuntaan, muuta maailmaa liikutetaan ja käännetään tarpeen mukaan. *Cameralla* on:

- Transform, eli 4x4-matriisi (view matrix), joka kuvaa maailman *Modelien* verteksit world spacesta view spaceen siten, että kamera on origossa $+z$-suunnassa
- metodi view matrixin rakentamiseen siten, että kamera on maailmaan nähden halutussa paikassa ja osoittaa haluttuun suuntaan

Luokka ***Renderer*** ottaa renderöitäväksi **Worldin** ja kuvaa sen **Modeleiden** verteksit *Cameran* view spacesta clip spaceen, eli koordinaatistoon, jossa tarkastellaan verteksien näkyvyyttä. Tämän jälkeen *Renderer* selvittää, mitkä *Triangleista* ovat kameran näkökentässä, hylkää sen ulkopuolella olevat, leikkaa sopivasti osittain näkyvissä olevia ja lopulta projisoi koko näkymän clip spacesta image spaceen, eli kaksiulotteiselle tasolle. *Renderer* luo z-koordinaatin mukaan järjestetyn listan piirrettävistä *Triangleista*, jotka on projisoitu perspektiiviprojektiolla xy-tasolle siten, että niiden x- ja y-koordinaatit on normalisoitu välille $[-1.0, 1.0]$.

3.3 Piirto ruudulle

Luokka ***Screen*** kuvaa näkyvää 2D-pikseliruudukkoa, johon kuva piirretään. *Screen* ottaa *Rendereriltä* listan projisoiduista *Triangleista*, kuvaa niiden verteksit väliltä $[-1.0, 1.0]$ pikselikoordinaatteihin ja piirtää ne näkyviin ruudulle materiaalit ja värit huomioiden.

3.4 Muut luokat

Luokka ***Engine*** käynnistää ohjelman, luo käyttöliittymän, kuuntelee käyttäjän syötettä ja päivittää grafiikkaa ja logiikkaa tasaisin väliajoin.

Luokka ***WorldLoader*** lukee maailman tiedostosta ja luo sen pohjalta *Worldin* ilmentymän.

Luokka **Constants** sisältää ohjelmassa käytettäviä vakioita.

Luokka **Helpers** sisältää usein tarvittavia apufunktioita.

4 Algoritmit

4.1 Renderöinti

Ohjelma renderöi kolmioista koostuvia kappaleita rasteroimalla. Kolmioiden verteksit projisoidaan perspektiiviprojektiolla 2D-tasolle näytölle piirtämistä varten. Kolmiot piirretään järjestyksessä kauimmaisesta lähimpään, jolloin kappaleet peittävät niiden takana mahdollisesti olevat toiset kappaleet.

Toinen vaihtoehto olisi voinut olla raycasting, jossa jokaista pikselien saraketta tai pikseliä kohtaan lähetetään kamerasta maailmaan säde, tarkastellaan, mihin kappaleeseen säde osuu ja värjätään pikseli tai sarake sen mukaan. Rasterointi on kuitenkin tehokkaampaa kuin raycasting ja se sallii kameran vapaan liikkeen ja mielivaltaisen maailman geometrian raycastingia helpommin.

4.1.1 Perspektiiviprojektio

Kamera on view spacen origossa osoittaen suuntaan $+z$ ja halutaan muodostaa projektio view spacesta tasolle $z = 1$.

Tarkastellaan projisoitavaa verteksiä p, jota kuvaa vektori $p = (x, y, z)$. Yksinkertainen ratkaisu on jakaa p sen z-koordinaatilla, jolloin saadaan projektio $p' = (x/z, y/z, 1)$. Ongelmana on, että tämä projektio kadottaa kai-

ken informaation projisoitavien pisteiden z-koordinaateista, jolloin ei voida tietää, missä järjestyksessä kolmiot tulisi piirtää. Tarvitaan projektio, joka kuvaa x- ja y-koordinaatit vastaavasti kuin aiemmin, mutta säilyttää z-koordinaattien keskinäisen järjestyksen.

Kameran näkemä avaruuden alue, eli *view frustum* on kamerasta $+z$-suuntaan kasvava katkaistu pyramidi, joka rajoittuu päistään kahteen tasoon: *near planeen* ja *far planeen*, eli $z = \text{near}$ ja $z = \text{far}$. View frustumin sisällä olevat verteksit ovat näkyvissä ja muut eivät. Halutaan projektio, joka kuvaa near planella olevan verteksin koordinaattiin $z = 0$, far planella olevan koordinaattiin $z = 1$ ja tasojen välissä olevat koordinaatit järjestyksessä välille $[0, 1]$.

Kun p on homogeenisissä koordinaateissa, eli $p = (x, y, z, (w = 1))$, tällainen projektio saadaan aikaan 4x4-projektiomatriisilla:

$$p' = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{far+near}{far-near} & -\frac{2 \cdot far \cdot near}{far-near} \\ 0 & 0 & 1 & 0 \end{pmatrix} \cdot p$$

Projektiomatriisi kuvaa verteksin clip spaceen. Kuvauksen jälkeen koordinaatit homogenisoidaan, eli jaetaan sellaisella skalaarilla, että taas pätee $w = 1$. Ne verteksit, jotka olivat alun perin view frustumin sisällä, ovat projektion ja homogenisoinnin jälkeen $x, y \in [-1, 1], z \in [0, 1]$-suorakulmion sisällä ja muut sen ulkopuolella.

Near plane tarvitaan, koska jos near $= 0$, z-informaatio menetetään kuvauksessa kuten aiemmin.

4.1.2 Kolmioiden leikkaus

Clip spacessa tarkastellaan, mitkä projisoiduista kolmioista ovat kameran näkökentässä. Helppoja tapauksia ovat, jos kolmion kaikki verteksit ovat $x, y \in [-1, 1], z \in [0, 1]$-suorakulmion sisällä tai sen ulkopuolella: sisällä olevat kolmiot ovat kokonaan näkökentässä ja ne piirretään sellaisenaan, ulkona olevat eivät näy ja ne voidaan jättää piirtämättä suoraan.

Mielenkiintoinen tapaus on, jos osa kolmion vertekseistä on kuution sisällä ja osa kuution ulkopuolella. Kolmiota ei voi piirtää kokonaan, koska kaikki verteksit eivät ole näkyvissä, mutta ulkopuolella olevia verteksejä ei voi jättää huomiottakaan, koska verteksit määrittelevät kolmion sivut.

Kolmio leikataan tällöin osiin siten, että saadut osakolmiot ovat joko kokonaan näkyvissä tai näkymättömissä. Leikkaus toistetaan view volumen kullekin sivulle. Voidaan laskea pisteet, jossa kolmion sivut leikkaavat tason, ja näistä pisteistä saadaan kolmiolle uudet kärkipisteet. Tasoja kuvataan yhden tason pisteen ja tason normaalivektorin avulla, ja kolmion sivuja vastaavasti vektorimuodossa kärkipisteen ja sivun suuntavektorin avulla.

4.1.3 Projisoitujen kolmioiden piirto

Kuva saadaan näkyviin piirtämällä projisoidut kolmiot 2d-tasolle. Kun *Rendereriltä* on saatu lista projisoiduista kolmioista, jotka ovat image spacessa, verteksien koordinaatit muutetaan pikselikoordinaateiksi ja kolmiot piirretään z-järjestyksessä kaukaisimmasta lähimpään tasolle materiaali ja verteksien värit huomioiden.

4.2 Törmäyksentunnistus

Maailman kappaleilla on koordinaattiakselien suuntaiset suorakulmion muotoiset collision boxit, jotka ovat juuri niin isoja, että kaikki kappaleen verteksit mahtuvat sen sisälle.

Kun kameraa yritetään liikuttaa, lasketaan piste, johon kamera liikkeen jälkeen päätyisi. Jos piste on jonkin kappaleen collision boxin sisällä, kameran ei anneta liikkua kyseiseen suuntaan.

5 Tietorakenteet

Ohjelma käyttää standardikirjaston tarjoamia tietorakenteita, kuten taulukoita, Buffereita ja Mappeja.

Jos renderöintiä tai törmäyksentunnistusta haluttaisiin tehostaa, maailman objektit voitaisiin tallentaa esimerkiksi octreehen, jolloin läpikäytävien objektien määrää saataisiin vähennettyä keskimääräisessä tapauksessa.

6 Tiedostot

Renderöitävä maailma ladataan `world`-nimisestä tekstitiedostosta, jonka rakenne on seuraava.

Ensimmäisellä rivillä on 3 desimaalilukua: kameran x, y ja z-koordinaatti alussa.

Toisella rivillä on ei-negatiivinen kokonaisluku n: kappaleiden määrä.

Seuraa n kappaleen kuvausta. Kunkin kuvauksen rakenne on seuraava:

Kuvauksen ensimmäisellä rivillä on kolme desimaalilukua x, y ja z: kappaleen keskipisteen (model spacen origon) sijainti world spacessa.

Kuvauksen toisella rivillä on ensin kappaleen tyyppi, joka on joko `SOLID` tai `WIREFRAME`. Tyypin jälkeen samalla rivillä on neljä kokonaislukua r, g, b, a ($0 \leq r, g, b, a \leq 255$), jotka kuvaavat kappaleen värin ja läpinäkyvyyden.

Kuvauksen kolmannella rivillä on epänegatiivinen kokonaisluku m: kappaleen kolmioiden määrä.

Seuraa m riviä, joista kullakin on 9 desimaalilukua $x1$, $y1$, $z1$, $x2$, $y2$, $z2$, $x3$, $y3$ ja $z3$: kolmion verteksien koordinaatit.

Esimerkki kelvollisesta tiedostosta, jossa on yksi yhdestä kolmiosta koostuva kappale:

```
1 0 0.5 0
2 1
3 0 0 0
4 WIREFRAME 255 0 0 255
5 1
6 -1 -1 0 0 1 0 1 -1 0
```

7 Testaus

Vektorien ja matriisien matemaattisilla operaatioilla on yksikkötestit.

Renderöintiä ja piirtoa ei yksikkötestattu, koska ongelma on hyvin visuaalinen, ja ohjelmaa käyttämällä huomasi helpommin ja monipuolisemmin ongelmia ja puutteita kuin "tämän pikselin pitäisi olla punainen" tyyppisillä yksikkötesteillä.

WorldLoaderilla olisi voinut ja kannattaisikin olla yksikkötestit, mutta niiden lisäämiseen ei ollut aikaa.

8 Puutteet ja viat

Kolmioita piirrettäessä z-järjestys ei ole täydellinen, eli joissakin tapauksissa toisen kolmion takana oleva kolmio voi näkyä sen edessä. Tällä hetkellä järjestys on, että kolmiot, joiden kameraa lähin verteksi on kauimpana,

piirretään ensin. Ongelmia tulee ainakin, jos jokin kolmio on osittain toisen edessä ja osittain toisen takana. Esimerkiksi lattian tekeminen on ongelmallista, jos sen päällä on seiniä.

Ongelman voisi ratkaista esimerkisi z-puskurilla, jossa jokaiselle pikselille pidetään yllä pienintä z-arvoa, johon kyseiseen pikseliin on tähän mennessä piirretty. Arvon perusteella voidaan määrittää, pitääkö kyseinen pikseli ylikirjoittaa uutta kolmiota piirrettäessä. Tällöin 2d-piirto pitäisi kuitenkin toteuttaa käsin pikseli pikseliltä ScalaFX:n primitiivien sijaan.

WorldLoaderia ei ole testattu kunnolla: kelvollisen, ei-rajatapaustiedoston pitäisi latautua oikein, mutta on mahdollista, että lataus ei toimi oikein joissakin rajatapauksissa.

9 Parhaat ja heikoimmat kohdat

+ Ohjelma toteuttaa renderöinnin yleisemmin kuin tehtävänannossa vaadittiin: kappaleiden muodot voivat olla mielivaltaisia, ne voivat olla missä asennossa tahansa ja voivat pyöriä. Kamera voi myös liikkua vapaasti avaruudessa.

– Kolmioiden väri määräytyy verteksien värien keskiarvona, mikä voi näkyä hyppyinä kolmioiden värityksessä. Oikeasti värin pitäisi määräytyä liukuvasti riippuen etäisyydestä kolmion vertekseihin. Toteutinkin version, jossa kiinteiden kappaleiden pikselien värit määräytyvät gradienttina, mutta piirto ScalaFX:n Canvasille osoittautui tällöin pullonkaulaksi.

– Kappaleiden mahdollista liikettä tai pyörimistä ei pysty määrittelemään kappalekohtaisesti `World`-tiedostossa, ja maailman ensimmäinen kappale on tällä hetkellä kovakoodattu pyöriväksi demotarkoituksia varten *Engine*-luokassa.

10 Poikkeamat suunnitelmasta

Ohjelma toteutettiin lähes juurikin suunnitelman mukaan, törmäyksentunnistusta yksinkertaistettiin hieman, koska tarkempi järjestelmä ei ollut kovin relevantti projektin kannalta.

Toteutusjärjestyksessä oli myös pieniä muutoksia. Oikea järjestys oli

1. (01.03) Vektorit ja matriisit
2. (02.03) Vertex, Triangle, Model, WorldObject
3. (03.03) Camera, World
4. (15.03) Renderer
5. (21.03) Screen, ruudulle näkyviin asioita
6. (30.03) Main loop
7. (13.04) Kolmioiden leikkaus
8. (14.04) Törmäyksentunnistus

eli ohjelman sai käyntiin suunniteltua myöhemmin.

11 Kokonaisarvio

12 Viitteet

- Jaakko Lehtinen. "Understanding View Frustums and Homogenous Coordinates". 2009.
- Jaakko Lehtinen. "Coordinate Transformations & Homogenous Coordinates". 2018. Kalvot kurssilta CS-C3100 Computer Graphics.
- http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/
- https://www.gabrielgambetta.com/computer-graphics-from-scratch/clipping.html
- http://geomalgorithms.com/a05-_intersect-1.html
- Scala documentation https://www.scala-lang.org/api/current/index.html