SIGB Opgave 3:
Computer graphics

Introduktion og problembeskrivelse

Rigtig meget grafik er opbygget af polygoner, og polygoner kan let opbygges ved brug af trekanter. For at disse trekanter kan blive interessante at se på er det essentielt at man kan generere pixels der kan repræsentere dem. Det er også essentielt at trekanterne kan udfyldes. For at optimere 3D-grafik kan man eliminere alle de polygoner der ikke kan ses, og derved reducere tegneoperationer.
Et andet utroligt vigtigt element når det kommer til 3D-grafik, er også at der er et lys-element, da det er essentielt hvis man vil give dybde til objekter.

Vi har i dette projekt arbejdet med:

- Generering af punkter for en trekants linjer ved brug af Bresenhams-linje algoritme
- Udfyldning af punkter ved brug af scan-line algoritme
- Identificering af synlige objekter i 3D-rum
- 3D-lys/skygger på trekanter ved brug af Phong og Gouraud shading

Triangle renderer

For at kunne tegne en trekant er det nødvendigt at have de tre kanter der udspænder trekanten. For at kunne tegne linjerne der bliver udspændt af kanterne, er det nødvendigt at identificere de pixels der skal ligge imellem. En effektiv algoritme der kan tegne denne linje kunne f. eks. være Bresenham, en linje-algoritme der har eksisteret siden 1962.

Bresenham's line algorithm

Bresenhams algoritme er en effektiv rasteriseringsalgoritme som konverterer linjer til pixelpositioner ved hjælp af heltalsaritmetik. Fordelen ved denne algoritme er netop fraværet af floating point-udregninger og afrundinger, operationer som generelt er tungere at processere end heltalsoperationer.
Idéen ved algoritmen er, at man ved at kende linjens endepunkter, (x_0,y_0) og (x_n,y_n), kan udregne hvilke pixelpositioner linjen skal repræsenteres ved. Dette sker ved trinvist at udregne forskellen mellem x- og y-værdierne for to på hinanden følgende punkter (sammenholdt med forskellen mellem endepunkternes koordinater), hvorefter man kan bestemme hvilken pixel der skal inkluderes i den grafiske repræsentation af linjen.

Følgende forklaring gælder for linjer med hældningskoefficienten m, for hvilken $0 \leq m \leq 1$

gælder.
Forestiller vi os at vi har en linje der går fra startpunktet (x_0,y_0) til endepunktet (x_n,y_n), udregner vi indledningsvis ∆x og ∆y:

$$[?][?][?]x = x_n - x_0$$
$$[?][?][?]y = y_n - y_0$$

Herefter kan vi udregne den indledende beslutningsparameter, p_0, for algoritmen. Denne defineres som følger:

$$p_0 = 2[?][?][?]x - [?][?][?]y$$

Denne gælder dog kun for det første trin. For trin *k+1* gælder følgende:

$$p_{k+1} = p_k + 2[?][?][?]y - 2[?][?][?]x(y_{k+1} - y_k)$$

Fælles for beslutningsparameteren, uanset om det er på trin 0 eller trin *k*, er, at hvis $p_k < 0$, så findes den næste pixel som skal inkluderes i linjens repræsentation på positionen (x_k+1, y_k). TIlsvarende gælder det, at hvis $p_k \geq 0$, så findes den næste pixel på positionen (x_k+1, y_k+1). Dette illustreres i nedenstående illustration, hvor algoritmen skal beslutte om pixel *E* eller *NE* skal inkluderes i linjens repræsentation. Her vil det gælde, at hvis beslutningsparameteren er mindre end 0, så skal pixel *E* vælges, ellers skal pixel *NE* vælges. Denne proces gentages indtil $x_k = x_n$ og $y_k = y_n$, hvorefter alle pixels der skal indgå i linjens repræsentation er fundet.

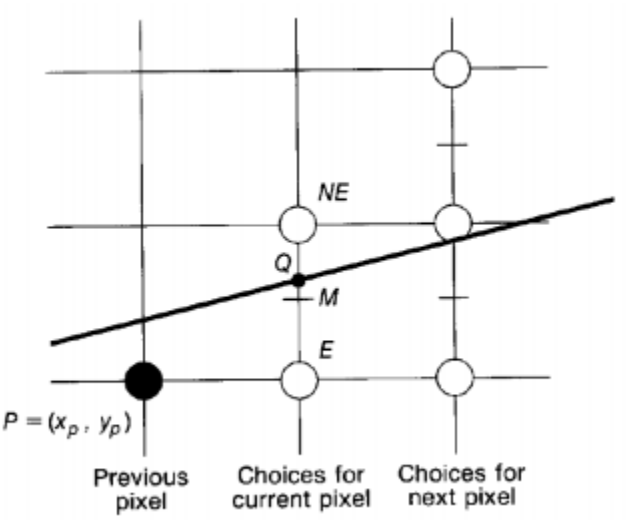
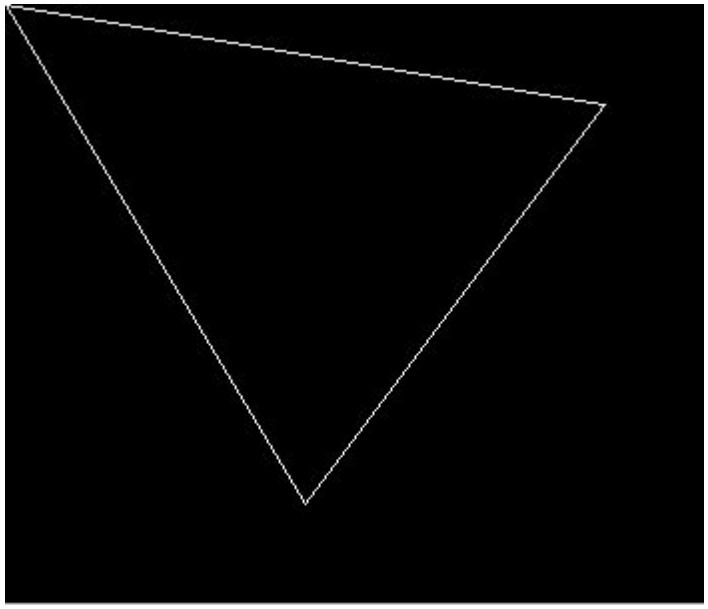


Illustration af beslutningsscenariet for Bresenhams algoritme [3]

For at tegne en trekant med Bresenham, skal vi have tre punkter: a, b og c. Der skal herefter tegnes 3 linjer, mellem a og b, a og c og b og c.
Da vi arbejder i et pixelrum skal negative punkter også blive negeret. Resultatet af trekanten med koordinaterne a = (0, 100), b = (150, -100), c = (-150, -150) kan ses nedenunder.
(Negative koordinater negeret: a = (151, 251), (301, 51), (1, 1))



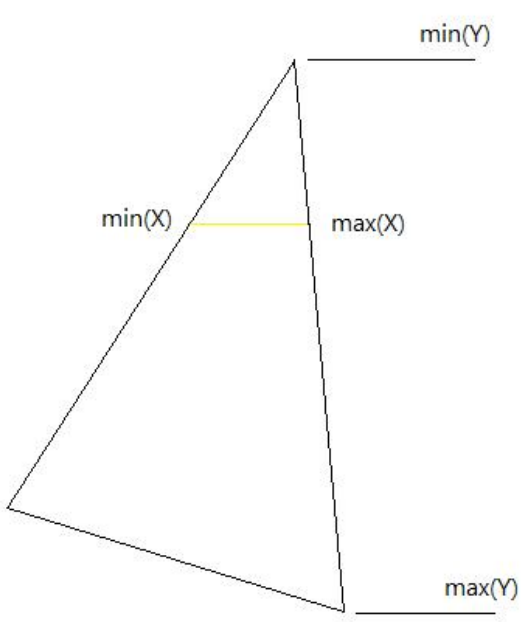
Trekant tegnet ved brug af Bresenham

Scanline filling

Scanline filling er en effektiv fremgangsmåde der “låser” enten y- eller x-aksen og itererer over punkterne mellem kanterne på trekanten. Alle identificerede punkter der ligger inden for trekanten udfyldes herefter med en given farve.

Implementation

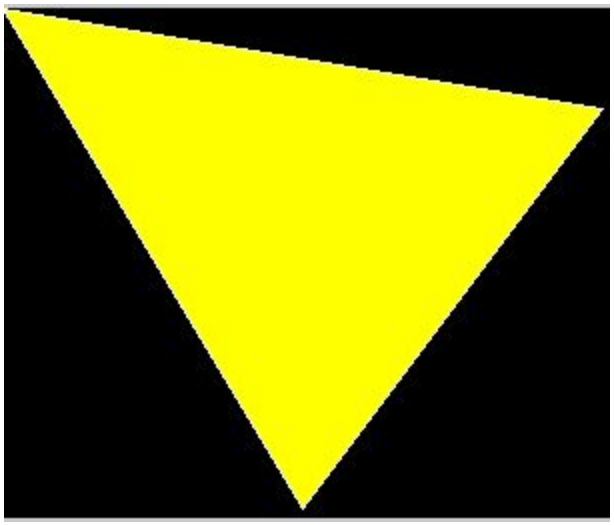
Vi benytter punkterne genereret af Bresenhams algoritme til at finde minimum og maximum for trekantens y-koordinater. Vi itererer herefter mellem disse ekstrema på y-aksen, og for hver y-værdi udregnes således en scanline. For hver scanline finder vi de tilsvarende x-punkter for alle tre linjer i trekanten, og benytter hhv. minimum og maksimum af disse x-værdier til at identificere de pixels som skal udfyldes. Herefter udfyldes alle disse pixels med input-farven. Princippet er illustreret i tegningen nedenfor.



Iterer mellem min(Y) og max(Y). Identificer alle tilsvarende x-punkter og tegn imellem min(X) og max(X).

Implementation af scanline

Resultatet er følgende:



Resultat af scanline

Visible surface detection

Visible surface detection er en grundlæggende optimering i alle programmer der omhandler visualisering af computergrafik. Teknikken går i alt sin enkelthed ud på ikke at spilde beregningstid på at visualliere overfalder brugeren af programmet ikke ser.
Normalt i computerspil handler dette også om at underlade at tegne hele modeller og områder brugeren ikke ser i det nuværende frame. For denne opgave handler problemstillingen om ikke

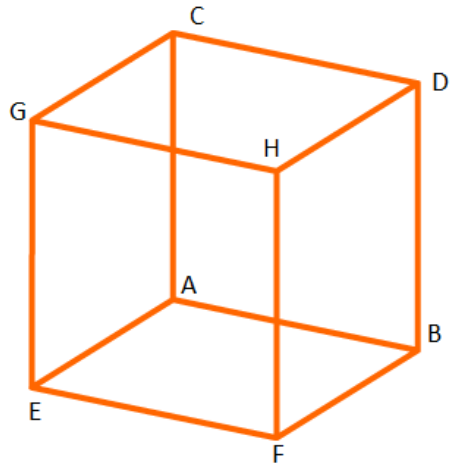
at tegne de overflader af en model som brugeren ikke ser.
Alle overflader som brugeren ikke ser vender naturligvis væk fra kameraet og alle dem brugeren ser vender mod kameraet. Så for at beregne om en overflade kan ses er dette det eneste der skal beregnes.
$v1 =$retningen kameraet peger
$v2 =$ retningen for normalvektoren for det nuværende trekants-polygon i modellen

Vinklen mellem kamera og normalvektoren = $v1 \cdot v2$

Hvis resultatet er under 0 kan overfladen ikke ses, hvis resultatet er over 0 kan overfladen ses.
For vinklen = 0 må der tages et valg i implementationen om dette er en synlig overflade eller ej.

For at tegne en kube skal vi først definere dens punkter. Den simpleste kube der kan defineres, med udgangspunkt i (0, 0, 0) har 8 punkter der er:

Bagerst nederst til venstre (A):	[-1 -1 -1]
Bagerst nederst til højre (B):	[1 -1 -1]
Bagerst øverst til venstre (C):	[-1 1 -1]
Bagerst øverst til højre (D):	[1 1 -1]
Forrest nederst til venstre (E):	[-1 -1 1]
Forrest nederst til højre (F):	[1 -1 1]
Forrest øverst til venstre (G):	[-1 1 1]
Forrest øverst til højre (H):	[1 1 1]



Visalisering af punkterne A-H

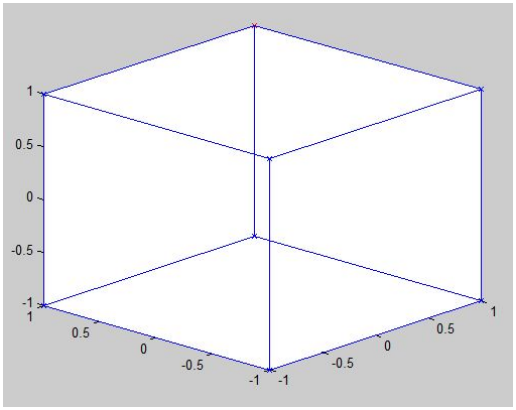
For at kunne tegne hele firkanten, så skal vi skal tegne mellem punkterne
A til B, B til D, D til C og C til A
E til F, F til H, H til G og G til E
G til C, C til D, D til H og H til G
E til A, A til B, B til F og F til E

Når det skal plottes ind i matlab foregår det ved at sammensætte alle x, y og z punkterne for de

fire “runder” rundt om kuben. Det bliver til:

```
X = [-1, 1, -1, -1, -1; -1, 1, -1, -1, -1; -1, 1, 1, 1, -1; -1, 1, 1, 1, -1; -1, 1, -1, -1, -1];
Y = [-1, -1, -1, 1, -1; -1, -1, -1, 1, -1; 1, 1, -1, 1, 1; 1, 1, -1, 1, 1; -1, -1, -1, 1, -1];
Z = [-1, -1, -1, -1, -1; 1, 1, 1, 1, 1; 1, 1, 1, 1, 1; -1, -1, -1, -1, -1; -1, -1, -1, -1, -1];
```

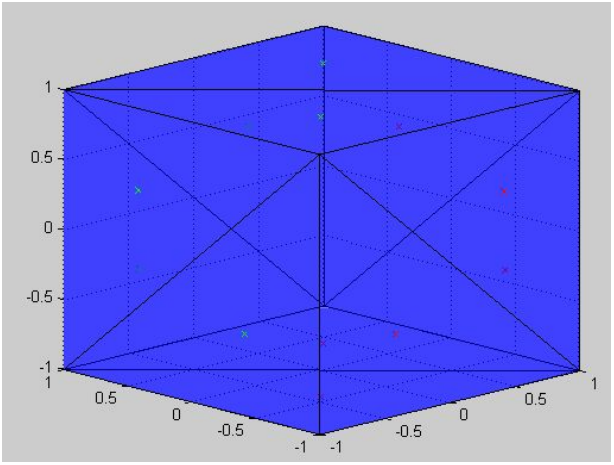
Når dette bliver 3D-plottet ind i matlab bliver resultatet det følgende:



3D-repræsentationen af kuben

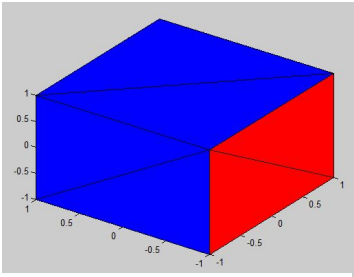
Der blev udleveret kode der kunne opdele vores kube i trekanter. Givet en synsretning, skulle vi kunne definere hvilke dele af trekanten der kunne ses.
Vi indstillede retningen til at kunne se halvdelen af kuben, ved at indstille den til [0.5 -0.5 -1].

Resultatet kan ses nedenunder. Trekanternes synlighed er indikeret i midten af trekanten med et x der enten har farven grøn for synlig eller farven rød for usynlig:

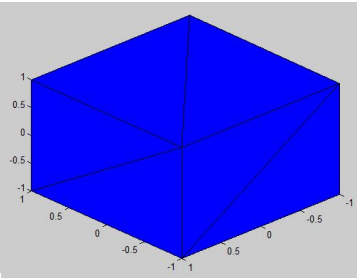


Trekantsopdeling af kuben.
Krydser definerer hvilke trekanter der er synlige

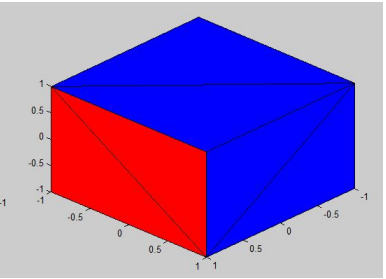
Nedenunder kan kuben ses farvet med en blå for synlig, og rød for usynlig. Kuben bliver roteret så man kan se den fra forskellige vinkler.



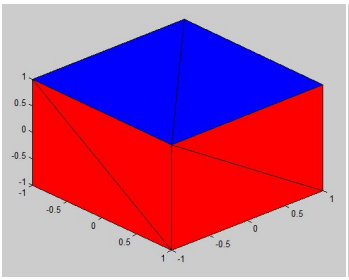
Synlig overflade 1



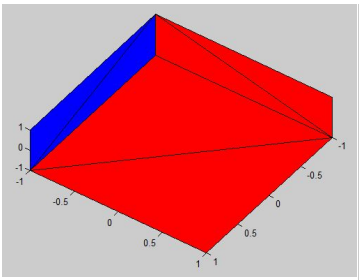
Synlig overflade 2



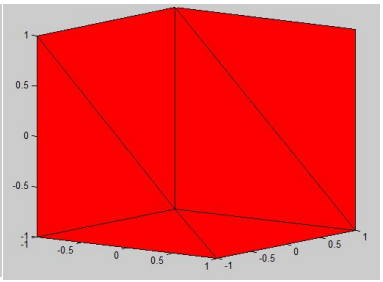
Synlig overflade 3



Synlig overflade 4



Synlig overflade 5



Synlig overflade 6

Shading of a triangle

Phong illumination model

Phong illumination model er en lokal illuminationsmodel som definerer lysforholdene for et punkt på en given overflade (f. eks. en trekant). Modellen er sammensat af tre former for lys: ambient, diffust og specular. Vi vil i det følgende antage, at der kun er én lyskilde og at denne er et point-light hvor lys bliver udsendt jævnt i alle retninger.

Diffused light

Diffused light er lys som reflekteres med lige stor intensitet i alle retninger når lys fra lyskilden rammer objektet, uanset fra hvilken vinkel man betragter objektet. Når et objekt reflekterer alt indkommende lys som diffused light kaldes objektet en *ideal diffuse reflector*. Ofte ønsker man dog ikke at alt lys reflekteres, og derfor defineres en diffuse-reflection koefficient, k_d , som bestemmer hvor stor en brøkdel af det indgående lys som reflekteres. Jo tættere denne værdi er på 1.0, des mere lys reflekterer objektet.
Måden lyset reflekteres på influeres dog ikke kun af mængden af lys fra lyskilden, men også lyskildens position i forhold til objektet, defineret ved vinklen mellem lyskilden og objektet. Hvis

vi definerer mængden af lys fra lyskilden som I_{lys} er forholdet mellem vinklen til lyskilden og mængden af lys der rammer objektet, *I,* denne:

$$I = I_{lys} * cos(\theta)$$

Ganger vi herefter diffuse-reflection koefficienten på, fås følgende:

$$I_{diff} = k_d * I_{lys} * cos(\theta)$$

Hvis vi nu definerer vinklen til lyskilden som prikproduktet mellem normalvektoren til et givent punkt i objektet, *N*, og retningsvektoren til lyskilden (jf. Lamberts lov), *L*, får vi da følgende:

$$cos(\theta) = N \bullet L$$
$$I_{diff} = k_d * I_{lys} * (N \bullet L)$$

For dette gælder, at når vinklen mellem normalvektoren og retningsvektoren til lyskilden er større end eller lig 90° (N•L ≤ 0), er så er $I_{diff} = 0$, idet lyskilden da er bagved objektet, og intet lys vil derfor blive reflekteret.

Ambient light

Ambient light er det generelle lysniveau for en scene eller et objekt, og ambient er det samme for alle objekter i scenen. Ambient light kan altså betegnes som scenens baggrundsbelysning; hvis et objekt ikke rammes af lys fra nogen lyskilde, vil det udelukkende reflektere ambient light. Definerer vi ambient light som I_a , samt en ambient light koefficient, k_a , udregnes mængden af ambient light som reflekteres således:

$$I_{amb} = I_a k_a$$

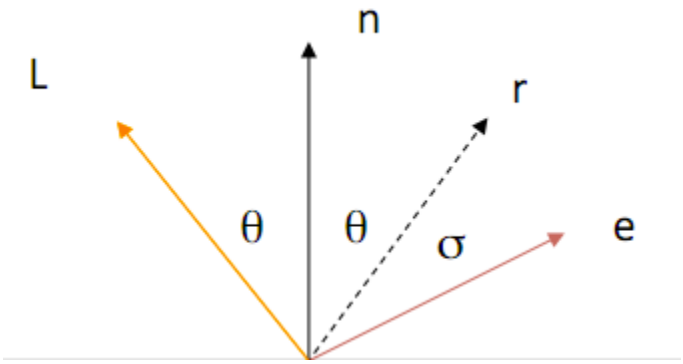
Hvis man arbejder med RGB-billeder defineres k_a en vektor med tre elementer (én pr. farvekanal); arbejder man med gråtonebilleder er denne en værdi mellem 0 og 1.

Specular light

Specular light på et objekt reflekteres direkte videre, uden at noget af lyset fra lyskilden absorberes af objektet. Et objekt som reflekterer alt lys videre fra lyskilden kaldes et perfekt spejl. Retningen på det reflekterede lys afhænger af indfaldsvinklen på lyset fra lyskilden. Definerer vi indfaldsvinklen fra lysklidens lys som θ_i og reflektionsvinklen som θ_r gælder følgende:

$$\theta_i = \theta_r$$

Med andre ord: indfaldsvinklen er lig med reflektionsvinklen. Specular light har dog ikke kun en retning; det reflekteres også over et givet område.



Vektorillustration af specular light [4]

Det reflekterede område defineres som en kegle med centrum i *r* (se ovenfor) og med diameter fra *r* til *e*, hvor *e* er en retningsvektor til kameraet/øjet; vi vil i det følgende kalde *e* for *v* (view), da dette er mere passende i forhold til den endelige formel. Størrelsen på det udspændte område er med til at definere hvor høj intensiteten af det reflekterede specular light er. Jo mindre det reflekterede område er, des større er intensiteten af det reflekterede lys.

Vi ønsker ikke at udlede formlen for specular light her, men vil definere den således:

$$I_{spec} = I_s k_s (r \bullet v)^{\alpha}$$

Her er I_s lyset fra lyskilden, k_s er specular light koefficienten (definerer meget af lyset fra lyskilden der skal reflekteres), *r* er retningsvektoren for reflektionen, *v* er retningsvektoren for kameraet og α er specular-reflection eksponenten. Specular-reflection eksponenten definerer hvor skinnende en overflade er; altså overfladens evne til at reflektere lys. For at emulere overflader som ikke reflekterer meget lys, fx ru/ujævne overflader, specificeres en lav værdi for eksponenten. Tilsvarende specificeres en høj værdi for specular-reflection eksponenten for overflader der reflekterer meget lys.

Den samlede model

Nu hvor vi har defineret alle tre former for lys, kan vi nu definere den samlede formel for Phong illumination model:

$$I = I_{amb} + f_{att} * (I_{diff} + I_{spec})$$

$$I = I_a k_a + f_{att} (I_d k_d n \bullet l + I_s k_s (r \bullet v)^{\alpha})$$

Overordnet set den samlede formel blot summen af hhv. ambient, diffuse og specular light - dog ganges summen af diffuse og specular light med f_{att}. f_{att} er afstanden mellem objektet og lyskilden. Denne skal ikke ganges på ambient light, da ambient light er uafhængig af lyskilden.

Phong illumination model bruges i både Phong og Gouraud shading-algoritmerne, som beskrives i det følgende.

Phong shading

Shading implementationen er en smule anderledes en den der blev fremlagt i undervisningen. Pga. problemer med implementationen fandt vi en anden kilde der implementerede phong shading.

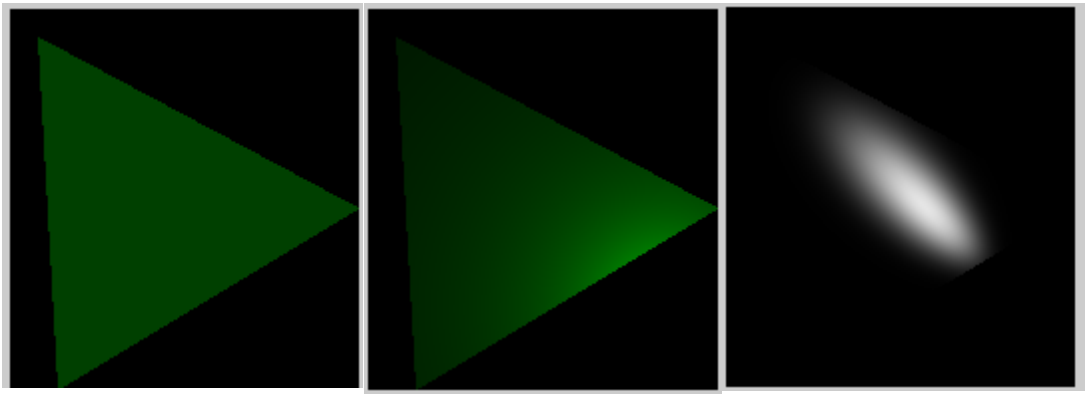
c =Polygonens rå RGB farve

Alle andre parametre er beskrevet i afsnittet om shading teori. Formlen for Phong shadingen brugt i opgaven ses nedenfor [2]. $\otimes$betyder at gange punktvis.

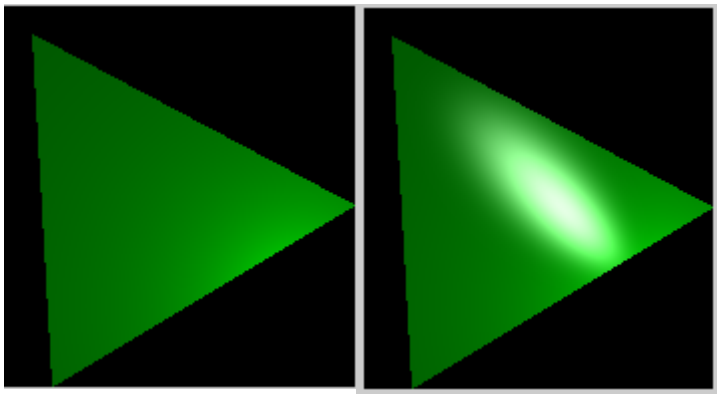
$$Image = c*((I_a \otimes k_a)+f_{att}*((I_d \otimes k_d)*max(L \bullet n,0)+(I_s \otimes k_s)*(max(v \bullet r,0)^a)))$$

For at finde hvilke pixels der skal farves bruger vi Bresenhams algoritme til at finde yndrepunkterne for trekantens x-koordinater.
Mellem de ydrepunkter beregnes Phong shading for hvert (x,y) koordinat indenfor trekanten. Dette giver det flotte resultat der ses nedenfor!

Standard parametre

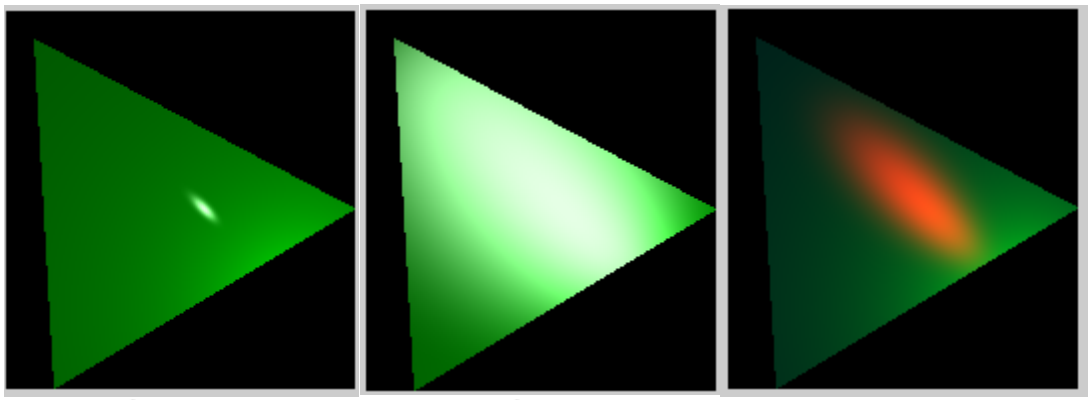


Ambient Light *Diffused Light* *Specular Light*



Ambient, Diffused *Ambient, diffused, specular*

Modificerede parametre:

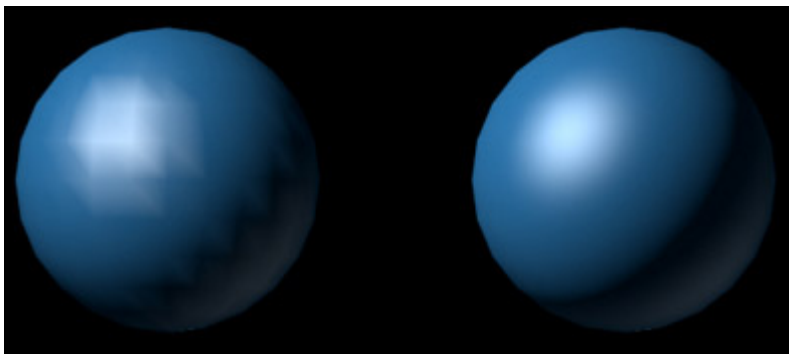


α = 1024 *α = 2* *Andre materiale egenskaber*

Gouraud shading

Gouraud shading er en simpel shading-teknik der er væsentlig hurtigere end Phong shading.
Den bruger Phong-illuminationsmodellen for de tre hjørnepunkter i modellen, men til forskel fra Phong shading beregner den ikke den specifikke illuminering for hver pixel.
I stedet tegnes der linjer mellem de tre hjørnepunkter hvor farven for den enkelte pixel bliver interpoleret ved brug af farverne fra hjørnepunkterne.
Dette bliver gjort for alle polygoner i modellen, hvor antagelsen er at en polygon er en trekant.
I situationen hvor en vertice er en del af flere forskellige polygoner tages der et gennemsnit mellem de tilhørendes polygoners normalvektorer for at finde en vertices normalvektor.

Grundet tidsmangel valgte vi at lade være med at prioritere implementationen af Gouraud-shading, da den giver dårligere resultater end Phong.
En sammenligning mellem Gouraud- og Phong-shading kan f.eks. ses nedenunder [5].



Gouraud-shading (til venstre) og Phong-shading (til højre)

Gouraud-shadings dårlige resultater er dog også dens styrke, da den er væsentligt hurtigere at beregne end Phong-shading.
I praksis er den visuelle kvalitet for dårlig til nutidens kvalitetskrav i spil og grafik.

Konklusion

Det er essentielt at kunne optimere illustrationen af computer-grafik.
Da meget computer-grafik er opbygget af polygoner er det essentielt at kunne tegne trekanter, og Bresenhams-linje-algoritme og scanlines-algoritmen er effektive løsninger.

Til moderne 3D-grafik er det essentielt at det ser godt ud, og Phong's shading-model fungerer rigtigt godt til dette. Jo mere grafisk kvalitet bliver vægtet jo mere komplekst og langsomt bliver det dog at rendere grafikken. Derfor er det essentielt at kunne optimere renderings-sekvensen ved f. eks. at eliminere polygoner som ikke er synlige. og at differentiere mellem synlige og usynlige polygoner ved brug af normal-vektorer fungerer rigtig godt.

Under opgaven fik vi ikke dækket andre teknikker som f. eks. normal-maps, texture mapping og antialiasing. Disse teknikker er grundlæggende for at nå de visuelle krav moderne 3D-spil forventes at have i dag som minimum.

Det var ærgeligt at vi ikke kunne få ligningen fra undervisningen til at virke, men resultatet vi fik med ligningen fra en anden bog [2] gav rigtigt gode resultater, som vi går ud fra er de forventede.

På trods af diverse implementationsproblemer mener vi at vi overordnet har opfyldt målene for opgaven og at de blev udført med tilstrækkelig succes for at opfylde læringsmålene for opgaven.

Referenceliste

1. Computer Graphics with OpenGL, 3/E Donald D Hearn M. Pauline Baker, University of Illinois ISBN-10: 0130153907 ISBN-13: 9780130153906 Publisher: Prentice Hall Copyright: 2004, side 556-577
2. Introduction to 3D Game Programming with DirectX 9.0c: A Shader Approach, Frank Luna, 2006, Wordware, kapitel 10
3. http://www.itu.dk/courses/SIGB/F2011/Slides/GraphicsBasics.pdf - Slide 52
4. http://www.itu.dk/courses/SIGB/F2011/Slides/Illumination&Textures.pdf - Slide 54
5. http://wiki.ohm-hochschule.de/roettger/uploads/Computergrafik/FlatGouraudPhong.jpg