Aleksi Anderson

Grafiikan rasterointi ja renderöinti

Opinnäytetyö

Tieto ja Viestintätekniikka

Kevät 2024

Tiivistelmä

Abstract

Sisällys

[1 Johdanto 1](#_Toc164073691)

[2 Teoria 2](#_Toc164073692)

[2.1 Konsoli 2](#_Toc164073693)

[2.2 Kanvaasi 3](#_Toc164073694)

[2.3 Rasterointi 3](#_Toc164073695)

[2.3.1 Viivojen rasterointi 3](#_Toc164073696)

[2.3.2 Kolmioiden rasterointi 8](#_Toc164073697)

[2.4 Teksturointi 9](#_Toc164073698)

[Lähteet 10](#_Toc164073699)

Liitteet

#### Symboliluettelo

# Johdanto

Tässä käydään läpi objektien renderöintiä ja rasterointia. Tarkoituksena on käyttää Windowsin konsolia renderöinti-ikkunana, jolloin kaikki piirtäminen tapahtuu ASCII-grafiikalla.

# Teoria

Grafiikan piirto on melko matemaattista, joten tässä käydään läpi yleisten esineiden rasterointi- ja renderöintiprosessi sekä niihin liittyvät algoritmit Python-tyylisellä pseudokoodilla. Käydään myös läpi näyttötavan, joka on Windowsin konsoli.

## Konsoli

Konsoli on sovellus, joka tarjoaa I/O-palveluja merkkitilan sovelluksille. Tämä tarkoittaa käytännössä sitä, että konsoli voi lukea käyttäjän syöttöjä, kuten näppäinten painalluksia tai hiiren liikkeitä, input-streamiin ja renderöidä output-streamin tekstin näytölle. Windows-konsoli pystyy renderöimään koko Unicode-merkistön, mutta tässä käytetään vain Windows-1252-merkistöä, koska siinä on kaikki tarvittavat merkit ja se yksinkertaistaa ohjelman toteutusta. [1.] Tärkeimmät merkit ovat 223(▀), 220(▄) ja 219(█), koska niillä voidaan näyttää ylempi, alempi ja kaksi päällekkäistä pikseliä, mikä käytännössä kaksinkertaistaa pystysuuntaisen resoluution. Merkit 176(░), 177(▒) ja 178(▓) saattavat myös olla hyödyllisiä, koska niiden avulla voidaan sekoittaa värejä, mutta niiden käyttäminen tarkoittaisi pystysuuntaisen resoluution puolittamista.

Konsolia voidaan käyttää kahdella eri tavalla; joko konsolin APIn tai virtuaaliterminaali-sekvenssien kautta. Konsolin API käyttää Microsoftin määrittämiä C++-funktioita konsolin hallintaan, kuten kursorin sijainnin asettamiseen, kynän värin muuttamiseen tai tekstin kirjoittamiseen. Virtuaaliterminaali-sekvenssit ovat toisaalta toimintoja, jotka esitetään ei-tulostettavina merkkeinä, jotka voidaan laittaa normaalin tekstin väliin terminaalin hallitsemiseen. Tässä käytetään virtuaaliterminaali-sekvenssejä, koska Microsoft suosittelee niitten käyttöä, ja ne ovat yhteensopivia monien muiden terminaaliemulaattorien kanssa, CMDn ja PowerShellin lisäksi. [1.]

Windows-konsoli pystyy renderöimään 16 eri väriä sekä taustalla että etualalla: jokaisella merkillä voi olla eri väri taustalle ja etualalle, mutta emme voi asettaa taustan ja etualan väriä samaksi [1]. Jos ylä- ja alapikseleillä on eri väri, on käytettävä taustan väriä yhteen niistä, koska yksi merkki esittää kahta päällekkäistä pikseliä. Sillä ei ole väliä, kumpaa näistä kahdesta tausta esittää, joten voidaan yksinkertaisesti asettaa käytäntö, jonka mukaan 223(▀)-merkki renderöidään aina, jos kaksi pikseliä ovat eri värejä.

## Kanvaasi

Ennen kuin renderöidään mitään konsolille, tarvitaan ensin kanvaasin objektien piirtämiseen. Tässä tapauksessa kanvaasi on 2D-pikseliryhmä, jossa jokainen pikseli voidaan värjätä erikseen. Tietokonegrafiikassa yleinen värimuoto on RGBA, joka käyttää yhteensä neljä tavua pikseliä varten, mutta koska kanvaasissa on vain 16 väriä terminaalin rajoitusten vuoksi, voidaan käyttää vain yhden tavun pikseliä kohti. Sen vuoksi täytyy määritellä, miten tätä tavua käytetään esittämään 16:ta väriä. Yksinkertainen ratkaisu on käyttää Microsoftin määrittämiä värejä, jotka ovat välillä 0-15. Voidaan myös määrittää värin läpinäkyväksi, jos se on enemmän kuin 15. Kanvaasi tarvitsee myös PutPixel-funktion, joka muuttaa pikselin värin annetuilla x- ja y-koordinaateilla. Sen avulla voidaan jo piirtää pisteitä. [2.]

Yksinkertainen kanvaasi:

canvas = Color[][]

PutPixel(x, y, color):

canvas[x][y] = color

## Rasterointi

Rasterointi on prosessi, jossa vektoripohjainen kuva tai vertisepohjainen 3D-objekti muunnetaan pikseleiksi. Se on paljon nopeampi renderöintiprosessi verrattuna muihin, kuten säteenseurantaan, mutta ei anna suoraan tietoa siitä, minkä värinen pikselin tulisi olla. Siksi rasterointi, erityisesti 3D-objektien rasterointi, yhdistetään usein pikselishadereihin jotka päättävät pikselin lopullisen värin. [3.] Koska meillä on vain 16 väriä, varjoja ei piirretä ja pikseleiden värit tulevat suoraan tekstuureista.

### Viivojen rasterointi

Yksinkertaisin geometrinen muoto piirtää pisteen jälkeen on viiva, joten aloitetaan siitä. Viivat esitetään usein kaavalla, y = mx + b, jossa x ja y ovat koordinaatit, m on kaltevuus eli y:n muutos x:n suhteen, ja b on pystysuora siirtymä eli y-koordinaatti, jossa viiva leikkaa y-akselin. Viivan piirtäminen tällä kaavalla onnistuu iteroimalla jokaisen x-sijainnin yli ja vastaavan y-arvon piirtämisellä. Tämä funktio pitäisi kuitenkin olla muodossa DrawLine(x0, y0, x1, y1), joten kaltevuuden laskemiseksi voimme käyttää m = (y1 - y0) / (x1 - x0). Lisäksi, koska tiedämme lähtökohdan, b ei ole tarpeellinen, joten voimme ohittaa sen. [2.] Algoritmi näyttäisi suunnilleen tältä, missä x ja y ovat kokonaislukuja:

DrawLine(x0, y0, x1, y1):

//Slope is rise/run

m = (y1 - y0) / (x1 - x0)

y = y0

//For each x position, plot the corresponding y

for x from x0 to x1:

PutPixel(x, y)

y += m

Tämä on hyvin yksinkertainen funktio, joten siinä on useita ongelmia. Ensiksikin se ei toimi pystysuorilla viivoilla, koska silloin m:n laskemisessa tulisi jakaa nollalla. Toiseksi se ei piirrä viivaa oikein, jos kaltevuus on suurempi kuin 1, koska se piirtää vain yhden pikselin per x-koordinaatti. Kolmanneksi, jos x0 on suurempi kuin x1, viiva piirretään väärään suuntaan. Voidaan korjata kaksi ensimmäistä ongelmaa tekemällä kopion funktiosta, joka piirtää viivan y-akselin perusteella, ja käyttämällä tätä toimintoa, jos kaltevuus on enemmän kuin yksi. Toinen ongelma on myös helppo korjata vaihtamalla alku- ja loppupisteet, jos ne eivät ole oikeassa järjestyksessä. [2.]

DrawLineY(x0, y0, x1, y1):

//Make sure starting point is before ending point

if y0 > y1:

swap(x0, x1)

swap(y0, y1)

//Slope is run/rise

m = (x1 - x0) / (y1 - y0)

x = x0

//For each y position, plot the corresponding x

for y from y0 to y1:

PutPixel(x, y)

x += m

Nämä kaksi funktiota yhdessä olisi jo toimiva DrawLine-funktio. Tätä algoritmia voidaan kuitenkin vielä parantaa, koska käytämme liukulukua m:lle, mikä aiheuttaa monta kallista liukuluvun jako- ja pyöristysoperaatioita. Olisi perusteltua päästä eroon niistä, jotta toimintomme toimisi nopeasti CPU:lla. Tätä varten voimme käyttää Bresenhamin viiva-algoritmia. Se on paras viivanpiirtoalgoritmi tähän tarkoitukseen, koska se toimii millä tahansa viivalla, ja on optimoitu käyttämään vain kokonaislukuja [4]. Sen suurin haittapuoli on anti-aliasingin puute, mutta koska sitä ei voida hyödyntää tässä, sen puute ei ole ongelmallista.

Bresenhamin viiva-algoritmi toimii seuraamalla kertynyttä virhettä viivan todellisessa y:ssä ja piirretyssä y:ssä jokaisessa x-sijainnissa. Kun pikseli on piirretty, virhe kasvatetaan kaltevuudella. Seuraavaksi algoritmi päättää, lisätäänkö piirrettyä y:tä 1:llä virheen määrän perusteella: jos virhe on yli 1/2, y:tä tulisi lisätä ja virhettä pienentää, jolloin tällä tavalla piirretään aina lähimmän mahdollisen pikselin suhteessa todelliseen y-koordinaattiin. [4.]

DrawLine(x0, y0, x1, y1):

m = (y1 - y0) / (x1 - x0)

y = y0

accumulatedError = 0.0

//For each x position, plot the corresponding y

for x from x0 to x1:

PutPixel(x, y)

accumulatedError += m

if accumulatedError > 0.5:

y += 1

accumulatedError -= 1.0

Tässä toteutuksessa käytetään vielä liukulukuja. Jotta saadaan funktio laitettua muotoon, joka käyttää vain kokonaislukuja, on muutettava kaksi riviä: *accumulatedError += m* ja *if accumulatedError > 0.5*.

*accumulatedError += m* voidaan kirjoittaa uudelleen seuraavasti: *accumulatedError = accumulatedError + dy / dx*, jossa dx ja dy ovat x0 - x1 ja y0 - y1.

Tämän jälkeen voidaan poistaa murtoluku: *dx \* accumulatedError = dx \* accumulatedError + dy*

Seuraavaksi voidaan poistaa murtoluvun riviltä *if accumulatedError > 0.5: if accumulatedError \* 2 > 1*

Nyt voidaan välttää \* 2 ja \* dx laskemisen kirjoittamalla yhtälöt seuraavasti: *2 \* dx \* accumulatedError = 2 \* dx \* accumulatedError + 2 \* dy*, ja *if 2 \* dx \* accumulatedError > dx*. Tämän avulla voimme ryhmitellä *2 \* dx \* accumulatedError* yhdeksi kokonaislukumuuttujaksi, nimeltään accumulatedErrorInt. [4.]

Optimoitu funktio:

DrawLine(x0, y0, x1, y1):

dx = x1 - x0

dy = y1 - y0

y = y0

accumulatedErrorInt = 0

//For each x position, plot the corresponding y

for x from x0 to x1:

PutPixel(x, y)

accumulatedErrorInt += 2 \* dy

if accumulatedErrorInt > dx:

y++

accumulatedErrorInt -= 2 \* dx

Tässä on vielä sovellettava kaikki alkuperäisen algoritmin korjaukset sekä otettava huomioon negatiivinen kaltevuus pienentämällä x:ää tai y:tä kasvattamisen sijaan. Valmis funktio näyttää tältä:

DrawLine(x0, y0, x1, y1):

dx = x1 - x0

dy = y1 - y0

//Slope is < 1

if abs(dx) > abs(dy):

//If slope is positive increment y, else decrement

yi = 1

if dy < 0:

yi = -1

dy = -dy

//Make sure starting point is before ending point

if x0 > x1:

swap(x0, x1)

swap(y0, y1)

y = y0

accumulatedErrorInt = 0

//For each x position, plot the corresponding y

for x from x0 to x1:

PutPixel(x, y)

accumulatedErrorInt += 2 \* dy

if accumulatedErrorInt > dx:

y += yi

accumulatedErrorInt -= 2 \* dx

//Slope is >= 1

else:

//If slope is positive increment x, else decrement

xi = 1

if dx < 0:

xi = -1

dx = -dx

//Make sure starting point is before ending point

if y0 > y1:

swap(x0, x1)

swap(y0, y1)

x = x0

accumulatedErrorInt = 0

//For each y position, plot the corresponding x

for y from y0 to y1:

PutPixel(x, y)

accumulatedErrorInt += 2 \* dy

if accumulatedErrorInt > dx:

x += xi

accumulatedErrorInt -= 2 \* dx

### Kolmioiden rasterointi

Seuraava geometrinen muoto, joka halutaan piirtää, on kolmio, koska useimmat 3D-mallit koostuvat kolmioista. Kolmio muodostuu kolmesta kärkipisteestä, joita kutsutaan nimillä v0, v1 ja v2. Koska meillä on 2D-kanvaasi, nämä kärkipisteet koostuvat vain x- ja y-koordinaateista. Voidaan käyttää DrawLine-funktiota kolmion piirtämiseen piirtämällä viivat, jotka yhdistävät kärkipisteet: [2.]

DrawWireframeTriangle(v0, v1, v2):

DrawLine(v0, v1)

DrawLine(v1, v2)

DrawLine(v2, v0)

Tämä funktio piirtää vain rautalankakolmion, mikä tarkoittaa, että vain sen reunat piirretään. Tarvitsemme myös funktion täytetyn kolmion piirtämiselle. Yksinkertainen tapa tehdä tämä on piirtää kolmio täysin vaakasuorista viivoista. Tätä varten voidaan käydä läpi jokaisen y-position kolmion ylä- ja alapisteiden välillä ja piirtää viivan vasemmalta oikealle: [2.]

for y from topY to bottomY:

rightBound, leftBound = CalculateBounds()

DrawLine(rightBound, y, leftBound, y)

TopY:n ja bottomY:n laskemiseen, voidaan lajitella pisteet ylhäältä alas ennen piirtämistä. Vaikea osa tästä funktiosta on oikean ja vasemman rajan laskeminen. Ratkaisemaan tämän voidaan päättää, että x-rajat määritellään viivoilla [v0, v1], [v1, v2] ja [v2, v0], missä yksi näistä viivoista on koko raja ja toinen puoli koostuu jäljellä olevista viivoista. Koska pisteet ovat lajiteltu, tiedetään että [v2, v0] on aina jatkuva puoli, kun taas [v0, v1] ja [v1, v2] muodostavat segmentoidun puolen. [2.] X-rajan laskemiseen täytyy interpoloida viivan x-sijainti jokaisessa y-sijainnissa:

InterpolateX(v0, v1):

m = (x1 - x0) / (y1 - y0)

x = x0

xPositions = []

xPositions.PushBack(x)

//For each y position, get the corresponding x

for y from y0 to y1:

x += m

xPositions.PushBack(x)

Tämä funktio palauttaa listan x-sijainneista jokaiselle y-sijainnille. Täytyy vielä yhdistää segmentoidun puolen listat ja selvittää, mikä luettelo on vasen raja ja mikä oikea. Segmentoituun listaan tulee myös kaksi samaa y-sijaintia siinä kohdassa, missä sen kaksi viivaa kohtaavat, joten yksi niistä on poistettava. Kun kaikki tämä yhdistetään, meillä on yksinkertainen funktio, joka piirtää täytettyjä kolmioita: [2.]

DrawTriangle(v0, v1, v2):

//Sort the vertices in descending y

if y1 < y0: swap(v1, v0)

if y2 < y0: swap(v2, v0)

if y2 < y1: swap(v2, v1)

//Calculate the x coordinate of every edge

topSegmentBounds = Interpolate(y0, x0, y1, x1)

bottomSegmentBounds = Interpolate(y1, x1, y2, x2)

continuousBounds = Interpolate(y0, x0, y2, x2)

//Combine the two lists of the segmented side

topSegmentBounds.PopBack()

segmentedBounds = topSegmentBounds + bottomSegmentBounds

//Check which is left and right

leftBounds = continuousBounds

rightBounds = segmentedBounds

i = round(continuousBounds.length / 2)

if leftBounds[i] > rightBounds[i]:

swap(leftBounds, rightBounds)

//Draw each horizontal line

for y from y0 to y2:

for x from leftBounds[y - y0] to rightBounds[y - y0]:

PutPixel(x, y)

Tämä on verrattain yksinkertainen funktio, ja vaikka se ei ole optimoitu, se toimii tähän tarkoitukseen. Huomattavaa on, että emme käyttäneet DrawLine-funktiota. Tämä johtuu siitä, että viivamme ovat aina vaakasuoria, joten voidaan tehdä optimoidun viivan-piirto-toiminnon tähän tarkoitukseen.

## Teksturointi

Lähteet

1. Windows Console Documentation. Microsoft. 10.30.2020. Saatavilla: https://learn.microsoft.com/en-us/windows/console/
2. Gambetta G. Computer Graphics from Scratch. 4.2021. Saatavilla: https://gabrielgambetta.com/computer-graphics-from-scratch/
3. Brian C. What’s the Difference Between Ray Tracing and Rasterization. [Internet]. [viitattu 7.3.2024]. Saatavilla: https://blogs.nvidia.com/blog/whats-difference-between-ray-tracing-rasterization/
4. Kennedy J. Bresenham's Integer Only Line Drawing Algorithm. 2012. Saatavilla: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=eb6f1c1f6ee1baf5fdb426af36f575b543ca7f4e