Bachelor i teknisk videnskab (Matematik og teknologi)

Procedural modellering af regulære strukturer

Linda Skovmand (s134729)

Kongens Lyngby 2017

DTU Compute
Institut for Matematik og Computer Science
Danmarks Tekniske Universitet

Matematiktorvet
Bygning 303B
2800 Kongens Lyngby, Danmark
Tlf. +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Resume

With additive layer manufactering (or 3D printing) objects are created layer-by-layer, which results in a staircase effect. This staircase effect depends on the shape of the object and the direction in which it is printed. This report seeks to model this staircase effect, and to implement an interactive interface, which would make it possible for the user to control the layer thickness and printing direction.

The aim of this report is to study how regular structures, like this staircase effect, can be procedurally modelled by combining simple functions and by procedurally using *bump* or *normal mapping*, in a data-free way. It is sought to model photorealistic renderings of this staircase effect, which are compared to actual images of 3D-printed objects.

The staircase effect is modelled by a simple algorithm, which assumes that the y-axis is the printing direction, such that the y-coordinate of a given point's normal can be modified based on the placememt of the camera according to that point's placement between 2 layers.

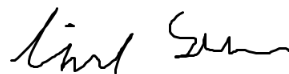
As these regular structures produce aliasing artifacts in excess, an anti-aliasing algorithm is implemented. This algorithm is a screen-space based super-sampling algorithm, which samples in an area of 2x2 pixels around the original fragment, and then invert the graphics pipeline transformations to get to the space where the procedural operations are performed, so that the color of the given sample can be computed. A shader which uses this anti-aliasing algorithm and the procedural modelling of the staircase effect is included in a paper about augmented reality interfaces for additive manufacturing, which can be found in the included zip-file [Eiriksson et al, 2017].

Forord

Procedural modellering af regulære strukturer er en rapport udarbejdet på Danmarks Tekniske Universitet, under instituttet DTU Compute. Rapporten, der repræsenterer 15 ECTS point, er den afsluttende opgave der skal skrives for at opnå en B.Sc. i Matematik og Teknologi, og er blevet udarbejdet i tidsperioden 28. september 2016 - 20. januar 2017.

I forbindelse med udarbejdelsen af denne rapport udsendes en speciel tak til vejlederen Jeppe Revall Frisvad for rådgivning og støtte. Derudover skal der også lyde en stor tak til de evigt hjælpsomme personer på *image*-gangen.

Kongens Lyngby, 31. januar 2017



Linda Skovmand (s134729)

Indhold

Resume i

Forord iii

Indhold v

1 Introduktion 1

2 Notation 3

3 Teori 5
3.1 Overfladerepræsentation 5
3.2 Lys-materiale interaktion 6
3.3 Procedural modellering og cases 14
3.4 Realtids grafik-pipeline 27
3.5 Anti-aliasing 32
3.6 Animation 37

4 Implementering 43
4.1 WebGL 43
4.2 Interaktivitets interface 47
4.3 Alpha blending 48

5 Resultater 51
5.1 Anti-aliasing 51
5.2 Mursten 54
5.3 Tråd 57
5.4 Lag-inddeling 60

6 Diskussion 71
6.1 Anti-aliasing 71
6.2 Mursten og tråd 71
6.3 Lag-inddeling 72
6.4 Interaktiv tilpasning 73

7 Konklusion **75**

8 Referencer **77**

A Appendix **81**

A.1 Kode 81

A.2 Projektmotivation og projektplan 82

KAPITEL 1

Introduktion

3D-printning er en teknologi der har været i udvikling i rigtig mange år, men efter årtusindskiftet kom et boom indenfor teknologien, der bl.a. banede vejen for desktop printere. Desktop printere er endnu ikke hvermandseje, men almindelige forbrugere har med desktop printere fået muligheden for at 3D-printe, og dermed tilfredsstille deres skabertrang.

Som forbruger vil man ofte opleve at det 3D-printede objekt og den digitale 3D-model ikke stemmer overens. Dette skyldes at 3D-printere opbygger modeller lag for lag, med en given lagtykkelse. Alt efter præcisionen af den givne printer, spænder sådanne lagtykkelser gerne mellem 20 og 200 μm, hvor de aller tyndeste lag, oftest kun ses i industri-printere. Denne fremstilling vha. lag, vil resultere i en såkaldt trappelignende effekt [He et al, 2014], hvor objektets facon og orienteringen der anvendes ved printningen vil have betydning for graden af denne trappe-effekt.

Givet en digital model, kunne det derfor være ønskeligt at kunne generere en realistisk model af hvordan resultatet af sådan en 3D-printning ville se ud. At tilpasse en given overflade manuelt ville være fuldstændig uoverkommeligt, og hvis det virkelig skal kunne komme en forbruger til gode, bør denne modellering også genereres i realtid, så forbrugeren kan vurdere resultatet af printningen, med specifikke lagtykkelser og print-retninger.

Da en sådan trappe-effekt er en regulær struktur, kan denne modelleres vha. procedurale teknikker. Med sådanne teknikker kan et specifikt udseende opbygges datafrit vha. matematik og algoritmer, frem for ved eksempelvis at anvende et teksturbillede, et *bump-map* eller andre pladskrævende metoder.

I denne rapport vil det undersøges hvordan regulære strukturer, som eksempelvis denne trappe-effekt, kan modelleres vha. simple procedurale teknikker. Alle de problemer der opstår ved generering af sådanne strukturer, vil også håndteres, hvor især repræsentationen af sådanne strukturer på et rasterdisplay leder til et stort problem. Det vil med denne modellering forsøges at komme så tæt på et fotorealistisk resultat som muligt (naturligvis begrænset af at det skal være i real-tid), hvor renderingerne vil sammenlignes kvalitativt med fotografier af 3D-printede objekter taget i et lyskontrolleret miljø, hvor et eksempel ses i figur 1.1. Derudover vil der også implementeres

en interaktiv brugerflade, som gør det muligt at anvende forskellige print-retninger og lagtykkelser, samt at navigere det virtuelle kamera og objektet rundt i den virtuelle scene i real-tid.

Figur 1.1: Fotografi af 3D-printet objekt, taget i et lyskontrolleret miljø.

KAPITEL 2

Notation

Matematisk notation
Den matematiske notation, der er blevet anvendt, er beskrevet herunder:

Skalarer og skalarfunktioner er skrevet i kursiv:

$$x = 2.21$$
$$f(x) = 2x + 3$$

Vektorer (punkter og retningsvektorer) og vektorfunktioner er skrevet i fed:

$$\mathbf{v} = [x, y, z]^T$$
$$\mathbf{f}(\mathbf{v}) = \mathbf{v} + \mathbf{v}$$

Matricer og matrixfunktioner er skrevet i fed, med store bogstaver:

$$\mathbf{A} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$
$$\mathbf{f}(\mathbf{A}) = \mathbf{A}^{-1}$$

Generel notation
Engelske fagudtryk der ikke kan oversættes direkte til dansk, vil noteres på engelsk i kursiv, f.eks. ”Der anvendes *anti-aliasing*.”

KAPITEL 3

Teori

3.1 Overfladerepræsentation

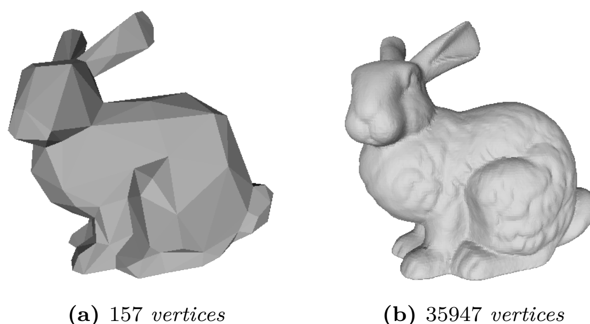
Den mest anvendte repræsentation af overflader indenfor computergrafikken, er det såkaldte trekantsmesh, som vha. en masse trekanter samlet langs hinandens kanter, kan forme en overflade. Trekantsmesh, anvendes bl.a. frem for andre polygonmesh, da en trekant altid vil være simpel, konveks og bestå af hjørner, som alle er indeholdt i den samme plan, hvilket medfører at dens indre er veldefineret [Angel, 2014]. Derudover kan enhver polygon også altid inddeles i trekanter, som er den mindst mulige polygon. Mængden af detaljer der kan repræsenteres med et trekantsmesh er afhængig af antallet af trekanter, og det samme er renderingstiden. I figur 3.1 ses 2 repræsentationer af det samme objekt, men med et forskelligt antal af trekanter.

Et trekantsmesh er defineret ved at sæt af såkaldte *vertices*, som er en form for datastruktur, der består af forskellige attributter, hvoraf positionen i rummet altid er indeholdt, i mens en dertilhørende normal, farve og teksturkoordinaterne (u, v) er 3 andre mulige attributter. *Vertex* normalen anvendes til *shading*, som vil gennemgås i nedenstående afsnit. Derudover kræves der naturligvis også information som specificerer hvorledes de givne *vertices* skal forbindes til trekanter. Rækkefølgen hvormed en trekant samles, bestemmer hvorvidt der er tale om en *front facing* eller *back facing* trekant, hvilket kan være vigtig information, i fht. at opnå en hurtigere renderingstid. En god mesh-repræsentation er først og fremmest pladsbesparende, da komplekse objekter som repræsenteres af rigtig mange trekanter, kan resultere i en såkaldt beregningsmæssig *bottleneck* på CPU'en, hvor det i sær er overførslen af data mellem CPU'en og GPU'en, som er en begrænsende faktor [Shontz, 2012]. Det er altså derfor ønskeligt at reducere mængden af data som skal overføres så meget som muligt, ved enten at reducere antallet af *vertices* eller mængden af data gemt for hvert *vertex*.

Et trekantsmesh kan repræsenteres på flere forskellige måder. Eksempelvis kan der anvendes en *vertex* repræsentation, hvor en liste af *vertices* grupperes i grupper af 3, hvor hver gruppe repræsenterer en trekant. Denne repræsentation er dog ikke særlig pladsbesparende, da det samme *vertex* kan være delt af mange trekanter, og derfor gemmes mange gange. Indenfor computergrafikken anvendes i stedet ofte en indekseret *vertex* repræsentation [Hughes et al, 2014], som i sin mest simple form anvender en *vertex* liste hvor alle *vertex* positioner i meshet er gemt en enkelt gang, samt en

indeksliste som beskriver hvordan disse skal forbindes til trekanter. Hvis man ønsker at designe sit mesh, således at det skal kunne modellere en glat overflade bør *vertex* normalen for et givent punkt altid være den samme, uafhængigt af hvilken trekant den hører til for den givne beregning, hvilket er konsistent med den simple indeks repræsentation. Hvis man derimod eksempelvis ønsker at modellere et kantet udtryk som det der ses i figur 3.1a, vil det være nødvendigt at kunne anvende forskellige *vertex* normaler for det samme punkt. Mængden af *vertices* kan eksempelvis også reduceres ved at simplificere meshet, hvor man f.eks. kan reducere mængden af trekanter, når distancen mellem objektet og kameraet øges (et koncept kaldet *level of detail*).

Da repræsentationen af overflader i computergrafikken nu er sat på plads, kan det i næste afsnit undersøges hvordan materialet på sådan et objekt interagerer med lyset i scenen.



(a) 157 *vertices* **(b)** 35947 *vertices*

Figur 3.1: Den kendte *Stanford bunny* repræsenteret vha. trekantsmesh bestående af henholdsvis 157 *vertices* (310 trekanter) og 35947 *vertices* (69451 trekanter). Hvis et *vertex* er indeholdt i flere trekanter, tæller det stadig kun som et enkelt, i fht. disse tal.

3.2 Lys-materiale interaktion

Når man renderer billeder af 3D modeller i realtid, er det vigtigt at anvende en passende repræsentationen af 3D modellerne (trekantsmesh), men det er mindst lige så vigtigt at disse 3D modeller får det ønskede visuelle udseende, som eksempelvis kunne være et fotorealistisk udseende, ved brug af en passende model. Dette visuelle udseende, bestemmes af interaktionen mellem lyskilderne og 3D-objekterne i scenen, som kan beskrives vha. såkaldte lysmodeller. Lysmodeller kan deles ind i de 2 klasser: lokale og globale lysmodeller. Indenfor realtids computergrafikken er det kun den lokale belysning fra en lyskilde der tages hensyn til, hvilket betyder at lyset kun kommer fra denne lyskilde, og ikke er påvirket af andet geometri end det punkt

som det netop ønskes at beskrive interaktionen for [Bærentzen et al]. Altså vil en lokal lysmodel *shade* et givent punkt uafhængigt af andre overflader i scenen. Når lys-materiale interaktionen skal beskrives, må man i sær overveje følgende elementer:

- Lyskilder
- Materiale-egenskaber
- Kameraets/øjets position
- Orientering af overfladen (normalen $\mathbf{n}$)

Der vil i de nedenstående afsnit gennemgås opbygningen af, og elementerne i, en lokal lysmodel, hvor fokus vil være på at vise de forskellige elementers vigtighed for det renderede udseende. Dette giver os nemlig mulighed for at modificere disse elementer, og dermed føre udseendet i en ønskelig retning.

3.2.1 Lyskilder

For at kunne karakterisere et materiales opførsel, må man kunne repræsentere mængden og retningen af det udgående lys, baseret på mængden og retningen af det indkommende lys, hvoraf dette afsnit vil gennemgå det indkommende lys. I fht. *shading* ligningen beskrevet i afsnit 3.2.2, ønsker man om en given lyskilde at kende retningen $\mathbf{w}_i$ (normaliseret), som beskriver retningen fra et givent punkt $\mathbf{x}$ mod lyskilden, samt overflade-irradiansen E_w.

Overflade-irradiansen E_w [W/m^2] beskriver den flux, som strømmer igennem en overflade, der står vinkelret på $\mathbf{w}_i$, pr. arealenhed, og beskriver således lysets generelle lysstyrke. Da lys kan være farvet repræsenteres irradians som en RGB vektor, med ubegrænsede værdier. Belysningen af et givent punkt $\mathbf{x}$ på en overflade, kan så beskrives ved irradiansen E, som beskriver den flux der strømmer igennem en overflade der står vinkelret på punktets normal $\mathbf{n}$. Denne overflade-irradians er afhængig af typen af lyskilde, så i de nedenstående afsnit gennemgås 2, inden for computergrafikken, meget anvendte lyskilder.

Retningsbestemt lys
Retningsbestemt lys, er lys der er produceret af en lyskilde der er placeret så langt væk fra scenen, at det kan antages at det udsendte lys kun vil ramme scenen i form af parallelle lysstråler. Retningsbestemt lys er derfor defineret ved denne ene retning $\mathbf{w}_e$, som alle lysstrålerne antages at have. Denne retning er således givet ved $\mathbf{w}_e = -\mathbf{w}_i$, hvor $\mathbf{w}_i$ er retningen mod lyset. I denne retning udsender lyset en konstant radians, og overflade irradiansen E, for et givent punkt $\mathbf{x}$, kan derfor i følge Lamberts cosinus lov, findes ved at modellere den ønskede spredning, som ses i ligning 3.1. θ_i beskriver vinklen mellem normalen $\mathbf{n}$ og lysets retning $\mathbf{w}_i$, og irradiansen vil derfor være størst når lyset rammer vinkelret ind på overfladen.

$$E = E_w \,\overline{\cos}\,\theta_i = E_w \;\max(\mathbf{n} \cdot \mathbf{w}_i, 0) \tag{3.1}$$

$\overline{\cos}$ indikerer at negative værdier af cosinus-operationen vil sættes til 0. Negative værdier ville nemlig kun opstå hvis vinklen mellem normalen og lyskilden var så stor, at lyskilden befandt sig under overfladen. Da mængden af lys, som rammer et givent punkt er afhængig af normalen, kan blot en ændring af normalen derfor resultere i et udseende, hvor det f.eks. ligner at objektet har *bumps* eller indrykninger, uden faktisk at ændre geometrien, hvilket udnyttes i afsnit 3.3.

Punktlys
Et punktlys udsender en konstant intensitet I_e [W/sr] i alle retninger, rundt om punktet $\mathbf{x}_e$, som beskriver lysets placering (det antages at et punktlys er placeret så langt væk fra scenen, ifht. lyskildens størrelse, at alt lys kan antages at udsendes fra dets centrum). Overflade-irradiansen E_w for et givent punkt $\mathbf{x}$, er afhængig af afstanden d mellem lyskilden og dette punkt, som bestemmes ved $d = ||\mathbf{x}_e - \mathbf{x}||$. Vektoren $\mathbf{w}_i$ mod lyset må derfor givet ved: $\mathbf{w}_i = (\mathbf{x}_e - \mathbf{x})/d$. Vha. afstandskvadratloven kan E_w findes ved [Akenine-Möller et al, 2008]:

$$E_w = \frac{I_e}{d^2} = \frac{I_e}{||\mathbf{x}_e - \mathbf{x}||^2} \tag{3.2}$$

Hvor overflade-irradiansen E, så kan findes som i ligning 3.1. Anvendelsen af denne lov, resulterer oftest i et for mørkt resultat, eller i uønskede artifakter, så inden for computergrafikken erstattes denne oftest af nedenstående udtryk [Angel, 2014]:

$$E_w = \frac{I_e}{a + bd + cd^2} \tag{3.3}$$

Hvor konstanterne a, b og c frit kan vælges for at give et ønskeligt resultat, hvor belysningen er blødere. Et typisk eksempel på et punktlys er en lyspære, og denne form for lys, kan derfor relativt nemt efterlignes i et lyskontrolleret miljø. Punktlys kan udvides til at beskrive andre former for lys, som f.eks. spotlys, og punktbaserede lyskilder danner derfor basis for mange lysmodeller [Hewitt et al, 1991]. Derudover anvendes punktlys ofte også som en approksimation til såkaldte *area* lyskilder, hvis disse er placeret meget langt fra objektet, i fht. de geometriske dimensioner af kilden selv [Gigahertz-optik].

3.2.2 Shading ligning

Shading er den proces, der svarer til at beregne det lys der forlader punktet $\mathbf{x}$ i retningen $\mathbf{w}_o$ (mod kameraet/øjet), baseret på scenens lyskilder og objekternes materialeegenskaber. Lyset L_o der forlader punktet imod retningen $\mathbf{w}_o$, er målt i radians [W/(m^{2}sr)], da dette kan måles af en sensor (altså et øje/kamera) [Akenine-Möller

et al, 2008]. Radians er også beskrevet ved en RGB-vektor, og konserveres langs en stråle i rummet, så længe denne er ublokeret.

Der vil i dette afsnit gennemgås opbygningen af en generel simpel *shader* ligning, som kan anvendes i realtid, og i underafsnit 3.2.2.1 vil forskelle modeller der kan anvendes i denne ligning beskrives. Som nævnt i afsnit 3.2.1, må mængden af indkommende og udgående lys kendes for at kunne beskrive et materiales egenskaber. Fra selvsamme afsnit vides det at indkommende lys er givet ved overflade irradiansen E [W/m^2], i mens det udgående lys er givet ved *exitance* M [W/m^2], som modsat irradians, svarer til det lys der strømmer ud af en overflade. Ratioen mellem *exitance* og irradians, kaldes reflektansen, og er også repræsenteret som en RGB vektor ρ (RGB-vektor med værdier mellem 0 og 1). Forholdet mellem irradians og *exitance* er lineært, således at en fordobling af irradians vil resultere i en fordobling af *exitance*. Oftest opdeles ρ i en spekulær farve ρ_s og en diffus farve ρ_d, således at $\rho = \rho_d + \rho_s$. Dette gøres da *shading* ligninger ofte opdeles i en spekulær og diffus del, hvor den spekulære del repræsenterer lys der reflekteres fra overfladen, i mens den diffuse del repræsenterer lys som først refrakteres ned i materialet, hvori det absorberes og spredes et utal af gange, for til sidst at sende noget af lyset ud i gennem overfladen igen i tilfældige udfaldsvinkler.

Den diffuse *exitance* M_{diff} kan nemt bestemmes da det indkommende lys E, fra retningen $\mathbf{w}_i$, kendes fra ligning 3.1 (hvis der antages at være tale om et retningsbestemt lys) og ratioen mellem indkommende og udgående diffust lys kendes som ρ_d:

$$M_{diff} = \rho_d \odot E_w \, \overline{\cos}\, \theta_i \tag{3.4}$$

Tegnet $\odot$ indikerer at der er tale om en punktvis vektormultiplikation. Ved at antage at den diffuse del repræsenterer lys som spredes uniformt i alle retninger i en hemisfære, kan den diffuse radians L_{diff}, i en specifik retning $\mathbf{w}_o$, bestemmes ved [Akenine-Möller et al, 2008]:

$$L_{diff} = \frac{M_{diff}}{\pi} = \frac{\rho_d}{\pi} \odot E_w \, \overline{\cos}\, \theta_i \tag{3.5}$$

På samme måde kan den spekulære *exitance* findes ved:

$$M_{spek} = \rho_s \odot E_w \, \overline{\cos}\, \theta_i \tag{3.6}$$

Spredningen af den spekulære radians er mere indviklet, da denne bl.a. afhænger af retningen mod kameraet og materialets ruhed. I BRDF-modeller (som vil gennemgås i nedenstående afsnit) er det ofte den spekulære radians som varieres fra model til model. For at kunne færdiggøre opstillingen af *shading* ligningen, vil den spekulære radians derfor blot defineres ved følgende, hvoraf k_s vil udbyttes med forskellige spekulære BRDF-modeller i afsnit 3.2.2.1:

$$L_{spek} = k_s M_{spek} = k_s \rho_s \odot E_w \, \overline{\cos}\, \theta_i \tag{3.7}$$

Hvis det antages at der blot arbejdes med en enkelt lyskilde, så vil den færdige *shading ligning*, som kombinerer de 2 dele for at få den totale udgående radians L_o i retningen $\mathbf{w}_o$, være givet ved:

$$L_o = (\frac{\rho_d}{\pi} + k_s \rho_s) \odot E_w \, \overline{\cos}\, \theta_i \tag{3.8}$$

Da lys er additivt, er det muligt blot at summe over bidraget fra hver lyskilde, hvis det ønskes at arbejde med flere lyskilder.

3.2.2.1 BRDF-modeller

I dette afsnit gennemgås 3 forskellige BRDF-modeller, hvoraf de første 2 er forskellige, men begge meget anvendte, varianter af Phong modellen, i mens den sidste er Kajiya's og Kaj's model for hår. De første 2 vil senere anvendes i et forsøg på at opnå fotorealistiske renderinger af 3D-printede objekter (plastic).

En BRDF (*Bidirectional Reflectance Distribution Funktion*), er en funktion, som beskriver hvordan et givent punkt reflekterer lys, og beskriver dermed materialets udseende. Som navnet antyder, beskrives dette ved hjælp af 2 retninger: retningen af det indkommende lys $\mathbf{w}_i$ og retningen mod kameraet $\mathbf{w}_o$. En BRDF er defineret ved ratioen mellem den udgående radians og den indkommende irradians, og beskrives derfor også som en RGB-vektor:

$$f(\mathbf{w}_i, \mathbf{w}_o) = \frac{dL_o(\mathbf{w}_o)}{dE(\mathbf{w}_i)} = \frac{L_o(\mathbf{w}_o)}{E_w \, \overline{\cos}\, \theta_i} \tag{3.9}$$

Hvoraf den sidste udregning, kun kan udføres ved at antage at der arbejdes med punkt eller retningsbestemte lyskilder [Akenine-Möller et al, 2008]. En sådan BRDF-værdi kan i realiteten have en værdi i mellem 0 og uendelig. Ved at isolere den udgående radians, og sammenligne med ligning 3.8, ses det altså hvordan en BRDF, passer ind i den simple *shading* ligning (hvor der stadig blot anvendes en enkelt lyskilde):

$$L_o(\mathbf{w}_o) = f(\mathbf{w}_i, \mathbf{w}_o) \odot E_w \, \overline{\cos}\, \theta_i \tag{3.10}$$

De 3 førnævnte modeller vil nu gennemgås, hvor der som nævnt vil være fokus på den spekulære del af BRDF'en:

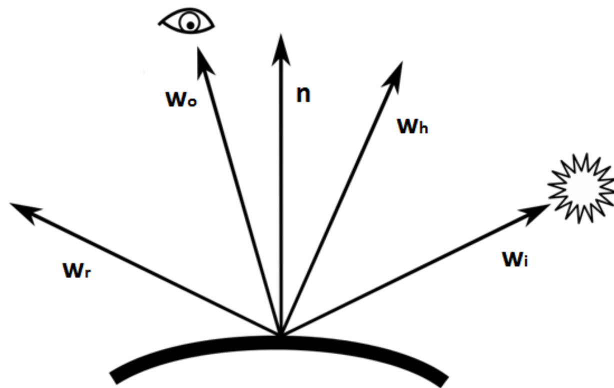
Modificeret Phong model

Den modificerede Phong model er defineret i [Akenine-Möller et al, 2008] ved:

$$f(\mathbf{w}_i, \mathbf{w}_o) = \frac{\rho_d}{\pi} + \rho_s \frac{m+2}{2\pi}(\mathbf{w}_r \cdot \mathbf{w}_o)^m \tag{3.11}$$

De forskellige vektorer der anvendes i denne BRDF, ses illustreret i figur 3.2. Denne model anvender en parameter m, som beskriver overfladens ruhed, en vektor $\mathbf{w}_r = 2(\mathbf{w}_i \cdot \mathbf{n})\mathbf{n} - \mathbf{w}_i$, som beskriver den perfekte reflektionsretning, hvor indfaldsvinkel er lig udfaldsvinkel, samt retningen mod kameraet $\mathbf{w}_o$. Bemærk at der kun bør være et spekulært bidrag hvis prikproduktet er større end 0.

Det bemærkes at den spekulære del, modsat den diffuse, i denne model er afhængig af retningen mod kameraet og retningen mod lyskilden (indirekte ved $\mathbf{w}_r$). Den spekulære del er således størst når retningen mod kameraet $\mathbf{w}_o$ og reflektionsretningen $\mathbf{w}_r$, svarer til den samme retning. Jo større parameteren m er, jo mindre og mere koncentreret bliver de spekulære highlights. Dette skyldtes at enhver retning mod kameraet, som ikke svarer til reflektionsretningen, vil have en cosinus-værdi som er mindre end 1, og derfor hurtigt gå mod 0, når det løftes op i den høje m-værdi.



Figur 3.2: De mest almindelige vektorer, der anvendes i forbindelse med løsningen af en *shading* ligning.

Modificeret Blinn-Phong model

Den modificerede Blinn-Phong model er defineret i [Akenine-Möller et al, 2008] ved:

$$f(\mathbf{w}_i, \mathbf{w}_o) = \frac{\rho_d}{\pi} + \rho_s \frac{m+8}{8\pi}(\mathbf{w}_h \cdot \mathbf{n})^m \tag{3.12}$$

Det bemærkes at denne anvender halvvektoren $\mathbf{w}_h = (\mathbf{w}_i + \mathbf{w}_o)/|(\mathbf{w}_i + \mathbf{w}_o)|$, frem for

reflektionsretningen $\mathbf{w}_r$. Halvvektoren er vektoren halvvejs mellem $\mathbf{w}_i$ og $\mathbf{w}_o$, som det ses i figur 3.2. For denne model bliver den spekulære del mindre, i takt med at vinklen mellem $\mathbf{n}$ og $\mathbf{w}_h$ bliver større, da en vinkel på 0 ville medføre at den perfekt spekulære reflektionsretning og kameraretningen var den samme. For at kunne opnå spekulære highlights af samme størrelsesorden som i ovenstående model, må der anvendes værdier af m, som er 4 gange så store. De spekulære highlights som produceres af denne og ovenstående model er forskellige, og især ved vinkler hvor kamera-retningen og overfladen næsten er parallelle. Ved reflektionsvektor-baserede BRDF'er vil highlightet være cirkulært uafhængigt af synsretningen, i mens halvvektor-baserede BRDF'er vil resultere i aflange highlights ved de førnævnte vinkler [Nielsen et al, 2014].

Kajiya-Kaj BRDF-model for hår

I [Kajiya og Kaj, 1989] præsenteres følgende model, her opdelt i den diffuse og spekulære del:

$$f(\mathbf{w}_i, \mathbf{w}_o)_d = \rho_d \sin(\cos^{-1}(\mathbf{t} \cdot \mathbf{w}_i)) \tag{3.13}$$

$$f(\mathbf{w}_i, \mathbf{w}_o)_s = \rho_s \left((\mathbf{t} \cdot \mathbf{w}_i)\,(\mathbf{t} \cdot \mathbf{w}_o) + \sin(\cos^{-1}(\mathbf{t} \cdot \mathbf{w}_i))\sin(\cos^{-1}(\mathbf{t} \cdot \mathbf{w}_o))\right)^p \tag{3.14}$$

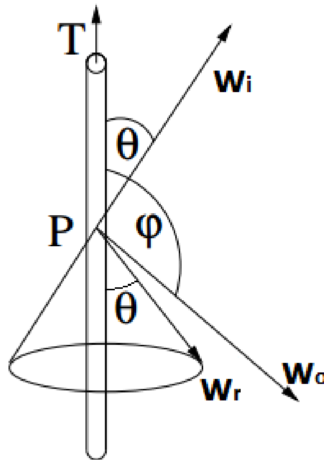
Det bemærkes at i denne model er det tangenten, frem for normalen, som spiller en vigtig rolle, i fht. at definere hvordan lyset reflekteres i retningen $\mathbf{w}_o$. Her antages at et hårstrå kan modelleres som en cylinder, med tangenter der følger hårstrået. Sinus til vinklen mellem 2 vektorer, som beregnes flere gange i ovenstående model, kan med fordel udregnes ved:

$$\sin(\mathbf{t}, \mathbf{w}_i) = \sin(\cos^{-1}(\mathbf{t} \cdot \mathbf{w}_i)) = \sqrt{1 - (\mathbf{t} \cdot \mathbf{w}_i)^2} \tag{3.15}$$

Den diffuse del opnås ved at integrere en diffus Lambert model langs omkredsen af en halv cylinder belyst af lyskilden, hvor udledningen kan findes i [Kajiya og Kaj, 1989]. Denne udledning er baseret på at projektere lysretningen $\mathbf{w}_i$ over på den flade som normalen ligger i, og ender ud med det simple udtryk der ses i ligning 3.13. Det ses heraf at den diffuse del afhænger af sinus til vinklen mellem tangenten og lysretningen, hvilket medfører at denne del vil vil størst når de 2 vektorer er vinkelrette, og mindst når de er parallelle, som det også observeres i virkeligt hår.

Den spekulære del er en modifikation af Phong modellen, som er tilpasset til at beskrive reflektionen fra en tynd cylinder overflade. Da et hårstå er repræsenteret ved en meget lille cylinder, vil cylinderens normaler pege i alle mulige retninger, vinkelret på tangenten. Dette medfører at lyset bør reflekteres i alle mulige retninger rundt om cylinderen, og det reflekterede lys repræsenteres derfor ved en kegle, frem for en retning, som det ses i figur 3.3. Den spekulære del som ses i ligning 3.14, vil derfor være størst når retningen mod kameraet er indeholdt i 'reflektions keglen', hvilket vil resultere i lange highlights som følger fiberstrukturen, som det også ses på hår i

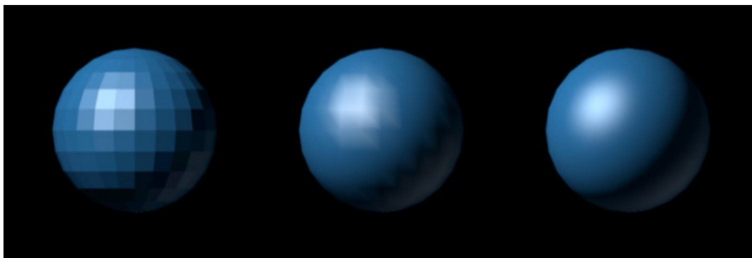
virkeligheden. Værdien p, er en Phong eksponent som ligesom i de øvrige modeller beskriver koncentration og størrelse af de givne hightlights.



Figur 3.3: Geometrien i Kajiya og Kaj's lysmodel.

3.2.3 Shading

Som beskrevet i afsnit 3.1 kan 3D-objekter approksimeres vha. polygoner, eller mere præcist trekantsmesh. For at undgå at objekter får et facetteret udtryk, er det vigtigt at anvende en passende *shading metode*. Når man shader en pixel, vha. en simpel *shader* ligning, findes der 3 typiske *shading* metoder, som her vil gennemgås, og som alle kan ses illustreret i figur 3.4.



Figur 3.4: Resultatet af *shadings* udført fra venstre med hhv. *flat*, *gouraud* og *phong shading*. Figur fra [Perea, 2012].

Flat shading
Flat shading, er en metode som blot udfører *shading* beregningerne en enkelt gang for hver polygon, således at hvert punkt på polygonen tildeles den samme farve. Dette gøres ved at antage at polygonen er flad, således at normalen $\mathbf{n}$ er konstant, hvilket betyder at en given *vertex*, må kunne tildeles forskellige normaler, som passer til hver trekant. Derudover antages også at kamera og lyskilde er placeret så langt væk, at retningerne $\mathbf{w}_o$ og $\mathbf{w}_i$, også kan antages at være konstante. Dette resulterer i et uønsket facetteret og fladt udseende.

Gouraud Shading
Med *Gouraud shading shades* polygonerne således at de ligner en kontinuert overflade. Dette gøres ved at udføre *shading* beregningerne en gang for hvert *vertex*, ved brug af hver *vertex*'s individuelle normal $\mathbf{n}$, retning mod kamera $\mathbf{w}_o$, retning mod lyskilde $\mathbf{w}_i$ og materialeegenskaber. Der kan så interpoleres lineært over de beregnede *shades*, således at hvert punkt indenfor polygonen kan tildeles en *shade*/farve. Dette vil resulterer i et glat udtryk, men man kan risikere at miste information (eks. highlights) som kun er inden i polygonen, eller at highlights vil tydeliggøre polygon-kanterne (se figur 3.4).

Phong shading
Phong shading er den mest anvendte metode, da denne kan producere en glat overflade, samt modellere spekulære highlights. Modsat *Gouraud shading*, interpolerer *phong shading* over attributterne for hvert *vertex*, således at hvert punkt kan udføre *shading* beregningerne, med individuelle (normaliserede) vektorer og materialeegenskaber.

3.3 Procedural modellering og cases

Der vil i dette afsnit først præsenteres de grundlæggende byggesten for opbygningen af procedurale modeller, hvorefter 3 cases vil gennemgås, der henholdsvis proceduralt modellerer udseendet af mursten, tråd og den lag-inddeling som ses på eksempelvis 3D-printede objekter.

3.3.1 *Layering* og *composition*

Procedural modellering er et værktøj ofte anvendt i computer grafikken, hvormed et specifikt udseende kan opbygges algoritmisk *'on-the-fly'*, frem for eksempelvis at anvende et teksturbillede, et *bump-map* eller andre pladskrævende metoder (eller naturligvis håndkraft). Et sådant specifikt udseende kan f.eks. opnås ved proceduralt at vælge en farve, regulere *visibility*'en eller modificere normalen, tangenten eller geometrien.

En ofte anvendt fremgangsmetode til at generere komplekse procedurale mønstre,

er at kombinere simple primitive funktioner. Sådanne funktioner kan kombineres på mange forskellige måder for at skabe komplekse mønstre, men de oftest anvendte teknikker kaldes *layering* og *composition*. Med teknikken *layering* "stables" simple funktioner ovenpå hinanden, hvilket vil sige at outputtet fra en funktion adderes, eller multipliceres, med outputtet fra en anden funktion. Med teknikken *composition* anvendes outputtet fra en eller flere funktioner som input til en anden funktion [Ebert et al, 2003].

I afsnittet nedenfor præsenteres de simple funktioner der i afsnit 3.3.3 kombineres på forskellige måder for at modellere de føromtalte udseender.

3.3.2 Primitive funktioner

Primitive funktioner fungerer som byggestenene i en procedural algoritme, og der findes naturligvis uendelig mange matematiske funktioner der kan anvendes som beskrevet ovenfor. I [Ebert et al, 2003] præsenteres mange sådanne funktioner, men i dette afsnit beskrives blot de vigtigste i fht. modelleringen udført i denne raport.

Almindelige funktioner
Nogle af de mere almindelige funktioner, som ikke behøver nogen videre forklaring, er givet ved:

- abs - absolutværdien af en inputværdi
- floor - det største heltal der er mindre end eller lig med inputværdien
- fract - den fraktionelle del af inputværdien, fract(x) = x - floor(x)

Mix
Funktionen mix(a,b,t), interpolerer lineært over de 2 værdier a og b på følgende måde:

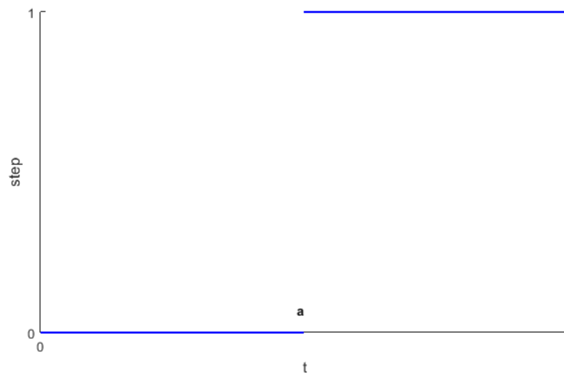
$$\text{mix}(a,b,t) = (1-t)a + tb \tag{3.16}$$

Hvor $t \in [0, 1]$ er interpolationskonstanten og a,b $\in \mathbb{R}^n$, $n > 0$. Denne funktion anvendes i sær til at interpolere lineært over farver eller retningsvektorer.

Step
Funktionen step(a,t) er defineret ved:

$$\text{step}(a,t) = \begin{cases} 0 & \text{for } t<a \\ 1 & \text{for } t \geq a \end{cases} \quad , a,t \in \mathbb{R} \tag{3.17}$$

Denne funktion anvendes især som en erstatning af if-sætninger, hvilket f.eks. kan gøres i mix funktionen når man ønsker strengt at vælge mellem en af de 2 værdier c og d, vha. kaldet : mix(c,d,step(a,t)). En graf af step-funktionen kan ses i figur 3.5.

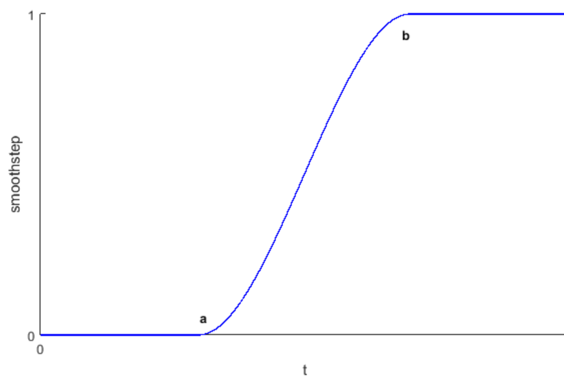


Figur 3.5: Step-funktionen, her vist for step(0.5,t).

Smoothstep
Ofte er det ønskeligt at anvende en graduel overgang fra 0 til 1, modsat den skarpe overgang der anvendes i step. Dette kan gøres vha. smoothstep-funktionen, som er defineret ved:

$$\text{smoothstep}(a,b,t) = \begin{cases} 0 & \text{for } t<a \\ 1 & \text{for } t \geq b \\ 3(\frac{t-a}{b-a})^2 - 2(\frac{t-a}{b-a})^3 & \text{for } a \leq t < b \end{cases} \quad s\ , a,b,t \in \mathbb{R}, b > a \tag{3.18}$$

Den graduelle overgang fra 0 til 1, i intervallet a til b, opnås vha. den kubiske funktion $3x^2 - 2x^3$, som sørger for at hældningen er 0 i a og b, og at funktionen har en værdi på 0 i a og på 1 i b. En graf af smoothstep-funktionen kan ses i figur 3.6.



Figur 3.6: Smoothstep-funktionen, her vist for smoothstep(0.3,0.7,t).

3.3.3 Cases

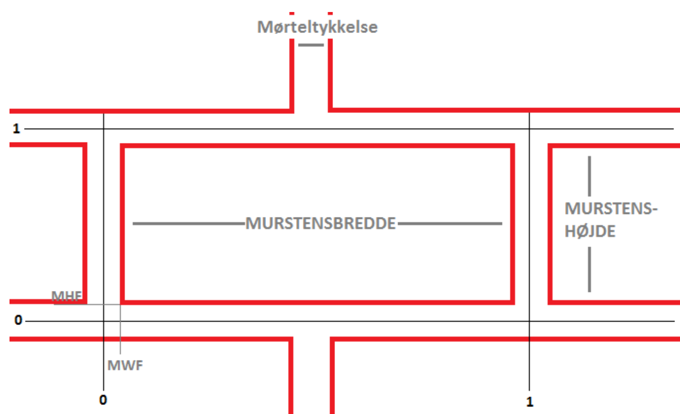
Der vil i dette afsnit gennemgås de 3 føromtalte cases, hvoraf de første 2 er baseret på det efterhånden klassiske murstens-eksempel, som bl.a. er beskrevet i [Ebert et al, 2003], i mens den sidste case bygger på en algoritme som er blevet udviklet i forbindelse med denne rapport.

I alle 3 cases er det som udgangspunkt *world space* fragment positionen, som danner basis for den procedurale modellering, men hvis man ønsker en modellering der er konsistent med geometrien, selvom denne flyttes eller roteres, kan *object space* anvendes i stedet.

3.3.3.1 Mursten

Her gennemgås de nødvendige trin for proceduralt at skabe en murstens-tekstur med *bump-mapping*, som er et eksempel der både illustrerer hvordan man proceduralt kan skabe en 2D/3D tekstur og modificere normalen på objektet for at, i dette tilfælde, opnå et mere realistisk udseende.

Procedural 2D/3D tekstur
Hvis man ser bort fra den støj, der gerne ses i virkelighedens murstensmure, såsom ridser og små uregelmæssigheder, så har murstensmure en meget veldefineret og regulær geometri. Der er blot tale om at relativt ens mursten stables ovenpå hinanden, og at hver mursten adskilles fra de øvrige vha. mørtel. Det murstens-mønster det ønskes at efterligne her, er et hvor hveranden række forskudes med en halv murstens bredde i fht. de rækker de omringer den (se figur 3.7).



Figur 3.7: Geometrien af en murstensmur.

En 2D murstens tekstur kan således genereres proceduralt vha. en algoritme der givet et punkt (x,y) i *world space* kan bestemme hvorvidt dette punkt tilhører mursten eller mørtel, og så farvelægge punktet derefter. Når der arbejdes med sådan en 2D tekstur i *world space* forventes det at overfladen der skal tekstureres kun er udspændt af 2 koordinater, som kan beskrive punktet (x,y).

De nødvendige trin der må udføres, og variablerne der må kendes, for at bestemme farven af et givent punkt (x,y) kan ses opridset med pseudokode i algoritme 1, samt i figur 3.7. I algoritmen ses det, at det første trin er at finde ud af hvilken række og søjle af mursten det givne punkt er i, hvilket gøres ved at dividere punktet med dimensionen af en mursten omringet af mørtel med halv mørteltykkelse. Bemærk at fordi der arbejdes med reelle tal, kan der også forekomme negative og fraktionelle række- og søjlenumre.

Algorithm 1 Pseudokode for farvevalg - if-sætninger erstattes af smartere løsninger

1: Givet: $(x, y) \in \mathbb{R}$, murstensbredde m_b og højde m_h, mørteltykkelse m, murstens- og mørtelfarve ($\mathbf{C}_{mn}$ og $\mathbf{C}_{ml}$), MWF og MHF
2:
3: Find murstensnummer i vertikal og horisontal retning:
4: $n_h = x/(m_b + m)$
5: $n_v = y/(m_h + m)$
6:
7: **if** (n_v er ulige)
8: $\quad n_h = n_h + 0.5$ ▷ forskyd række med en halv murstensbredde
9: **end if**
10:
11: Udregn position i fht. den givne mursten:
12: $pos_x = n_h$ - floor(n_h)
13: $pos_y = n_v$ - floor(n_v)
14:
15: Undersøg om denne position tilhører mørtel eller mursten:
16: **if**($pos_x < MWF$ **or** $pos_x > (1 - MWF)$ **or** $pos_y < MHF$ **or** $pos_y > (1 - MHF)$)
17: $\quad$ **return** $\mathbf{C}_{ml}$
18: **else**
19: $\quad$ **return** $\mathbf{C}_{mn}$
20: **end if**

Denne information anvendes så til at bestemme hvorvidt den givne række n_v skal forskydes med en halv murstensbredde. I stedet for at anvende en if-sætning, kan dette bestemmes vha. fract og step funktionen på følgende måde:

$$\text{step}(0.5, \text{fract}(0.5\, n_v)) \tag{3.19}$$

Dette udtryk vil returnere 1, når fract($0.5 \cdot n_v$) er lig med eller større end 0.5, hvilket vil være tilfældet når heltalsdelen af n_v er et ulige tal. Rækken n_v kan derfor blot forskydes en halv murstens bredde, ved at lægge funktion 3.19 til n_h ganget med 0.5.

Næste trin er at finde ud af hvor punktet er placeret i fht. den givne mursten (omringet af mørtel med halv mørteltykkelse), hvor det af figur 3.7 ses at denne position vil være givet ved (pos_x, pos_y) $\in$ [0,1]. Positionen findes ved hjælp af fract funktionen, og det bemærkes at dette også fungerer for negative tal, hvor f.eks. et punkt i række -3.2 vil have positionen pos_y = -3.2 - (-4) = 0.8, på grund af definitionen på fract.

Endelig kan farven af det givne punkt bestemmes, ved at undersøge hvorvidt denne position tilhører mørtel eller mursten, baseret på geometrien fra figur 3.7. Dette kan i stedet for den lange if-sætning som ses i psedokode algoritmen, gøres ved at udregne 2 værdier, en for x-retningen og en for y-retningen, som vil være 0 når punktet er i mørtel og 1 når det er inde i murstenen, i den givne retning. Ved at multiplicere disse 2 værdier opnås så den endelig værdi t, som bestemmer farven af punktet. De 2 udtryk der giver den ønskede effekt er givet ved:

$$\begin{aligned} &\text{step}(MWF, pos_x) - \text{step}(1 - MWF, pos_x) \\ &\text{step}(MHF, pos_y) - \text{step}(1 - MHF, pos_y) \end{aligned} \tag{3.20}$$

Hvoraf betydningen af MWF og MHF kan aflæses i figur 3.7, og f.eks. MWF udregnes ved 0.5 $m/(m_b{+}m)$. Farven vælges så vha. mix($\mathbf{C}_{ml}, \mathbf{C}_{mn}, t$), og første del af den procedurale 2D tekstur er nu opbygget.

Algoritmen kan nu udvides til anvendelse i 3D, således at der kigges på en murstensterning frem for blot en murstensflade. Dette gøres ved at tilføje en z koordinat, og undersøge hvilken farve et punkt i området (x,y,z) $\in \mathbb{R}$ skal have. På denne måde opnår vi en 3D tekstur, hvor det også er muligt at anvende andre objekter end blot en terning, da terningen så og sige også er farvelagt indvendigt, og det er derfor muligt at 'skære' et andet objekt ud af terningen. Dette kaldes også en *solid 3D texture*.

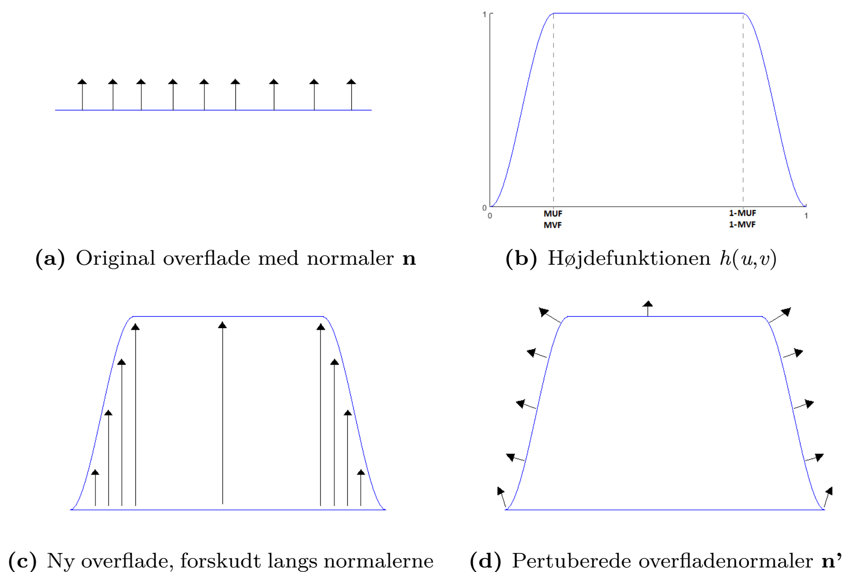
Ovenstående algoritme udvides til 3D ved at tilføje en murstenslængde m_l i z-retningen, og en variabel MLF = 0.5 $m/(m_l{+}m)$. Det kan også her vælges at forskyde hverandens række i z-retningen, på en lignende måde, som forklaret tidligere. Nedenstående farvetjek for z-retningen tilføjes, og der opnås dermed en murstens 3D tekstur, hvor murstensrækkerne er stablet bag ved hinanden.

$$\text{step}(MLF, pos_z) - \text{step}(1 - MLF, pos_z) \tag{3.21}$$

Procedural bump-mapping

Procedural *bump-mapping* vil nu tilføjes til algoritmen, for at give et indtryk af at murstenene stikker frem i fht. mørtlen. *Bump-mapping* er en metode der simulerer

bump eller indrykninger i en overflade, ved at modificere overfladenormalen, men uden at ændre overfladen. *Bump-mapping* blev introduceret i [Blinn, 1978], hvori den konceptuelle idé er, at man kan simulere et bump med højde $h(u,v)$ ved at forskyde et punkt med $h(u,v)$ langs overfladenormalen $\mathbf{n}$, og så udføre lysberegningerne med dette forskudte punkts normal, også kaldet den perturberede overfladenormal $\mathbf{n'}$. Denne idé kan ses illustreret i figur 3.8, hvor den viste højdefunktion $h(u,v)$ er den der ønskes at blive anvendt. Altså ønskes det at selve murstenen forskydes 1.0 langs overfladenormalerne, i mens mørtlen alt efter dens placering forskydes $h(u,v) \in [0,1]$.



(a) Original overflade med normaler $\mathbf{n}$

(b) Højdefunktionen $h(u,v)$

(c) Ny overflade, forskudt langs normalerne

(d) Pertuberede overfladenormaler $\mathbf{n'}$

Figur 3.8: De forskellige trin i en *bump-mapping* proces.

Denne højdefunktion $h(u,v)$ kan implementeres på følgende måde, hvor højden i fht. u-retningen multipliceres med højden i fht. v-retningen givet et punkt (u,v):

$$
\begin{aligned}
h_u &= \text{smoothstep}(0, MUF, u) - \text{smoothstep}(1 - MUF, 1, u) \\
h_v &= \text{smoothstep}(0, MVF, v) - \text{smoothstep}(1 - MVF, 1, v) \\
h(u,v) &= h_u\, h_v
\end{aligned}
\tag{3.22}
$$

Hvor teksturkoordinaterne (u,v) findes, givet 3D-punktet $\mathbf{x} = (pos_x,\ pos_y,\ pos_z) \in [0,1]$, der beskriver positionen i fht. den givne mursten, ved at undersøge parametriseringen af en flade:

$$
\begin{aligned}
&\mathbf{x} = \mathbf{x}_0 + u\mathbf{t} + v\mathbf{b} \Rightarrow \\
&\mathbf{t} \cdot \mathbf{x} = \mathbf{t} \cdot \mathbf{x}_0 + \mathbf{t} \cdot (u\mathbf{t}) + \mathbf{t} \cdot (v\mathbf{b}) \Rightarrow \\
&u = (\mathbf{x} - \mathbf{x}_0) \cdot \mathbf{t}
\end{aligned} \tag{3.23}
$$

På samme måde kan det udledes at $v = (\mathbf{x} - \mathbf{x}_0) \cdot \mathbf{b}$, hvor $\mathbf{t}$ er tangenten og $\mathbf{b}$ er binormalen, som er angivet for hver flade af kuben. Således vil højdefunktionen for f.eks. et punkt $\mathbf{x} = (pos_x, pos_y, pos_z)$, på den forreste flade af kuben, hvor $\mathbf{t} = (1{,}0{,}0)$ og $\mathbf{b} = (0{,}1{,}0)$, afhænge af $u = pos_x$ og $v = pos_y$. De dertilhørende værdier *MUF* og *MVF* bestemmes på en lignende måde til at være *MWF*, *MHF* eller *MLF*.

Med højdefunktionen på plads kan forskydningen nu udføres og den perturberede normal udregnes, som forklaret i [Bærentzen et al]. Givet et punkt $\mathbf{x}$ og dennes normal $\mathbf{n}$ (enhedsnormal), vil en forskydning langs normalen resultere i følgende position:

$$
\mathbf{x'} = \mathbf{x} + h(u,v)\mathbf{n} \tag{3.24}
$$

Hvor normalen $\mathbf{n'}$ for denne nye forskudte position vil være givet ved krydsproduktet af dennes tangent og binormal (enhedsvektorer, der svarer til de partielt afledede i fht. u og v):

$$
\mathbf{n'} = \mathbf{t'} \times \mathbf{b'} = \frac{\partial \mathbf{x'}}{\partial u} \times \frac{\partial \mathbf{x'}}{\partial v} = \frac{\partial}{\partial u}(\mathbf{x} + h(u,v)\mathbf{n}) \times \frac{\partial}{\delta v}(\mathbf{x} + h(u,v)\mathbf{n}) \tag{3.25}
$$

Ved at anvende produktreglen (og sumreglen), kan de partielt afledte begge omskrives på følgende måde:

$$
\frac{\partial}{\partial u}(\mathbf{x} + h(u,v)\mathbf{n}) = \frac{\partial \mathbf{x}}{\partial u} + \frac{\partial h}{\partial u}\mathbf{n} + h(u,v)\frac{\partial \mathbf{n}}{\partial u} \tag{3.26}
$$

Vha. disse udtryk kan udtrykket for den perturberede normal simplificeres, ved at antage at højdefunktionen $h(u{,}v)$ er så lille at der kan ses bort fra det sidste led i ligning 3.26. Den perturberede normal kan så beskrives ved følgende:

$$
\begin{aligned}
\mathbf{n'} &\approx (\frac{\partial \mathbf{x}}{\partial u} + \frac{\partial h}{\partial u}\mathbf{n}) \times (\frac{\partial \mathbf{x}}{\partial v} + \frac{\partial h}{\partial v}\mathbf{n}) \\
&= \frac{\partial \mathbf{x}}{\partial u} \times \frac{\partial \mathbf{x}}{\partial v} + \frac{\partial h}{\partial v}(\frac{\partial \mathbf{x}}{\partial u} \times \mathbf{n}) + \frac{\partial h}{\partial u}(n \times \frac{\partial \mathbf{x}}{\partial v}) \\
&= \mathbf{n} - \frac{\partial h}{\partial v}\frac{\partial \mathbf{x}}{\partial v} - \frac{\partial h}{\partial u}\frac{\partial \mathbf{x}}{\partial u}
\end{aligned} \tag{3.27}
$$

Med dette udtryk er det nu muligt at udregne den perturberede normal, hvis normalen $\mathbf{n}$, tangenten $\delta\mathbf{x}/\delta u$ og binormalen $\delta\mathbf{x}/\delta v$ for det oprindelige punkt er tilgængelig.

Tilbage er blot først at udregne de partielt udledede af h. Dette kan eksempelvis gøres vha. *central differences*:

$$\frac{\partial h}{\partial u} = \frac{h(u + \Delta, v) - h(u - \Delta, v)}{2\Delta} \tag{3.28}$$

Hvoraf Δ er defineret således at man ved at lægge denne til teksturkoordinaten undersøger højden for en nærliggende teksturkoordinat, hvor denne gerne skal kunne opfange en højdeforskel. Et godt bud kunne derfor være at anvende $\Delta = 1/\text{H}$, hvor H er antallet af pixels i højden. Endelig kan den perturberede normal udregnes, og lysberegningerne kan udføres i det ønskede rum. Da den perturberede normal er i *tangent space*, vælges det oftest at udføre lysberegningerne i dette rum. En given vektor $(x_w,\ y_w,\ z_w)$ i *world space*, som skal anvendes i lysberegningerne, kan transformeres til *tangent space* på følgende måde [McReynolds, 1997]:

$$\begin{bmatrix} x_t \\ y_t \\ z_t \end{bmatrix} = \begin{bmatrix} t_x & t_y & t_z \\ b_x & b_y & b_z \\ n_x & n_y & n_z \end{bmatrix} \begin{bmatrix} x_w \\ y_w \\ z_w \end{bmatrix} \tag{3.29}$$

Hvor tangent, binormal og normal er defineret i *world space*.

3.3.3.2 Tråd

I dette afsnit gennemgås trinene der må udføres for proceduralt at modellere en trådtekstur, som er en tekstur der bygger på de samme principper, som ovenstående murstens-tekstur. Da lysmodellen for hår/tråd beskrevet i afsnit 3.2.2.1, anvender tangenten, vil det også gennemgås hvordan denne kan vælges proceduralt. Hensigten med denne tekstur, er at modellere en stofkugle, så mappingen til en sfære vil også kort beskrives.

Sfære-mapping
Da det ønskes at modellere en stofkugle som er betrukket af stof, og altså ikke har stof indeni, må der anvendes en 2D tekstur. Givet et punkt på kuglen, må dette derfor mappes til teksturkoordinater $(u,\ v)$, hvilket kan gøres ved at anvende enhedskuglen som en såkaldt *intermediate surface* [Angel, 2014]. Den normaliserede normal $\mathbf{n}$, knyttet til punktet på kuglen, beskrevet i *world space* koordinater, kan repræsentere det korresponderende punkt på enhedskuglen, og en mulig mapping kan derfor findes ved at omskrive til sfæriske koordinater, og skalere således at $(u,\ v) \in [0{,}1]$, på følgende måde:

$$u = \frac{\tan^{-1}(n.x/n.z)}{2\pi}\ ,\ v = \frac{\cos^{-1}(n.y)}{\pi} \tag{3.30}$$

Hvor u repræsenterer breddegraderne fra $[0,\ 2\pi]$, og v repræsenterer længdegraderne

fra $[0, \pi]$. Givet teksturkoordinaterne (u, v), kan en 2D tråd tekstur genereres proceduralt, vha. den samme algoritme som beskrevet i ovenstående case. Der er dog enkelte modifikationer, så algoritmen gennemgås kort i nedenstående delafsnit:

Procedural 2D tekstur

Som det ses af figur 3.9, har trådteksturen stort set samme geometri, som murstensgeometrien. Dog anvendes der kvadratiske lufthuller, fremfor rektangulære, og der udføres ikke en forskydning på hver anden række. Følgende variable skal bruges: teksturkoordinaterne (u, v), hulbredden h, trådtykkelsen t, MWF og MHF (betydning beskrevet i afsnit 3.3.3.1), samt farverne der beskriver tråden $\mathbf{C}_t$ og lufthullet $\mathbf{C}_h$. Bemærk at farven for lufthullet bør have en w komponent der er lig 0, da dette svarer til alpha-værdien, således at farven i lufthullet bliver gennemsigtig. Hvilken farve der skal anvendes for det givne punkt (u, v) bestemmes så ved først at udregne:

$$pos_x = \text{fract}(u/(t+h)) \ , \ pos_y = \text{fract}(v/(t+h)) \tag{3.31}$$

$$w = \text{step}(MWF, pos_x) - \text{step}(1 - MWF, pos_x) \tag{3.32}$$

$$h = \text{step}(MHF, pos_y) - \text{step}(1 - MHF, pos_y) \tag{3.33}$$

Farven for det givne punkt kan så bestemmes ved mix($\mathbf{C}_t$, $\mathbf{C}_h$, $w \cdot h$).

Proceduralt valg af tangent

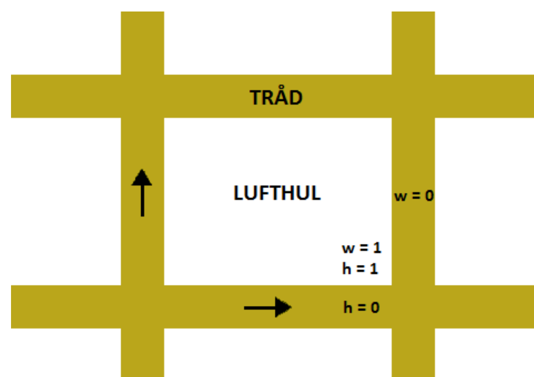
Kajiya og Kaj's lysmodel der beskriver hår/tråd anvender, som sagt, den tangent som følger trådretningen, og denne må derfor bestemmes for et givent punkt. Som det ses af figur 3.9, er der tale om en tangent der følger den vertikale trådretning når $w = 0$, og en der følger den horisontale retning når $h = 0$. Da trådteksturen er mappet til en sfære, må der dog i stedet være tale om at tangenterne følger længde- og breddegraderne (altså v og u-retningen).

Det ønskes at beregne disse tangenter på en måde, så disse er konsistente med sfæremappingen. En tangent der følger u-retningen, må derfor udregnes ved at projektere normalen $\mathbf{n}$ ned på (x,z)-planen. Hvis det antages at u-retningen svarer til højre rundt om kuglen (som i figur 3.9), kan en sådan tangent findes vha. følgende krydsprodukt:

$$\mathbf{t}_u = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \times \begin{bmatrix} n.x \\ n.y \\ n.z \end{bmatrix} \tag{3.34}$$

Hvor der vil være en singularitet når normalen er givet ved (0,1,0) eller (0,-1,0), altså akkurat på hhv. toppen og bunden af kuglen. Tangenten som følger v-retningen, kan så nemt bestemmes ved nedenstående udtryk, hvor det antages at trådretningen her er opadgående:

$$\mathbf{t}_v = \mathbf{n} \times \mathbf{t}_u \tag{3.35}$$



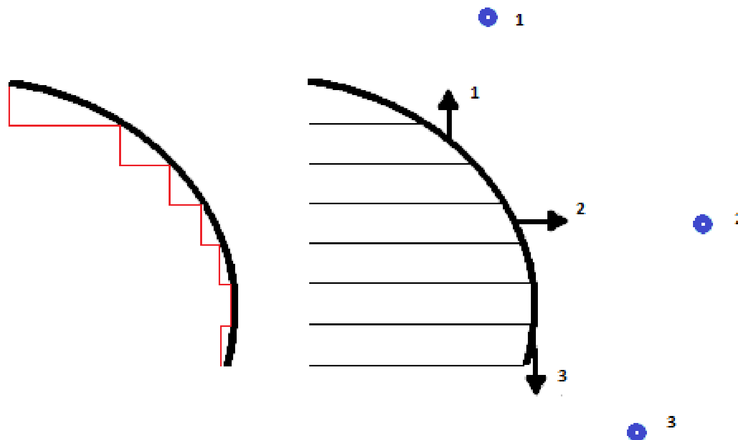
Figur 3.9: Geometrien af tråd. Pilene indikerer den ønskede tangentretning, og værdierne w og h, beskriver resultatet af ligning 3.32 og 3.33 for de viste positioner.

Hvilken tangent der skal anvendes bestemmes så blot ved at undersøge værdierne w og h, som er blevet bestemt proceduralt. Bemærk at når $w = h = 0$, befinder punktet sig der hvor 2 tråde krydser hinanden. Den nemme løsning vil her være blot at altid anvende den samme tangent (enten $\mathbf{t}_u$ eller $\mathbf{t}_v$), men et mere realistisk udtryk ville nok opnås hvis man skiftevis anvendte den ene eller den anden, for at give indtrykket af at trådene flettes ind i hinanden. Når $w = h = 1$, befinder punktet sig i et lufthul, og det antages at tangenten her kan sættes til normalen $\mathbf{n}$. Denne antagelse bygger på at stof i virkeligheden, har meget små lufthuller, samt en udbredelse i alle 3 dimensioner - tangenten bør derfor følge denne trådtykkelse langs den 3. dimension.

3.3.3.3 Lag-inddeling

I dette afsnit gennemgås tankerne bag, samt de nødvendige trin der må udføres, for at proceduralt modellere det lag-inddelte udseende der eksempelvis ses på 3D-printede objekter, vha. såkaldt *normal mapping*.

Normal mapping
Konceptet bag algoritmen, der proceduralt kan modellere et lag-inddelt udseende, eller en såkaldt trappeeffekt, kan ses illustreret i 2D i figur 3.10. Hvis det antages at y-aksen beskriver print-retningen, består algoritmen i at proceduralt modificere y-koordinaten af normalen $\mathbf{n}$, baseret på placeringen af kameraet i fht. punktets placering mellem 2 lag. Hvor det altså af figur 3.10 ses, at der må opstilles en form for mål for om strålen fra kameraet i gennem et givent punkt $\mathbf{p}$ på objektet, vil ramme hvad der svarer til 'toppen', 'bunden' eller 'kanten' af et trin, som henholdsvis svarer til scenarie 1, 3 og 2 i figuren.



Figur 3.10: Til venstre: Den sorte kurve repræsenterer objektet det ønskes at *normal mappe*, for at give det lag-inddelte udseende som er repræsenteret af de røde streger.
Til højre: 3 scenarier, der demonstrerer hvad det ønskes at normalen for et givent punkt skal ændres til, vha. *normal mappingen*. Nummereringen beskriver hvilke kamerapositioner (blå cirkler) og *mappede* normaler der er relateret til hinanden.

Første punkt i algoritmen er derfor at bestemme hvilket lag, der er det første strålen fra kameraet vil ramme, efter at den har ramt punktet $\mathbf{p}$, hvor et lag er repræsenteret ved en plan der står vinkelret på y-aksen. Givet et punkt $\mathbf{p}$ i *world space* og en lagtykkelse l, kan indekset i for laget under punktet $\mathbf{p}$, bestemmes ved:

$$i = \text{floor}(p_y/l)\ ,\ i \in \mathbb{Z} \tag{3.36}$$

Da kameraet kun vil pege mod laget under punktet **p**, hvis enheds retningsvektoren **w**, som beskriver retningen fra kameraets position mod **p**, har en negativ y-koordinat, tilføjes følgende til ligning 3.36:

$$i = \text{floor}(p_y/l) + \text{step}(0, w_y)\ ,\ i \in \mathbb{Z} \tag{3.37}$$

Givet dette indeks i, som beskriver det første lag strålen i retningen **w** vil ramme (hvis stråle og lag ikke er parallelle), ønskes det nu at bestemme afstanden t. Denne svarer til afstanden fra punktet **p** til den eventuelle skæringen med laget med indeks i, i fht. retningen **w**. Denne afstand t, vil så anvendes til at modificere normalen **n**'s y-koordinat, og svarer altså dermed til det føromtalte mål for hvilken del af et trin, strålen fra kameraet vil ramme. Afstanden t, kan findes ved at anvende *ray-plane intersection*, som også er mere dybdegående forklaret i afsnit 3.5.2. Ved at skyde en stråle med retning **w**, fra punktet **p** mod laget, kan denne afstand t bestemmes ved [Hughes et al, 2014]:

$$t = \frac{(\mathbf{q} - \mathbf{p}) \cdot \mathbf{n}_p}{\mathbf{w} \cdot \mathbf{n}_p} \tag{3.38}$$

Hvoraf vektorerne **q** og $\mathbf{n}_p$, henholdsvis er et punkt og en normal som kan repræsentere den plan, som svarer til laget. En sådan normal må givetvis være $\mathbf{n}_p$ = (0,1,0) og et punkt på planen findes også nemt da lagets indeks er kendt: **q** = (0, $i \cdot l$, 0). Ligning 3.38, kan derfor simplificeres til følgende:

$$t = \frac{i \cdot l - p_y}{w_y} \tag{3.39}$$

Givet denne afstand t, kan *normal mappingen* endelig udføres, hvor det, som sagt, kun ønskes at modificere normalens y-koordinat. Hvis afstanden t er mindre end lagtykkelsen l, antages det at kameraet kigger mod 'toppen'/'bunden' af et trin, og y-koordinaten sættes derfor til: $n_y = \pm 1$. Hvis afstanden t derimod er større end lagtykkelsen l, må der kigges mod 'kanten' af et trin, og derfor sættes $n_y = 0$. Denne nye modificerede normal $\mathbf{n}^*$ kan beregnes proceduralt vha. nedenstående udtryk:

$$\mathbf{n}^* = (n_x,\ \text{step}(0, l - t) \cdot \frac{n_y}{\text{abs}(n_y)},\ n_z) \tag{3.40}$$

Hvilket dermed slutter algoritmen, bestående af ligning 3.37, 3.39 og 3.40. Denne kan naturligvis modificeres til at have en anden akse som print-retning, eller til f.eks. at have 2 på en gang, for at give udtrykket af at objektet er opdelt i lag på 2 akser (altså give det et '*minecraft*'/*voxel*-agtigt udtryk).

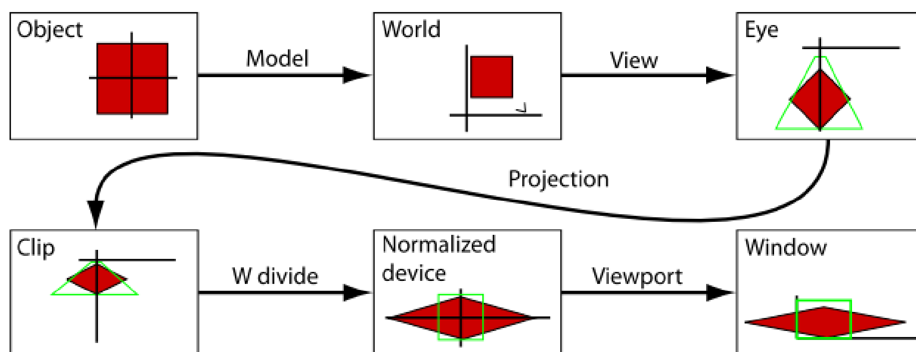
3.4 Realtids grafik-pipeline

Pipelinen i et grafikkort kan groft sagt opdeles i en geometri-, en rasteriserings- og en fragmentdel, hvor hver af disse trin i sig selv også kan udgøre en pipeline. I de nedenstående afsnit vil detaljerne for hver del af pipelinen gennemgås, som gjort i [Bærentzen et al], og problemerne ved at arbejde med procedurale modeller i sådan en pipeline vil belyses.

Før det første trin, må geometrien defineres og den virtuelle scene opstilles, hvilket gøres ved at placere kamera og lyskilder, samt ved at definere geometrien vha. primitiver (polygoner, linjer eller punkter defineret vha. *vertices* - ofte trekanter, da der arbejdes med objekter opbygget af trekantsmesh), beskrevet ved homogene koordinater. Alt dette fungerer som input til det første trin i pipelinen.

3.4.1 Trin 1: Geometri

Geometrien som inputtes til dette trin er defineret i fht. dets eget 3D koordinatsystem, og det må derfor transformeres til det koordinatsystem som repræsenterer det 2D display det ønskes at outputte det renderede resultat på. Geometrien må gå i gennem en lang række af transformationer, og befinder sig dermed i en masse forskellige koordinatsystemer, før dette kan lade sig gøre. De forskellige koordinatsystemer og transformationerne i mellem dem, kan ses illustreret i figur 3.11.



Figur 3.11: Figuren illustrerer de forskellige transformationer (ord over pile) som vil anvendes på geometrien, samt de koordinatsystemer (ord i bokse) de vil være i, på deres vej igennem denne geometridel af pipelinen. Den røde boks illustrerer geometrien der renderes, i mens den grønne boks indikerer kameraets view frustum. Figur fra [Bærentzen et al].

Når man arbejder med realtids rendering, er det vigtigt at have en god forståelse af disse koordinatsystemer, da man altid skal vide hvilket rum man er i, således at man kan sørge for at de elementer man skal anvende i en beregning, også befinder sig i dette rum. I de efterfølgende underafsnit vil de 5 transformationer illustreret i figur 3.11, derfor gennemgås.

Model transformation

World space er det koordinatsystem der repræsenterer den virtuelle scene, og oftest vælges det at anvende et separat koordinatsystem, et såkaldt *object space*, for hvert objekt i scenen. Hvis der arbejdes med et *object space*, hvor ens 3D objekt f.eks. er defineret relativt til objektets centrum, må dette transformeres for at placere og orientere det i den virtuelle scene. Et givent *vertex* kan i princippet transformeres fra *object space* til *world space*, vha. en hver form for 4x4 matrix, men oftest anvendes blot en kombination af eventuelle translationer, rotationer og skaleringer. Således kunne en modelmatrix **M**, opstilles som den der ses i ligning 3.41, som består af en multiplicering af en skalerings-, en rotations- og en translationsmatrix. 3x3 matricen **A** beskriver rotation og skalering, i mens vektoren **t** beskriver translation.

$$\mathbf{M} = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & t_1 \\ a_{2,1} & a_{2,2} & a_{2,3} & t_2 \\ a_{3,1} & a_{3,2} & a_{3,3} & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.41}$$

View transformation

View transformationen består af en translation efterfulgt af en rotation. Translationen sørger for at det virtuelle kamera placeres i origo, i mens rotationen sørger for at kameraet kigger langs den negative z-retning. Et kamera beskrives oftest ved de 3 vektorer: kamerapositionen **e**, midten af scenen som kameraet kigger i mod **l** (også kaldet 'lookat' punktet) og en op-retning **o**. Translationen af kameraet til origo må derfor blot være givet ved **-e**, i mens en orthornormal basis **u**, **v**, **n** for *eye space* må opstilles for at kunne udføre rotationen. En sådan en kan findes ved:

$$\begin{aligned} \mathbf{v} &= -\frac{\mathbf{l}-\mathbf{e}}{||\mathbf{l}-\mathbf{e}||} \\ \mathbf{n} &= \frac{(\mathbf{l}\text{-}\mathbf{e}) \times \mathbf{o}}{||(\mathbf{l}\text{-}\mathbf{e}) \times \mathbf{o}||} \\ \mathbf{u} &= \mathbf{n} \times \mathbf{v} \end{aligned} \tag{3.42}$$

Den første basisvektor findes nemt, da det vides at kameraet skal kigge langs den negative z-retning, i mens den anden, som svarer til skærmens x-retning, findes som krydsproduktet mellem kameraets specificerede syns- og op-retning. *View* transformationen er så givet ved:

$$\mathbf{V} = \begin{bmatrix} n_x & n_y & n_z & -\mathbf{n}\cdot\mathbf{e} \\ u_x & u_y & u_z & -\mathbf{u}\cdot\mathbf{e} \\ v_x & v_y & v_z & -\mathbf{v}\cdot\mathbf{e} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.43}$$

Projection transformation

Målet med *projection* transformationen er at definere et *viewing volume*, som bestemmer hvordan et givent objekt projekteres op på skærmen (altså, ved perspektiv eller ortografisk projektion). Dette *viewing volume* opnås ved at transformere det givne *view frustum* (se figur 3.11), og dennes indhold, ind i en enhedskube (grunden til at det ikke *ligner* en enhedskube i figuren, er fordi w-komponenten ikke er 1). Den matrix, som transformerer *view frustum* til en enhedskube, kan eksempelvis findes ved hjælp af aspektratioen A, FOV vinklen α (i y-retningen) samt de 2 klippeplaner n og f :

$$\mathbf{P} = \begin{bmatrix} \frac{1}{A}\cot\frac{\alpha}{2} & 0 & 0 & 0 \\ 0 & \cot\frac{\alpha}{2} & 0 & 0 \\ 0 & 0 & \frac{n+f}{n-f} & \frac{2nf}{n-f} \\ 0 & 0 & -1 & 0 \end{bmatrix} \tag{3.44}$$

Ikke overraskende, er det i dette *clip space* at der udføres *clipping*, således at det kun er de primitiver som er inde i det såkaldte *view volumen* der sendes videre til det næste trin. Dette betyder at de primitiver som er ude af *view volumen* fjernes fuldstændig, i mens primitiver som er delvist inde, klippes så kun de dele der er udenfor fjernes. I sidstnævnte tilfælde, erstattes de klippede *vertices*, af nye som automatisk placeres på skæringspunktet mellem linjen (udspændt af 2 *vertices* der beskriver primitivet) og *view volumen* [Akenine-Möller, 2008]. Dette medfører at en klippet trekant kan blive erstattet af flere trekanter. En *vertex* (x_c, y_c, z_c, w_c) i *clip space* er inden for *view volumen* når følgende er opfyldt:

$$-w_c \leq x_c \leq w_c \tag{3.45}$$

$$-w_c \leq y_c \leq w_c \tag{3.46}$$

$$-w_c \leq z_c \leq w_c \tag{3.47}$$

Perspective divide

Clip space koordinaterne transformeres fra de homogene koordinater (x_c, y_c, z_c, w_c) til *ndc space* koordinater (*normalized device coordinates*) (x_{ndc}, y_{ndc}, z_{ndc}, 1), ved at dividere med w_c. Projektionsmatricen er opstillet således, at efter transformationen fra *eye* til *clip space*, vil w_c-koordinatens størrelse afhænge af hvor langt væk objektet

er placeret i fht. kameraet (altså af *eye space* z-koordinaten - se ligning 3.44). Derfor vil en division med w_c betyde at jo længere væk et objekt er jo mere vil det blive trukket mod centrum af skærmen, hvilket giver indtrykket af perspektiv. I *ndc space*, ses således kun den synlige geometri inde i en enhedskube, hvor der gælder at x_{ndc}, y_{ndc}, $z_{ndc} \in$ [-1,1].

Viewport transformation

I det sidste trin i geometridelen af pipelinen, transformeres punkterne fra *ndc space* koordinater til *window space* koordinater, som er et koordinatsystem målt i pixels. Givet en skærm med bredden b og *højden* h pixels, samt en *near-plane* n og *far-plane* f, så vil *viewport transformationen* blot bestå af en skalering og en translation. Et *window space* (x_w, y_w, z_w) koordinat findes således ved [Ahn, 2013], hvor der gælder at $x_w \in [0, b]$, $y_w \in [0, h]$ og $z_w \in [n, f]$:

$$\begin{bmatrix} x_w \\ y_w \\ z_w \end{bmatrix} = \begin{bmatrix} b\frac{x_{ndc}+1}{2} \\ h\frac{y_{ndc}+1}{2} \\ \frac{(f-n)z_{ndc}+f+n}{2} \end{bmatrix} \tag{3.48}$$

Udtrykket for z-koordinaten kan simplificeres en del da *near* og *far-plane* i *window space* er henholdsvis 0 og 1, som svarer til det interval Z-bufferen arbejder med. (x_w, y_w), for sig selv kaldes nogen gange også for *screen space* koordinater, og disse svarer til et punkt i pixelkoordinater på skærmen, i mens z_w beskriver dybden i *window space*.

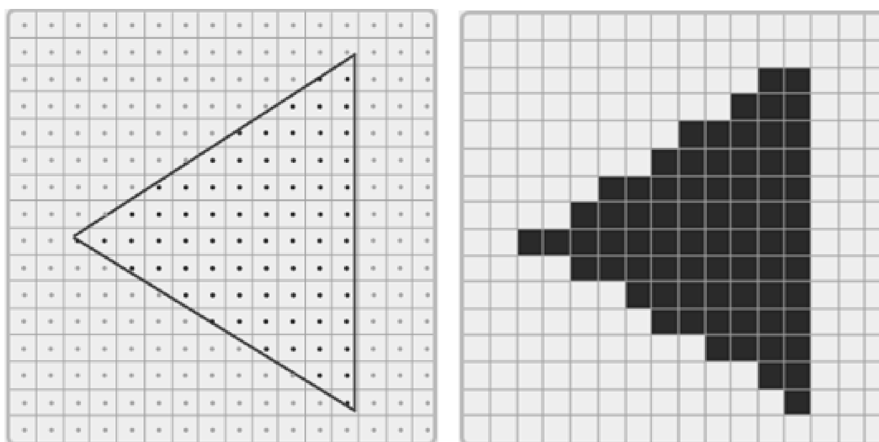
Når den virtuelle scene er beskrevet i *window space*, er den klar til at blive rasteriseret i næste pipeline trin. Hvis eksempelvis man har transformeret 3 punkter til *window space*, kan disse samles til en trekant og rasteriseres.

3.4.2 Trin 2: Rasterisering

Efter geometridelen modtager dette trin primitiverne (polygoner, linjer eller punkter), samt deres tilknyttede *vertices* (position i *window space*, farver og eventuelle andre attributter). Primitiverne må i dette trin konverteres til pixels i framebufferen, hvilket gøres ved at bestemme hvorvidt en pixel er indenfor et givent 2D primitiv. Dette gøres ved at danne et sæt af såkaldte fragmenter for hvert primitiv. Fragmenter kan ses som potentielle pixels der indeholder information om f.eks. pixelpositionen i framebufferen (i *screen space*), pixelfarve, teksturkoordinater og dybde. Et sådant fragment anvendes i næste pipeline step til at opdatere den korresponderende pixel i framebufferen [Angel, 2014].

Dette vil altså sige at rasteriseringsprocessen består i at opdele hvert primitiv i fragmenter af pixelstørrelse, for hver pixel som ligger inden for primitivet. Hvis attributter, så som farve, normaler eller teksturkoordinater, er knyttet til de forskellige *vertices*, så er det også i dette trin at der interpoleres over disse, således at disse interpolerede attributter kan knyttes til hvert fragment.

Konventionen er at en pixel ligger inden for primitivet hvis dennes centrum gør, hvor det i *screen space* koordinater gælder at centrum er placeret i heltals positionen + (0.5, 0.5). Der eksisterer mange forskellige algoritmer, der kan vurdere hvorvidt en pixel er indenfor et 2D primitiv, hvor nogle er beregnet til at approksimere linjer med fragmenter og andre til at fylde polygoner med fragmenter, men det bør bemærkes at grafikkort anvender optimerede metoder, som også er hardwarebestemte. Det er ikke hensigten med dette afsnit at gennemgå sådanne algoritmer, men nærmere at demonstrere de problemer der givetvis må opstå når man transformerer en kontinuert scenebeskrivelse til et diskret sæt af pixels, vha. sampling (et enkelt i midten), uanset hvilken algoritme der anvendes. Dette problem kan ses illustreret i figur 3.12, hvor der ses såkaldte savtakkede overgange, som er karakteristiske ved brugen af raster displays.



Figur 3.12: Til venstre: Grid af pixels, hvor hvert pixelcentrum bestemmer hvorvidt pixlen er indenfor trekanten.
Til højre: Den rasteriserede trekant .

3.4.3 Trin 3: Fragment

I dette sidste trin, processeres fragmenterne fra de rasteriserede polygoner, hvorefter de anvendes til at opdatere pixels i framebufferen. Denne processering kan eksempelvis bestå i *blending* eller *depth testing*, hvoraf sidstnævnte bestemmer hvorvidt et fragment vil være synligt, hvis 2 eller flere fragmenter overlapper den samme pixel. Det afgøres hvilket fragment der er forrest ved at kigge på dybden af fragmentet, som er gemt i z-koordinaten i *window space*. Fragmentfarven kan også i dette trin modificeres, hvis det ønskes at farven ikke blot skal være den interpolerede farve fundet i

ovenstående trin, da denne del af pipelinen er programmerbar. Dette kan eksempelvis gøres vha. procedural modellering, hvor en farve måske bestemmes ud fra den korresponderende *world space* koordinat. Hvis farven af et fragment ændres proceduralt på denne måde, vil *aliasing* forekomme på samme måde i mellem farveovergange, som det sås for trekanten i figur 3.12, hvis det procedurale mønster ikke følger skærmens *pixel-grid*.

Ud over denne såkaldte *aliasing* artifakt kan bl.a. moire mønstre også være en uønsket artifakt, som er en optisk effekt der opstår når 2 eller flere *grids* krydser hinanden, og laver visuel interferens. Dette kan derfor især være at problem når der arbejdes med fine, regulære strukturer, da interferens så kan forekomme grundet interaktion med skærmens *pixel-grid*.

Pipelinen vil altså resultere i forskellige *aliasing*-artifakter, som mestendels skyldes undersampling, og *anti-aliasing* vil derfor beskrives i nedenstående afsnit.

3.5 Anti-aliasing

Når der arbejdes med procedurale modeller, vil *aliasing* være et problem, som netop forklaret i ovenstående afsnit, og en form for *anti-aliasing* må derfor udføres. Der eksisterer mange forskellige *anti-aliasing* metoder, som ofte gør brug af *screen space* afledte eller specielle funktioner som kan *anti-aliase* specifikke proceduralt modellerede strukturer. Det kan være en stor udfordring at tilføje disse former for *anti-aliasing* til komplicerede procedurale modeller, så som et alternativ vælges det oftest i stedet at anvende en strategi som er mere praktisk at implementere og som fungerer på alle former for proceduralt modellerede strukturer [Ebert et al, 2003].

En sådan strategi, er at udføre *anti-aliasing* vha. en form for *screen space*-baseret supersampling, hvor pixelfarven **p** baseres på samples taget rundt om den originale fragmentposition, i stedet for blot at basere farven på den enkelte position, hvilket vil resultere i en blødere overgang i mellem forskellige farver. Den generelle strategi, som er forklaret i [Akenine-Möller et al, 2008], er at anvende et specifikt *sampling pattern* til at sample forskellige *screen space* positioner, og så summe og tage gennemsnittet af de samplede farver, for at opnå pixelfarven **p**:

$$\mathbf{p}(x,y) = \sum_{i=1}^{n} w_i \mathbf{c}(i,x,y) \tag{3.49}$$

Hvoraf n er antallet af samples, w_i er vægten af hver farve (oftest valgt til at være $1/n$) og $\mathbf{c}(i,x,y)$ er farven af det samplede punkt. I praksis kan man vælge at sample over så stort et område der ønskes, men et godt bud er at sample i et område der svarer til 1x1 eller 2x2 pixels rundt om den originale fragmentposition. Et sample i *screen space* kan så henholdsvis findes ved at lægge en forskydning $(x_f, y_f) \in$ [-0.5,0.5]

eller $(x_f, y_f) \in$ [-1,1] til fragmentpositionen.

I stedet for at sætte et specifikt *sample pattern* op, kan en *pseudo-random number generator*, som givet et bestemt *seed* altid vil generere den samme sekvens af pseudo-tilfældige tal, anvendes til at bestemme de pseudo-tilfældige forskydninger. En sådan *pseudo-random number generator*, som blev præsenteret i [Knuth, 1969], og bl.a. er blevet anvendt i [Frisvad og Wyvill, 2007], er givet ved:

$$I_j = (aI_{j-1} + c) \mod m \tag{3.50}$$

Denne generator genererer en sekvens af heltal I_1, I_2,... , som alle ligger i intervallet [0,m-1], og en division med m er derfor nødvendig for at opnå et tal i intervallet [0,1[. I_1 bestemmes i hver pixel ud fra det samme *seed* I_0, og der anvendes $a = 3125$, $c = 49$ og $m = 65536$ (maksimalt positivt heltal). Således kan det første sample ($x_{w,s}$, $y_{w,s}$, $z_{w,s}$) i *window space* findes ved at lægge en forskydning til fragmentets position ($x_{w,c}$, $y_{s,c}$), hvor der her samples inden for et område på 2x2 pixels. Bemærk desuden at der blot anvendes den originale z-koordinat, der beskriver fragmentets dybde, hvilket der vil tages hensyn til senere:

$$\begin{bmatrix} x_{w,s} \\ y_{w,s} \\ z_{w,s} \end{bmatrix} = \begin{bmatrix} x_{w,c} + \; (2\,\frac{I_1}{m} - 1) \\ y_{s,c} + \; (2\,\frac{I_2}{m} - 1) \\ z_{s,c} \end{bmatrix} \tag{3.51}$$

Givet sådan et sample i *window space*, ønskes det nu at bestemme farven af det, således at det kan bidrage til den endelige farve i ligning 3.49. For at gøre dette må grafik pipeline transformationerne inverteres, så samplets koordinater i *world space* kan findes, da de procedurale modeller beskrevet i afsnit 3.3.3, er baseret på disse koordinater. Inverteringen af denne pipeline er beskrevet i nedenstående afsnit 3.5.1, mens der i afsnit 3.5.2 forklares hvordan der kan tages hensyn til det faktum at de fundne *window space* samples anvender det originale fragments dybde.

3.5.1 Invertering af grafik pipeline transformationerne

I afsnit 3.4.1 er de traditionelle grafik pipeline transformationer, hvor der transformeres fra *world space* til *window space*, beskrevet. I dette afsnit beskrives det omvendte - altså transformationen fra *window space* til *world space*. For et punkt (x_w, y_w, z_w) i *window space*, på en skærm med bredde b, højde h, *near-plane* $n = 0$ og *far-plane* $f = 1$, gælder det at $x_w \in [0,b]$, $y_w \in [0,h]$ og $z_w \in [n,f]$. Mappingen fra *window space* til *ndc space*, hvor x_{ndc}, y_{ndc} og $z_{ndc} \in$ [-1,1], er derfor givet ved:

$$\begin{bmatrix} x_{ndc} \\ y_{ndc} \\ z_{ndc} \end{bmatrix} = \begin{bmatrix} 2\,\frac{x_w}{b} - 1 \\ 2\,\frac{y_w}{h} - 1 \\ \frac{2\,z_w - f - n}{f - n} \end{bmatrix} \tag{3.52}$$

4D *clip space* koordinaterne opnås ved at multiplicere med w-komponenten af *clip space* koordinaterne, hvilket er det omvendte af en såkaldt *perspective divide*. For at kunne udføre denne beregning, må man antage at w_{clip_f}, som kendes fra det originale fragmentpunkt kan anvendes til multipliceringen, hvilket er en rimelig antagelse, når der samples indenfor et så relativt lille område:

$$\begin{bmatrix} x_{clip} \\ y_{clip} \\ z_{clip} \\ w_{clip} \end{bmatrix} = \begin{bmatrix} x_{ndc}\ w_{clip_f} \\ y_{ndc}\ w_{clip_f} \\ z_{ndc}\ w_{clip_f} \\ w_{clip_f} \end{bmatrix} \tag{3.53}$$

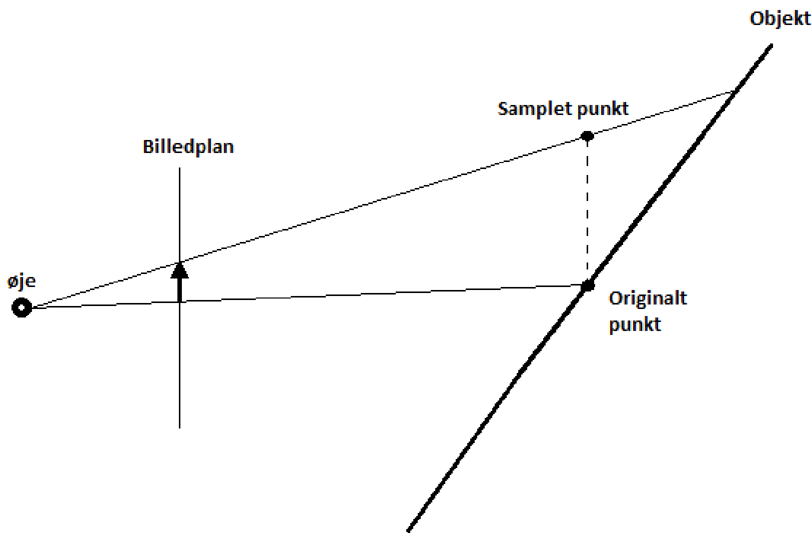
Disse kan transformeres til *eye space* koordinater vha. den inverse projektionsmatrix, som endelig transformeres til *world space* koordinater vha. den inverse *view*-matrix:

$$\begin{bmatrix} x_{eye} \\ y_{eye} \\ z_{eye} \\ w_{eye} \end{bmatrix} = \mathbf{P}^{-1} \begin{bmatrix} x_{clip} \\ y_{clip} \\ z_{clip} \\ w_{clip} \end{bmatrix} , \begin{bmatrix} x_{world} \\ y_{world} \\ z_{world} \\ w_{world} \end{bmatrix} = \mathbf{V}^{-1} \begin{bmatrix} x_{eye} \\ y_{eye} \\ z_{eye} \\ w_{eye} \end{bmatrix} \tag{3.54}$$

Hvis et fundet sample inverteres med denne metode, vil den fundne *world space* position ikke være korrekt, da det originale fragments dybde i *window space* er anvendt. Inverteringen må derfor inkludere en håndtering af dette, hvor en mulig løsning beskrives i nedenstående afsnit.

3.5.2 Håndtering af dybden

Der vendes nu tilbage til det faktum at den samme z-koordinat er blevet anvendt uafhængigt af den samplede (x,y) *screen space* koordinat. Dette må håndteres, hvilket vælges at gøres i *eye space*, da *z-koordinaten* er lineær i dette rum, men nedenstående løsning på problemet kan også modificeres til at blive udført i f.eks. *world space*. Som det ses afbilledet i figur 3.13, består problemet i at det samplede punkt vil have samme dybde z_{eye} som det originale punkt, men en *tracing* af strålen fra øjet i gennem det forskudte punkt, vil ikke nødvendigvis skære overfladen der. Derfor er det altså ikke det korrekte *eye space* koordinat af det samplede punkt der er blevet fundet.



Figur 3.13: Når skærmkoordinaterne (x_s, y_s) forskydes, og der anvendes ovenstående metode til at mappe til *eye space*, vil det samplede punkt, have samme dybde z som det oprindelige punkt.

Den åbenlyse løsning til dette problem er at *trace* strålen fra øjet i gennem det samplede punkt, og så undersøge hvor strålen skærer objektet. Da der samples indenfor et relativt lille område, virker det rimeligt at repræsentere overfladen af objektet ved en første-ordens approksimation, hvorfor der altså kan anvendes *ray-plane intersection* til at finde afstanden til det faktiske skæringspunkt, som beskrevet i [Hughes et al, 2014]. Et netop samplet punkt i *eye space* kan, når det normaliseres, beskrive retningen $\mathbf{d}$ af strålen med oprindelse $\mathbf{o}$ i øjet, således at strålen er givet ved:

$$\mathbf{r} = \mathbf{o} + \mathbf{d}t = \mathbf{d}t \, , \; t \geq 0 \tag{3.55}$$

Hvor det ses at oprindelsen $\mathbf{o}$ af strålen går ud af ligningen, da øjet i *eye space* er placeret i origo. Det originale punkt $\mathbf{q}$ (i *eye space*) og den dertilhørende normal $\mathbf{n}$ (i *eye space* vha. normalmatricen), kan så anvendes til at beskrive planen som det ønskes at finde strålens skæringspunkt med. Følgende må derfor gælde for et hvert givent punkt $\mathbf{p}$, der ligger på planen:

$$(\mathbf{p} - \mathbf{q}) \cdot \mathbf{n} = 0 \tag{3.56}$$

Derfor må strålen skære planen, når følgende er opfyldt:

$$((\mathbf{o} + \mathbf{d}t) - \mathbf{q}) \cdot \mathbf{n} = 0 \tag{3.57}$$

Afstanden t mellem oprindelsen af strålen $\mathbf{o}$ og skæringspunktet er derfor givet ved:

$$t = \frac{(\mathbf{q} - \mathbf{o}) \cdot \mathbf{n}}{\mathbf{d} \cdot \mathbf{n}} = \frac{\mathbf{q} \cdot \mathbf{n}}{\mathbf{d} \cdot \mathbf{n}} \tag{3.58}$$

Hvor der ikke vil være nogen skæring når nævneren er lig nul, da strålen og planen er parallelle i dette tilfælde. Givet afstanden t, kan det nye approksimerede, men mere korrekte *eye space* koordinat findes ved at multiplicere den normaliserede retningsvektor $\mathbf{d}$ med afstanden t:

$$\begin{bmatrix} x_{eye,s} \\ y_{eye,s} \\ z_{eye,s} \\ w_{eye,s} \end{bmatrix} = \begin{bmatrix} d_x t \\ d_y t \\ d_z t \\ 1 \end{bmatrix} \tag{3.59}$$

Og som beskrevet i ovenstående afsnit, kan den nye *world space* koordinat findes vha. den inverse *view* matrix. Dette *world space* koordinat anvendes så til proceduralt at bestemme den samplede farve $\mathbf{c}(i, x, y)$ (ved at udføre lysberegninger), som indsættes i ligning 3.49. Den endelig pixel farveværdi er så gennemsnittet af de n samplede farver, der findes ved at gentage ovenstående proces n gange, hvilket dermed slutter *anti-aliasing* algoritmen.

Når der i algoritmen anvendes *ray-plane intersections* svarer dette, som sagt, til en første-ordens approksimation af overfladen, hvilket virker rimeligt når det samples inden for så lille et område som der gør, men hvis man f.eks. arbejder med en sfære, kan et bedre resultat opnås ved at anvende *ray-sphere intersection*. Dette kan bl.a. være nyttigt i forhold til case 2 i afsnit 3.3.3.2, hvor det forsøges at proceduralt modellere en stofkugle. Derfor vil *ray-sphere intersections* også kort gennemgås her.

En stråle kan defineres på samme måde som i ligning 3.55. Givet en sfære med center $\mathbf{q}$ (i *eye space*) og radius r, må et hvert punkt $\mathbf{p}$ på sfæren opfylde:

$$(\mathbf{p} - \mathbf{q})(\mathbf{p} - \mathbf{q}) = r^2 \tag{3.60}$$

Strålen må derfor skære sfæren når følgende er opfyldt:

$$((\mathbf{o} + \mathbf{d}t) - \mathbf{q})((\mathbf{o} + \mathbf{d}t) - \mathbf{q}) = r^2 \tag{3.61}$$

Som også forklaret i [Hughes et al, 2014], kan afstanden t til skæringspunktet derfor findes ved (der vil være 0, 1 eller 2 skæringer, da der er tale om et 2. grads polynomium):

$$t = -(\mathbf{o} - \mathbf{q}) \cdot \mathbf{d} \pm \sqrt{((\mathbf{o} - \mathbf{q}) \cdot \mathbf{d})^2 - ((\mathbf{o} - \mathbf{q}) \cdot (\mathbf{o} - \mathbf{q}) - r^2)} \tag{3.62}$$

Hvor der ikke vil være noget skæringspunkt når udtrykket inde i kvadratroden er negativt, og ellers vil man oftest anvende det skæringspunkt der er tættest på strålens oprindelse **o**. Når man f.eks. som i case 2, anvender blending vil det dog være nødvendigt at undersøge hvilket skæringspunkt der skal anvendes. Afstanden t, anvendes så på samme måde, som tidligere forklaret, til at finde det korrekte *eye space* koordinat.

3.6 Animation

Når man arbejder med proceduralt modellerede strukturer, hvor udseendet afhænger af positionen af kameraet/øjet, vil det naturligvis være interessant at kunne navigere dette rundt i scenen, således at objektet kan inspiceres fra forskellige synsvinkler. Derudover kan det også være ønskeligt at kunne navigere et givent objekt rundt i scenen, hvilket bl.a. vil være nyttigt i fht. at kunne lade en modelleret struktur afhænge af *world space* koordinaterne - således at strukturen f.eks. kan være låst fast omkring *world space* y-aksen, selvom objektet ikke er.

Navigation af kamera og objekter er oftest implementeret i real-tids 3D-applikationer, vha. en virtuel trackball, der er forbundet til en computermus inputenhed. En vigtig del af sådan en virtuel trackball er brugen af rotationer, som f.eks. kan repræsenteres vha. såkaldte kvaternioner, og disse vil derfor først præsenteres i afsnit 3.6.1, hvorefter kamera-navigation vil gennemgås i afsnit 3.6.2 og objekt-navigation i afsnit 3.6.3.

3.6.1 Kvaternioner

Det matematiske koncept kvaternioner blev allerede opfundet i 1843 af Sir William Rowan Hamilton, men det var først i 1985 at [Shoemake, 1985] introducerede brugen af disse til computergrafikken. Kvaternioner kan beskrive rotationer og orienteringer på en kompakt måde, og er på mange punkter overlegen i fht. både rotationsmatricer og euler vinkler [Akenine-Möller, 2008]. Hensigten med dette afsnit er ikke at forklare den matematiske baggrund for kvaternioner, men nærmere at vise hvad de kan bruges til, som gjort i [Akenine-Möller, 2008].

En kvaternion kan repræsenteres ved: $\hat{\mathbf{q}} = (\mathbf{q}_v,\ q_w) = (q_x,\ q_y,\ q_z,\ q_w)$, og de mest almindelige operationer for sådan en kvaternion ses herunder:

$$\begin{aligned}
&\textbf{Multiplikation} : \hat{\mathbf{q}}\hat{\mathbf{r}} = (\mathbf{q}_v \times \mathbf{r}_v + r_w\mathbf{q}_v + q_w\mathbf{r}_v,\, q_w r_w - \mathbf{q}_v \cdot \mathbf{r}_v) && (3.63)\\
&\textbf{Addition} \quad : \hat{\mathbf{q}} + \hat{\mathbf{r}} = (\mathbf{q}_v + \mathbf{r}_v,\, q_w + r_w) && (3.64)\\
&\textbf{Konjugeret} \quad : \hat{\mathbf{q}}^* = (-\mathbf{q}_v,\, q_w) && (3.65)\\
&\textbf{Norm} \quad : n(\hat{\mathbf{q}}) = \sqrt{q_x^2 + q_y^2 + q_z^2 + q_w^2} && (3.66)\\
&\textbf{Identitet} \quad : \hat{\mathbf{i}} = (\mathbf{0}, 1) && (3.67)\\
&\textbf{Invers} \quad : \hat{\mathbf{q}}^{-1} = \frac{1}{n(\hat{\mathbf{q}})^2}\hat{\mathbf{q}}^* && (3.68)
\end{aligned}$$

En enhedskvaternion er en kvaternion hvorom der gælder at $n(\hat{\mathbf{q}}) = 1$, og sådan en kan derfor beskrives ved ligning 3.69, hvis $\mathbf{u}$ er en 3D vektor, hvorom det gælder at $||\mathbf{u}|| = 1$:

$$\hat{\mathbf{q}} = (\sin\phi\,\mathbf{u}, \cos\phi) \tag{3.69}$$

En sådan enhedskvatertion kan repræsentere enhver 3D rotation, eller mere præcist kan den repræsentere en rotation på 2ϕ rundt om aksen $\mathbf{u}$. Hvis det eksempelvis ønskes at rotere punktet eller vektoren $\mathbf{p} = (p_x,\, p_y,\, p_z,\, p_w)$, beskrevet med homogene koordinater, gøres dette ved at indsætte de 4 koordinater i hver sin komponent i en kvaternion $\hat{\mathbf{p}}$. Hvis det antages at $\hat{\mathbf{q}}$ er en enhedskvaternion, så vil nedenstående udtryk rotere $\hat{\mathbf{p}}$, og altså dermed $\mathbf{p}$, 2ϕ rundt om $\mathbf{u}$:

$$\hat{\mathbf{q}}\hat{\mathbf{p}}\hat{\mathbf{q}}^{-1} = \frac{\hat{\mathbf{q}}\hat{\mathbf{p}}\hat{\mathbf{q}}^*}{n(\hat{\mathbf{q}})} = \hat{\mathbf{q}}\hat{\mathbf{p}}\hat{\mathbf{q}}^* \tag{3.70}$$

Hvor det af ligning 3.68 kan ses at $\hat{\mathbf{q}}^{-1} = \hat{\mathbf{q}}^*$, da der er tale om en enhedskvaternion. Det kan være nyttigt at transformere en enhedskvaternion $\hat{\mathbf{q}}$ til en matrix $\mathbf{A}^q$, således at multiplikationen af denne med punktet $\mathbf{p} = (p_x,\, p_y,\, p_z,\, p_w)$, vil resultere i den samme rotation som beskrevet i ligning 3.70. En sådan matrix er for enhedskvaternioner givet ved:

$$\mathbf{A}^q = \begin{bmatrix} 1-2(q_y^2+q_z^2) & 2(q_xq_y - q_wq_z) & 2(q_xq_z + q_wq_y) & 0 \\ 2(q_xq_y + q_wq_z) & 1-2(q_x^2+q_z^2) & 2(q_yq_z - q_wq_x) & 0 \\ 2(q_xq_z - q_wq_y) & 2(q_yq_z + q_wq_x) & 1-2(q_x^2+q_y^2) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.71}$$

Enhedskvaternioner kan også anvendes til at specificere rotationen fra en retning $\mathbf{s}$ til en anden retning $\mathbf{t}$. Givet enhedsvektorerne $\mathbf{s}$ og $\mathbf{t}$, som beskriver 2 retninger i rummet, kan enheds rotationsaksen $\mathbf{u}$ bestemmes ved: $\mathbf{u} = (\mathbf{s} \times \mathbf{t})/||\mathbf{s} \times \mathbf{t}||$. Hvis 2ϕ svarer til vinklen mellem $\mathbf{s}$ og $\mathbf{t}$, må følgende gælde: $\mathbf{s}\cdot\mathbf{t} = \cos(2\phi)$ og $||\mathbf{s}\times\mathbf{t}|| = \sin(2\phi)$.

Enhedskvaternionen $\hat{\mathbf{q}}$, som beskriver rotationen fra $\mathbf{s}$ til $\mathbf{t}$, må derfor være givet ved $\hat{\mathbf{q}} = (\sin\phi\,\mathbf{u}, \cos\phi)$ hvilket vha. nogle trigonometriske beregninger kan simplificeres til nedenstående:

$$\hat{\mathbf{q}} = (\frac{\sin\phi}{\sin 2\phi}(\mathbf{s}\times\mathbf{t}),\ \cos\ \phi) = (\frac{1}{\sqrt{2(1+\mathbf{s}\cdot\mathbf{t})}}(\mathbf{s}\times\mathbf{t}),\ \frac{\sqrt{2(1+\mathbf{s}\cdot\mathbf{t})}}{2}) \tag{3.72}$$

Hermed slutter den korte præsentation af kvaternioner, som i nedenstående afsnit vil anvendes til at beskrive rotationerne ved brug af henholdsvis kamera- og objektnavigation.

3.6.2 Kamera-navigation

Der vil i dette afsnit beskrives hvorledes en virtuel *trackball* kan anvendes til interaktiv kameranavigation.

En virtuel *trackball* kan ses som en sfære, der er placeret i midten af scenen i *view space* eller måske i midten af et specifikt objekt i scenen. En sådan *trackball* kan styres vha. en computermus, hvor forskellige muse *events* kan definere hvorvidt kameraet skal roteres, panorere eller flyttes frem/tilbage.

Trackballens centrum $\mathbf{l}$ initialiaseres først ved en specificeret startposition $\mathbf{l}_s$, hvoraf $\mathbf{l}$ altså vil være det punkt kameraet roterer rundt om og peger hen i mod - altså er der tale om kameraets såkaldte 'lookat' punkt. Derudover specificeres også en distance z_{eye}, som beskriver hvor langt hen af z-aksen kameraet som udgangspunkt skal placeres i fht. centeret $\mathbf{l}_s$, hvilket bestemmes som afstanden mellem $\mathbf{l}_s$ og den specificerede start kameraposition $\mathbf{e}_s$. *Trackballen* har en translationsvektor $\mathbf{t} = (t_x,\ t_y,\ t_z)$, som kan modificeres ved et specifikt muse *event*, og dermed translatere kameraet. Denne anvendes i *view space*, og derfor vil en modificering af $(t_x,\ t_y)$ resultere en panorering i mens modificering af t_z resulterer i en frem/tilbage bevægelse.

Enhedskvaternionen $\hat{\mathbf{q}}_{rot}$ beskriver den aktuelle rotation af kameraet, og er dermed også basis for *view space* koordinatsystemet. Denne initialeres som rotationen fra z-aksen til retningen mod den specificerede start kameraposition $\mathbf{e}_s$, hvilket opnås vha. ligning 3.72. Enhedskvaternionen $\hat{\mathbf{q}}_{inc}$ anvendes til at beskrive rotationen fra den forhenværende synsretning til en ny en, specificeret af et muse *event*. En ny synsretning, findes vha. positionen af computermusen, som projekteres op på den sfære der beskriver *trackballen*. Da *trackballen* er placeret i midten af scenen $\mathbf{l}$ i *view space*, som også er det punkt som kameraet altid peger i mod, vil dette nye projekterede punkt kunne specificere en ny synsretning. Igen kan rotationen $\hat{\mathbf{q}}_{inc}$ fra den forhenværende til den nye synsretning bestemmes vha. ligning 3.72.

For at kunne beskrive kameraets orientering må en *view* matrix opstilles, og som

beskrevet i afsnit 3.4.1, kan dette gøres blot ved hjælp af kamerapositionen $\mathbf{e}$, 'lookat' punktet $\mathbf{l}$ samt en op-retning $\mathbf{o}$. I følge [Frisvad, 2004] kan disse bestemmes på nedenstående måde, hvis man vælger at anvende $(\mathbf{k}_x, \mathbf{k}_y, \mathbf{k}_z) = ((1,0,0),(0,1,0),(0,0,1))$, som ortonormal basis for kameraets orientering:

$$\hat{\mathbf{o}} = \hat{\mathbf{q}}_{rot}\hat{\mathbf{k}}_y\hat{\mathbf{q}}_{rot}^* \tag{3.73}$$

$$\hat{\mathbf{l}} = t_y\hat{\mathbf{o}} + t_x\hat{\mathbf{q}}_{rot}\hat{\mathbf{k}}_x\hat{\mathbf{q}}_{rot}^* \tag{3.74}$$

$$\hat{\mathbf{e}} = \hat{\mathbf{q}}_{rot}((z_{eye} + t_z)\hat{\mathbf{k}}_z)\hat{\mathbf{q}}_{rot}^* + \hat{\mathbf{l}} \tag{3.75}$$

Hvor hver resulterende kvaternion svarer til den ønskede vektor i homogene koordinater. Det bemærkes at første gang disse beregnes, vil resultatet være de vektorer som specificerer start kamera orienteringen, da $\hat{\mathbf{q}}_{rot}$ er initialiaseret som den rotation der er nødvendig for at rotere z-aksen over til retningen mod den specificerede start kameraposition. Det ses at den nye op-retning blot kan findes ved at rotere den originale basis op-vektor, i fht. den aktuelle rotation af kameraet, i mens udregningen af de 2 øvrige vektorer, må tage hensyn til panorering eller frem/tilbage bevægelse. Altså findes det nye 'look-at' punkt $\mathbf{l}$ ved at translatere langs det nye roterede systems x og y-akse, mens kamerapositionen findes ved at rotere den originale kameraposition plus evt. frem/tilbage translation, som antages at ligge langs z-aksen. Bemærk at hvis der er specificeret et start 'lookat' punkt $\mathbf{l}_s$, som er forskelligt fra 0, vil det være nødvendigt at tilføje denne i ligning 3.74.

Kameraet kan nu roteres trinvist rundt i scenen, ved at udregne $\hat{\mathbf{q}}_{rot} = \hat{\mathbf{q}}_{rot}\hat{\mathbf{q}}_{inc}$ for hvert *frame*, da $\hat{\mathbf{q}}_{inc}$, som sagt beskriver rotationen fra den forhenværende synsretning til en ny, bestemt af muse *events*.

3.6.3 Objekt-navigation

Navigation af kameraet, som forklaret i ovenstående afsnit, kan udvides til navigation af et objekt. Når det ønskes at navigere et objekt, må kameraet fryses fast i dens givne position, hvilket betyder at *view* transformations matricen, specificeret ved $\hat{\mathbf{o}}$, $\hat{\mathbf{l}}$ og $\hat{\mathbf{e}}$, for denne opstilling gemmes. Når det er objektet der skal navigeres, ønskes det at de forskellige muse *events* skal specificere objektets *model* transformations matrix i *world space*. Hvis man vurderer at det er mere intuitivt at disse *muse* events kontrollerer bevægelsen i *view space*, da det er dette rum brugeren arbejder i, kan dette gøres som beskrevet i [Frisvad, 2004].

Panorering og frem/tilbage bevægelse i fht. objektet i *world space* kan udføres ved først at antage at *trackballens* centrum er placeret i objektets centrum i *world space*. På samme måde som i ligning 3.73, 3.74 og 3.75, kan man så opstille følgende 3 kvaternioner, som for at mindske forvirringen, kaldes $\hat{\mathbf{o}}_o$, $\hat{\mathbf{l}}_o$ og $\hat{\mathbf{e}}_o$, :

$$\hat{\mathbf{o}}_o = \hat{\mathbf{q}}_{rot}\hat{\mathbf{k}}_y\hat{\mathbf{q}}^*_{rot} \quad (3.76)$$

$$\hat{\mathbf{l}}_o = t_y\hat{\mathbf{o}}_o + t_x\hat{\mathbf{q}}_{rot}\hat{\mathbf{k}}_x\hat{\mathbf{q}}^*_{rot} \quad (3.77)$$

$$\hat{\mathbf{e}}_o = \hat{\mathbf{q}}_{rot}((t_z)\hat{\mathbf{k}}_z)\hat{\mathbf{q}}^*_{rot} + \hat{\mathbf{l}}_o \quad (3.78)$$

Det bemærkes her at den eneste forskel er at i ligning 3.78 anvendes z_{eye} ikke, da det blot ønskes at flytte objektet baglæns/forlæns i fht. centrum af objektet, og altså ikke øjets placering. Derudover, hvis der er specificeret et start 'lookat' punkt $\mathbf{l}_s$, bør dette heller ikke tilføjes til ligning 3.77, da det også blot ønskes at panorere (flytte objektet til siden/op/ned) i fht. centrum af objektet.

Givet disse 3 kvaternioner, og den viden at objektets centrum i *world space* er placeret i midten af *trackballen*, kan panorering og frem/tilbage bevægelse defineres vha. en translationsmatrix $\mathbf{T}_{obj}$. $\hat{\mathbf{l}}_o$ beskriver centrum af objektet/*trackballen*, i mens $\hat{\mathbf{e}}_o$ kan ses som en form for kameraposition, der som udgangspunkt vil være placeret i objektets centrum (hvilket betyder at objektet ses i sin default-position fra scenens egentlige kamera), og altid vil pege hen mod denne. Hvis et muse *event* specificerer at 'kameraet' eksempelvis skal translateres en hvis mængde bagud, vil samme effekt kunne opnås ved at translatere objektet samme mængde væk fra centrum i den modsatte retning i *world space*. Translationen i $\mathbf{T}_{obj}$ vil derfor altid bestå af en translation fra centrum i den negative retning af 'kameraets' position $\mathbf{e}_o$ i fht. centrum:

$$\mathbf{T}_{obj} = \begin{bmatrix} 1 & 0 & 0 & -e_{o,x} \\ 0 & 1 & 0 & -e_{o,y} \\ 0 & 0 & 1 & -e_{o,z} \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3.79)$$

For at kunne rotere objektet intuitivt i *world space*, må dette først translateres til *world space* koordinatsystemets origo. Givet objektets centrum $\mathbf{C}_{obj}$ i *world space*, vil denne translation være givet ved:

$$\mathbf{T}_{obj,-\mathbf{C}} = \begin{bmatrix} 1 & 0 & 0 & -C_{obj,x} \\ 0 & 1 & 0 & -C_{obj,y} \\ 0 & 0 & 1 & -C_{obj,z} \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3.80)$$

Og efter at have udført de ønskede rotationer, kan objektet translateres tilbage til dens originale position vha. følgende translationsmatrix:

$$\mathbf{T}_{obj,\mathbf{C}} = \begin{bmatrix} 1 & 0 & 0 & C_{obj,x} \\ 0 & 1 & 0 & C_{obj,y} \\ 0 & 0 & 1 & C_{obj,z} \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3.81)$$

Der kan så opstilles en enhedskvaternion $\hat{\mathbf{q}}_{rot,obj}$, som beskriver den aktuelle rotation af objektet, og dermed er basis for *object space* koordinatsystemet. Denne initialiaseres som en identitetskvaternion, hvilket svarer til ingen rotation. Når *trackballen* roteres, kan objektets rotation således beskrives ved at konvertere $\hat{\mathbf{q}}_{rot,obj}$ til en rotationsmatrix $\mathbf{A}^{q_{rot,obj}}$, som beskrevet i ligning 3.71. Endelig kan den ønskede *model* matrix $\mathbf{M}$ opstilles, som først translaterer objektet ind til *world space* origo, roterer objektet, translaterer det tilbage og herfra translaterer det i fht. panorering og frem/tilbage bevægelser:

$$\mathbf{M} = \mathbf{T}_{obj}\,\mathbf{T}_{obj,\mathbf{C}}\,\mathbf{A}^{q_{rot}}\,\mathbf{T}_{obj,-\mathbf{C}} \tag{3.82}$$

Objektet kan så roteres og translateres trinvist rundt i scenen, ved at udregne $\hat{\mathbf{q}}_{rot,obj} = \hat{\mathbf{q}}_{rot,obj}\hat{\mathbf{q}}_{inc,obj}$, da $\hat{\mathbf{q}}_{inc,obj}$ beskriver rotationen fra en forhenværende retning til en ny en, bestemt af muse *events*, som for kamera navigation.

KAPITEL 4

Implementering

4.1 WebGL

Dette afsnit vil kort beskrive *WebGL*, og dennes pipeline, vha. et meget simpelt eksempel. Derudover vil der også beskrives hvor i pipelinen de nødvendige trin i en procedural modellering udføres.

WebGL er et *application programming interface* (API), som kan rendere interaktiv 2D og 3D grafik i en *WebGL*-kompatibel webbrowser. *WebGL* er baseret på *OpenGL* ES 2.0, og har en lignende renderings funktionalitet, blot i en HTML og Javascript kontekst [Khronos group]. *WebGL* anvender HTML5's canvas element, som der dynamisk kan tegnes grafik på, via *scripting* i Javascript. Ved at anvende *WebGL*, får man hardware-acceleret 3D grafik direkte inde i browseren, og uden brug af plug-ins eller installering af ekstra software [Anyurur, 2012].

En *WebGL* applikation består typisk af HTML, Javascript og shader filer (skrevet i GLSL (OpenGL Shading Language)), som eksekveres inde i webbrowseren. Derudover indeholder den også en form for data som repræsenterer 3D modellen/modellerne, der displayes på websiden. *WebGL*'s pipeline består af nedenstående steps, som også ses illustreret i figur 4.1.

- API
- *Vertex shader*
- Samling af primitiver
- Rasterisering
- *Fragment shader*
- Fragment operationer
- Framebuffer

Hvert af disse steps vil kort gennemgås i de nedenstående afsnit, med fokus på step 1, 2 og 5:

API

Dette trin beskriver stort alt det der må defineres før den information, som beskriver hvordan 3D-modellerne skal renderes, kan sendes ned i gennem pipelinen. Først må HTML5 canvas'et defineres i HTML-filen:

```
<canvas id="webgl" width="500" height="500"></canvas>
```

Dette kan så findes i Javascript-filen ved:

```
var canvas = document.getElementById("webgl");
```

En *WebGL* rendering kontekst for canvas elementet må så kreeres:

```
gl = WebGLUtils.setupWebGL(canvas);
```

Bemærk at her anvendes Javascript biblioteket *WebGLUtils*. Der er mange funktioner som ikke er direkte tilgængelige i *WebGL*, så det kan være en stor hjælp af anvende sådanne biblioteker (eksempelvis i fht. kvaternioner, matricer, indlæsning af .obj-filer og initialiasering af shadere). Herefter kreeres 2 shadere, GLSL koden (*vertex* og *fragment shadere*) oploades, shaderne kompileres og linkes til et program, vha *initShaders*:

```
gl.program = initShaders(gl,"vertex-shader", "fragment-shader");
gl.useProgram(gl.program);
```

Da der nu er givet et GLSL program på GPU'en må data sendes til det, som eksempelvis kan være en attribut (variable der muligvis varierer for hvert *vertex*: *vertex* position, normal, farve - og derfor kun kan sendes til *vertex shaderen*) eller en uniform (værdier der er det samme for hele den renderede frame: matricer, lysposition, konstanter). For at sende en attribut må en buffer kreeres, bindes, og dataen puttes ind i den. Her vist for et array af *vertex* positioner der definerer en enkelt trekant:

```
var vBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER,vBuffer);
var vertices =[vec3(0,0,0), vec3(1,1,0), vec3(1,0,0)];
gl.bufferData(gl.ARRAY_BUFFER,flatten(vertices),gl.STATIC_DRAW);
```

Derudover skal det også specificeres hvad for en slags data der er i bufferen, som det ønskes at sende til attributten i programmet, som vi også må *enable*:

```
var vPosition = gl.getAttribLocation(program, "vPosition");
gl.vertexAttribPointer(vPosition,3,gl.FLOAT,false,0,0);
gl.enableVertexAttribArray(vPosition);
```

Endelig kan den opstillede scene renderes, og den ønskede information sendes ind i pipelinen. Dette gøres i sin simpleste form ved følgende, som specificerer at array dataen skal tegnes som trekanter.

```
gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
gl.drawArrays(gl.TRIANGLES,0,3);
```

Vertex shader
Vertex shaderen er den første programmerbare del af pipelinen. *Vertex shaderen* kaldes en gang per *vertex*, og kræver at man sætter den globale variabel gl_Position, som er den givne *vertex*'s position i *clipspace*. I Javascript-filen defineres følgende, som specificerer skærmens højde og bredde i pixels, således at *WebGL* selv kan konvertere *clip space* til *window space* koordinater, når dette skal gøres:

```
gl.viewport(0,0,canvas.width,canvas.height);
```

I sin simpleste form ville en *vertex shader* (skrevet i GLSL) se således ud, hvor det givne vertex i *object space* transformeres ved hjælp af *modelviewprojection* matricen.

```
<script id = "vertex-shader" type = "x-shader/x-vertex">
attribute vec3 vPosition;
uniform vec4 MVPmatrix;

void main()
{
gl_Position = MVPmatrix*vec4(vPosition,1.0);
}
</script>
```

Lysberegninger kan også udføres i *vertex shaderen*, men *phong shading* må udføres i *fragment shaderen*. For at kunne gøre dette må *vertex* attributter som eksempelvis normalen eller retningen mod lyset og kameraet, sendes til *fragment shaderen*, som en *varying*, hvor der så vil linæert interpoleres over normalerne fra de 3 *vertices*. Det samme gøres ofte med *vertex* positionen, så fragmentets position i *object space* også kendes.

Samling af primitiver
I drawArrays-kaldet (eller drawElements, hvis man arbejder med indekserede *vertices*, hvilket oftest vil være tilfældet for trekantsmesh), specificeres det hvilket slags primitiv der skal tegnes. Når der anvendes gl_ TRIANGLES, vil *vertex shaderen* kaldes for en gruppe af 3 *vertices*, hvorefter disse samles til en trekant. *Clipping* foregår også i dette trin således at det kun er primitiver inden for *view volumen* som sendes videre til næste trin.

Rasterisering
I dette trin konverteres primitiverne til fragmenter, som sendes til *fragment shaderen*.

Figur 4.1: *WebGL*'s pipeline, figur fra [Anyuru, 2012].

Fragment shader

Fragment shaderen er den anden programmerbare del af pipelinen, hvortil fragmenterne fra rasteriseringsprocessen sendes en ad gangen. *Fragment shaderen* kræver at man sætter den globale variable gl_FragColor, som beskriver det givne fragments farve. I sin simpleste form, kunne en *fragment shader* (skrevet i GLSL), derfor se således ud, hvilket sætter farven til hvid:

```
<script id="fragment-shader" type="x-shader/x-fragment">
precision mediump float;

void main(){
gl_FragColor = vec4(1.0,1.0,1.0,1.0);
```

```
}

</script>
```

Oftest ønskes det dog i stedet at beregne en farve vha. *phong shading*, og de interpolerede variable sendt fra *vertex shaderen*. Derudover er det også her at man typisk implementerer procedural modellering, såsom teksturering og *bump mapping*, ved at modificere per-fragment shading rutinen, da interpolation af proceduralt valgte attributter fra *vertex shaderen* ville resultere i et ikke ønskeligt udseende. *Fragment shaderen*, har nogle forskellige indbyggede variable, hvor gl_FragCoord, er den vigtigste i fht. implementeringerne i denne rapport. Denne variabel kan nemlig anvendes i *anti-aliasing* algoritmen, som også udføres i *fragment shaderen*, da denne beskriver fragmentets postion i *window space*, givet ved: $(x_w,\ y_w,\ z_w,\ 1/w_w)$.

Fragment operationer
Efter *fragment shaderen*, sendes hvert fragment til denne del af pipelinen, hvor forskellige operationer udføres på hvert fragment, hvoraf disse operationer kan ses indenfor den stiplede linje i figur 4.1. Det er muligt i Javascript-filen at aktivere og deaktivere disse operationer, samt at styre deres funktionalitet. Resultatet af disse operationer anvendes så til at opdatere den korresponderende pixel i framebufferen.

4.2 Interaktivitets interface

I afsnit 3.6 beskrives hvorledes et kamera- og objektnavigations system kan opstilles. Dette afsnit vil derfor blot gennemgå selve det implementerede interface - altså bl.a. hvilke *muse* events der styrer hvilke bevægelser, og hvorledes objektnavigation kan anvendes til både at låse den procedurale modellering fast i fht. *object space* eller *world space*.

4.2.1 Interface

Den implementerede *trackball*, og navigationen af kameraet, er en implementering som oprindeligt blev uddelt i forbindelse med kurset 02561 *Computer Grafik* på DTU. Objektnavigation er således en udvidelse til denne implementering. I fht. både kamera og objektnavigation beskriver de samme muse *events* de samme bevægelser (panorering, zooming og translation), og der er indsat 2 knapper, som styrer hvilken navigation der ønskes. *Trackballens* translationsvektor $\mathbf{t} = (t_x, t_y, t_z)$, som beskriver hvorvidt kameraet skal zoome/panorere eller om objektet skal translateres (for at give samme effekt), kan modficeres vha. 2 forskellige muse *events*. Panorering opnås ved at bevæge musen mens den højre museknap holdes nede, og zooming ved at bevæge musen frem eller tilbage mens den midterste museknap holdes nede.

Når musen bevæges mens den venstre museknap holdes nede, vil positionen af musen projekteres til sfæren, som repræsenterer den virtuelle *trackball*, hvormed denne nye position kan beskrive rotationen. Bemærk at da objektnavigationen udføres i *world space*, kan man risikere at rotationen af og til virker ulogisk, hvis man først har navigeret kameraet rundt i scenen. Alt efter hvad man synes virker mest intuitivt kan det også vælges at anvende den konjugerede $\hat{\mathbf{q}}^*_{rot}$ til at beskrive rotationen. Ved at udføre objektnavigationen i *view space* i stedet, vil rotationen altid virke intuitiv, men kamerarotationen kan så til gengæld hurtigt virke ulogisk, hvis man først har navigeret objektet rundt i scenen.

4.2.2 Objektnavigation

Når objektnavigation er valgt, er implementeringen opbygget således at den procedurale modellering stadig er låst fast i fht. *world space*, hvilket i fht. case 3, kan give den ønskede effekt at brugeren selv kan styre hvilken retning lagene printes i. Dette udtryk opnås ved at sende *modelviewprojection* matricen til vertex-shaderen, så denne kan beregne den korrekte placering af objektet efter eventuel rotation eller translation, i mens det blot er *viewprojection* matricen som sendes til fragment-shaderen, og der, i *anti-aliasing* algoritmen, anvendes til at transformere et sample i *screen space* til *world space*. Dette *world space* koordinat danner så basis for den procedural modellering. Hvis man i stedet sendte *modelviewprojection* matricen til fragment-shaderen, ville den procedurale modellering altid være konsistent med objektet, da den udføres på basis af *object space* koordinater.

Det antages derudover i implementeringen at objektets centrum, som udgangspunkt, er placeret i (0,0,0) i *world space*, og at objektet har en passende størrelsesorden, da lag-inddelingen i *world space* ellers ikke vil give nogen mening. Objektet antages at være defineret som en .obj-fil, og denne kan skaleres når objektet læses ind. Man kunne naturligvis have implementeret skalering og translation af objektet, vha. en *bounding box* teknik, når objektet læses ind, men dette er ikke gjort for at holde koden så simpel som muligt, så man nemt i koden kan se hver af de trin beskrevet i afsnittet om objektnavigation. En nem måde at sikre sig at ens objekt opfylder de nævnte krav, er at importere .obj-filen til et 3D-program, hvor objektet kan translateres og skaleres for tilsidst at blive eksporteret som en ny .obj-fil (*3D Builder* for windows 10 er et godt bud på sådan et program).

4.3 Alpha blending

Dette afsnit kan ses som en meget lille guide, i fht. at arbejde med *alpha blending* i *WebGL*, hvilket i denne rapport er tilfældet for case nr. 2, der modellerer en stofkugle. Der er altså ikke tale om en grundig gennemgang af dette emne.

Alpha blending i *WebGL* gør brug af *alpha*-kanalen, som er den fjerde kanal når der anvendes RGBA (eller RGBα) farver. Når *blending* er aktivereret, anvendes en *blending* funktion til at kombinere farverne fra flere fragmenter, som således alle har mulighed for at bidrage til farven i den samme pixel.

De forskellige trin i implementeringen af *alpha blending* i *WebGL*, vil nu kort gennemgås. Første trin er at kreere en *WebGL Context* på en lignende måde:

```
gl = canvas.getContext("webgl",{alpha:false});
```

Ved at sætte alpha:false, sørger man for at *canvaset* ikke vil have en *alpha* komponent, og derfor ikke have mulighed for at *blende* med baggrunden (på websiden, bag *canvaset*). Herefter aktiveres brugen af *blending*, og den *blending* funktion, som det ønskes at anvende til at kombinere 2 fragmenter med vælges:

```
gl.enable(gl.BLEND);
gl.blendFunc(gl.SRC_ALPHA, gl.ONE_MINUS_SRC_ALPHA);
```

Denne *blending* funktion er meget anvendt. Parametrene i denne funktion, bygger på at man siges at arbejde med et *source* fragment og et *destination* fragment, hvoraf det første er det man netop er ved at tegne og det andet er det som allerede er i framebufferen. Inputtet i funktionen bestemmer således hvordan farverne i disse skal kombineres til en enkelt farve. Hvor eksempelvis den resulterende røde farvekomponent af *WebGL* vil findes ved: $R_{res} = R_s \cdot \alpha_s + R_d \cdot (1 - \alpha_s)$, hvor R_s er den røde farvekomponent for *source* fragmentet, med dertilhørende α_s værdi, og R_d er *destination* fragmentets røde komponent.

Det bliver heraf tydeliggjort at man også må inkludere følgende i sin kode:

```
gl.disable(gl.DEPTH_TEST);
```

Hvis dette ikke er inkluderet, kan man risikere at et delvist gennemsigtigt fragment først tegnes, hvorefter et bagved tegnes, som så vil kasseres af *depth* testen. Dette er dog ikke hele løsningen på problemet, da resultatet af en *blending* er meget afhængig af rækkefølgen hvormed fragmenterne tegnes. For at opnå et optimalt resultat må uigennemsigtige primitiver renderes først, efterfulgt af delvist gennemsigtige primitiver, startende bagfra [OpenGL FAQ]. En nem løsning på dette problem, når der arbejdes med et enkelt objekt, som en sfære, er følgende:

```
gl.enable(gl.CULL_FACE);
gl.cullFace(gl.FRONT);
gl.drawElements(gl.TRIANGLES, number_of_elements, gl.UNSIGNED_SHORT, 0);
gl.cullFace(gl.BACK);
gl.drawElements(gl.TRIANGLES, number_of_elements, gl.UNSIGNED_SHORT, 0);
```

Hvor man ved at anvende *culling*, først kan tegne den del af objektet som består af *back facing* trekanter, og derefter *front facing* trekanter.

KAPITEL 5

Resultater

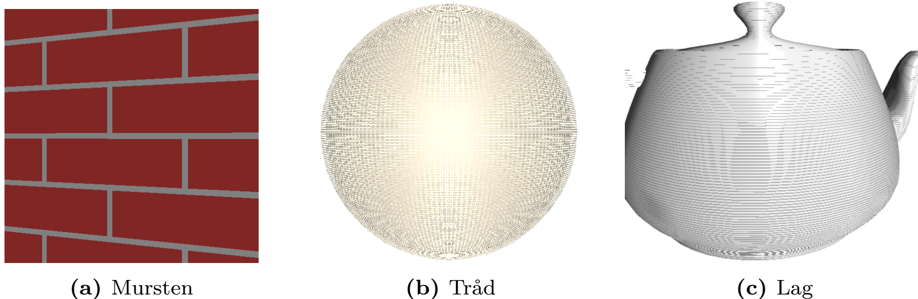
Dette kapitel består af 4 forskellige afsnit, hvoraf det første præsenterer *anti-aliasing* algoritmen, i mens de 3 sidste præsenterer renderinger for hver case, hvor der naturligvis er anvendt *anti-aliasing*. Der er lagt mest fokus på den sidste case, hvor renderingerne bl.a. kvalitativt vil sammenlignes med fotografier, og der vil vises eksempler på interaktiv tilpasning. I alle afsnit er den procedurale modellering og lysberegningerne udført i fragment shaderen (*phong shading*), på basis af *world space* koordinaterne. De forskellige implementeringer kan findes i den til opgaven vedhæftede zip-fil, eller på websiden linket til i appendix A.1.

5.1 Anti-aliasing

Der vil i dette afsnit først præsenteres renderinger der viser de *aliasing*-artifakter der vil ses for de 3 cases. Derefter vil der præsenteres renderinger, hvor det søges at minimere disse artifakter vha. den implementerede *anti-aliasing* algoritme.

5.1.1 Aliasing artifakter

I nedenstående figur præsenteres 3 billeder (et for hver case) hvor ingen *anti-aliasing* er blevet udført, og de procedurale valg altså er baseret på et enkelt sample:



(a) Mursten **(b)** Tråd **(c)** Lag

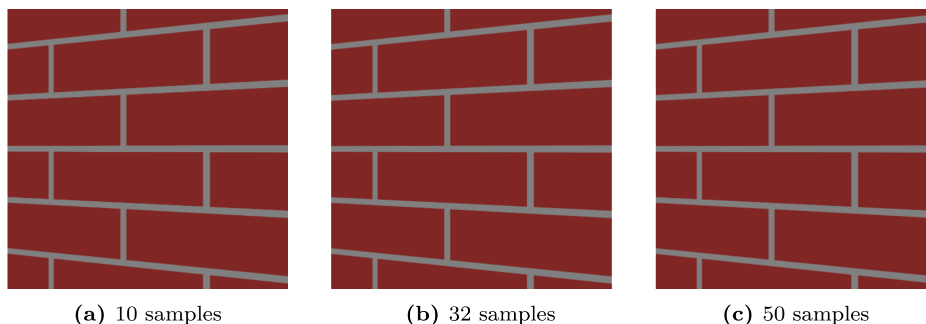
Figur 5.1: *Aliasing-artifakter* for de 3 forskellige cases.

Som forventet ses der i alle 3 cases savtakkede overgange, og i tråd- og lagcasen ses moire-mønstre, grundet de fine, regulære strukturer.

5.1.2 Antal anti-aliasing samples

Der vil her præsenteres renderinger med forskellige antal af *anti-aliasing* samples, for at vurdere hvilket antal der er optimalt. Derudover vil der også præsenteres den visuelle forskel på at sample i et område svarende til 1x1 og 2x2 pixels rundt om fragmentpositionen.

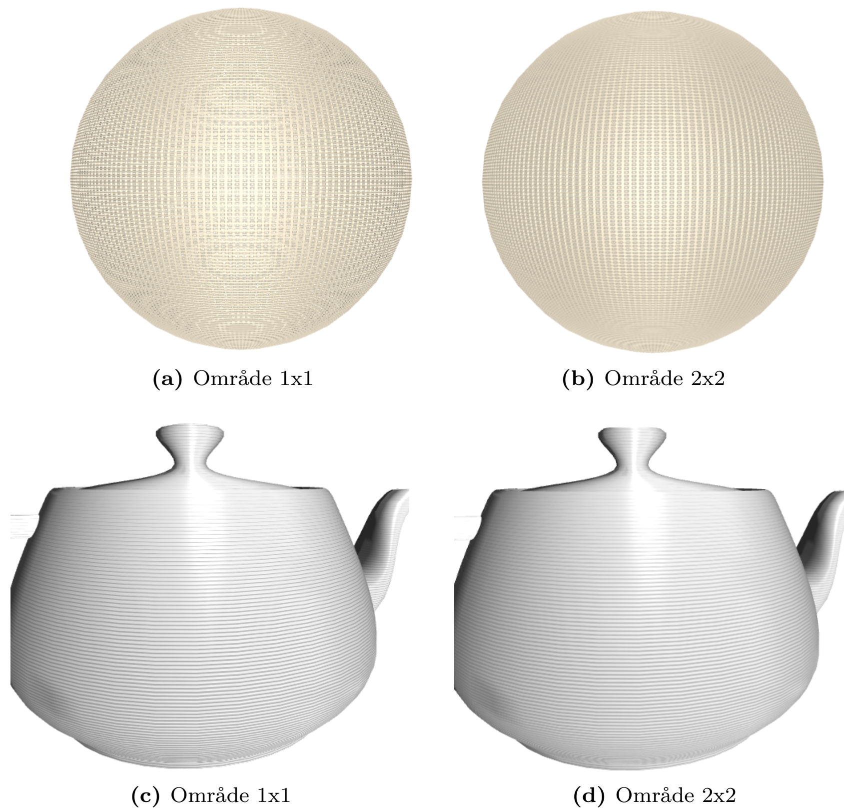
I figur 5.2 ses den proceduralt modellerede murstenstekstur, *anti-aliased* med henholdsvis 10, 32 og 50 samples.



(a) 10 samples **(b)** 32 samples **(c)** 50 samples

Figur 5.2: Murstensteksturen *anti-aliased* ved henholdsvis 10, 32 og 50 *anti-aliasing* samples .

Hvoraf det må kunne udledes at et sted i mellem 32 og 50 *anti-aliasing* samples, er et tilstrækkeligt antal, hvor situationen bestemmer hvad det bør vælges (renderkvalitet vs. rendertid). I ovenstående figur 5.2, samples der i et område på 2x2 pixels rundt om fragmentpositionen. For især lag- og trådcasen er forskellen på at sample indenfor 1x1 og 2x2 pixels synlig, da sidstnævnte tilfælde resulterer i en mindre grad af moire-mønstre, som det ses i figur 5.3 (bemærk dog at mængden af moire-mønstre også naturligvis afhænger af størrelsen af billedet, når det ses på din computerskærm).



(a) Område 1x1 **(b)** Område 2x2

(c) Område 1x1 **(d)** Område 2x2

Figur 5.3: Trådmodellen og lagmodellen samplet, med 50 *anti-aliasing* samples, i områder på henholdsvis 1x1 og 2x2 pixels.

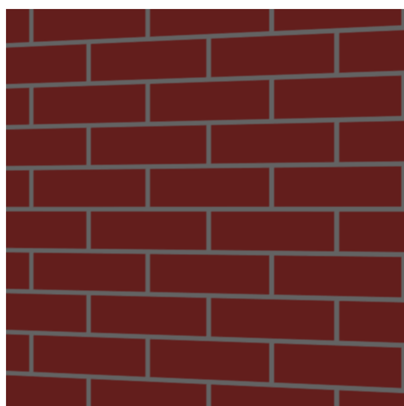
På baggrund af undersøgelserne i dette afsnit, vil renderingerne i de øvrige afsnit udføres med 50 *anti-aliasing* samples, samplet i et område på 2x2 pixels rundt om fragmentpositionen.

5.2 Mursten

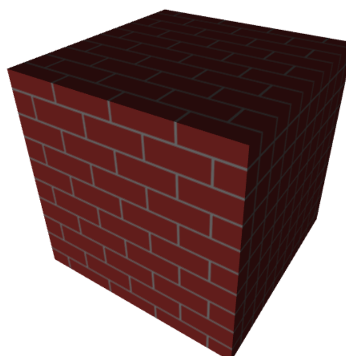
Dette afsnit vil præsentere den procedurale murstensmodel i henholdsvis 2D, 3D, med og uden *bump-mapping*. I alle tilfælde er renderingerne udført vha. en fragment *shader* med en standard *Phong*-model (dog kun med det et diffust og ambient bidrag).

5.2.1 Procedural teksturering

I nedenstående figur præsenteres renderinger af murstensmodellen i henholdsvis 2D og 3D, hvoraf førstnævnte er modelleret på en flade og sidstnævnte på en kasse. Denne model bygger kun på et proceduralt valg af farve, og der er altså ingen *bump-mapping* udført, hvilket resulterer i et fladt og urealistisk udtryk.



(a) 2D

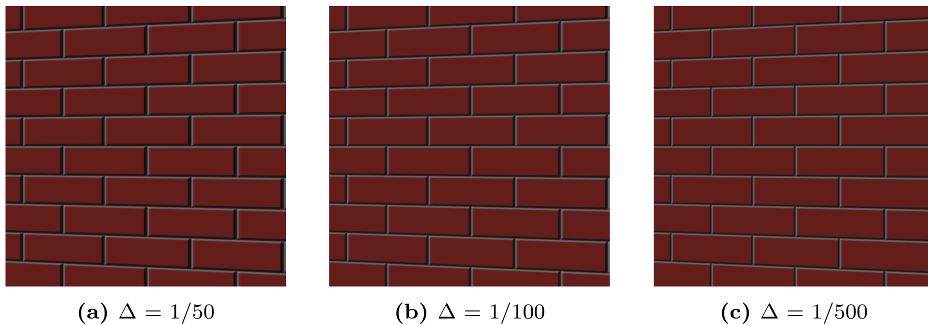


(b) 3D

Figur 5.4: 2D og 3D procedural murstenstekstur, med murstensbredde 0.5, murstenshøjde 0.16 og mørteltykkelse 0.02 (samt murstenslængde 0.19 for 3D).

5.2.2 Bump-mapping

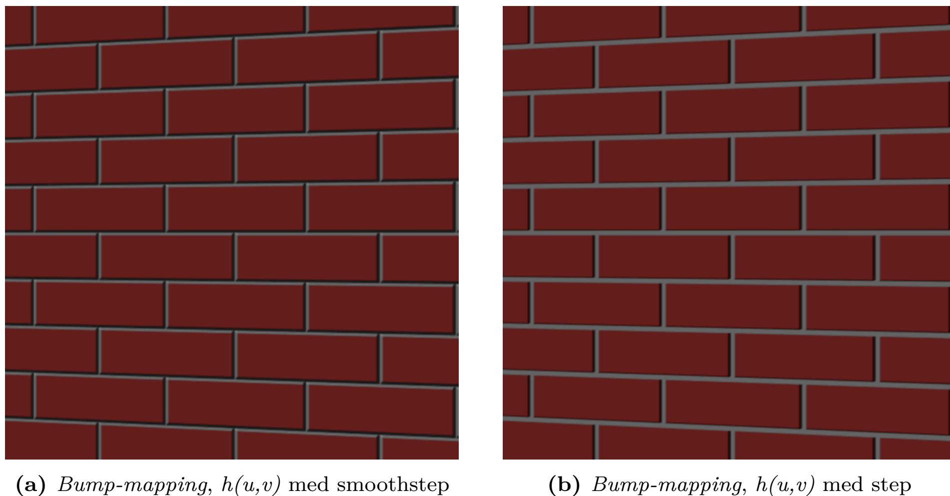
I dette afsnit vises resultatet af at tilføje *bump-mapping* til modellen. I figur 5.5 undersøges hvilke Δ-værdier der resulterer i det mest realistiske udseende, da denne værdi gerne skal kunne opfange en vis højdeforskel.



(a) $\Delta = 1/50$ (b) $\Delta = 1/100$ (c) $\Delta = 1/500$

Figur 5.5: *Bump-mapping* udført med forskellige Δ-værdier, her vidst for en højdefunktion baseret på smoothstep-funktionen .

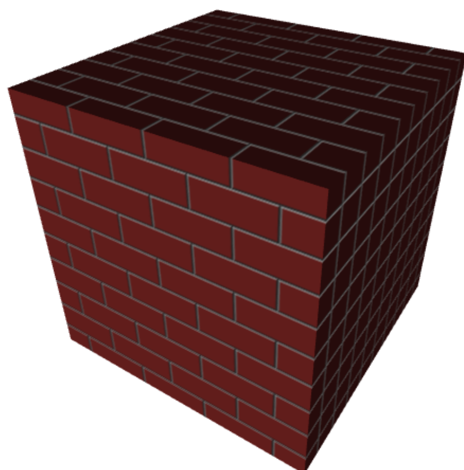
På baggrund af disse resultater, anvendes $\Delta = 1/\mathrm{H} = 1/500$ i de øvrige renderinger. Figur 5.6a, viser resultatet af at anvende en højdefunktionen baseret på smoothstep-funktionen, i mens figur 5.6b, anvender en baseret på step-funktionen, hvilket resulterer i et udtryk med en brat overgang mellem mursten og mørtel. For at kunne opnå sådan et udtryk som ses i figur 5.6b, er det dog nødvendigt at anvende en noget højere H-værdi.



(a) *Bump-mapping*, *h(u,v)* med smoothstep (b) *Bump-mapping*, *h(u,v)* med step

Figur 5.6: *Bump-mapping* udført med forskellige højdefunktioner .

Der ses i begge tilfælde den ønskede effekt, og det kan på skyggerne bemærkes at lyset må være placeret oppe i venstre hjørne. I figur 5.7 ses en 3D tekstur med *bump-mapping*.



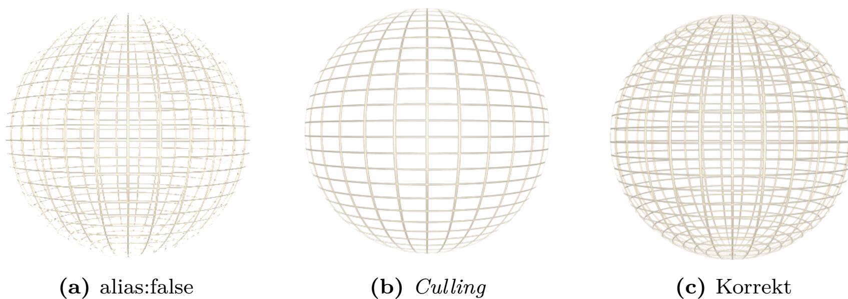
Figur 5.7: 3D murstenstekstur med *bump-mapping*.

5.3 Tråd

Dette afsnit vil præsentere den procedurale trådmodel, og i denne præsentation bl.a. undersøge vigtigheden af korrekt brug af *blending*, en passende lysmodel og det procedurale valg af tangenter.

5.3.1 Korrekt brug af blending

I dette afsnit præsenteres vigtigheden af at anvende *blending* på en korrekt måde, som forklaret i afsnit 4.3. I figur 5.8, ses henholdsvis resultatet af en *blending* uden tilføjelsen af alias:false, uden brug af *culling* (hvilket svarer til det samme resultat, man ville opnå hvis man slet ikke aktiverede *blending*) samt en korrekt udført *blending*.

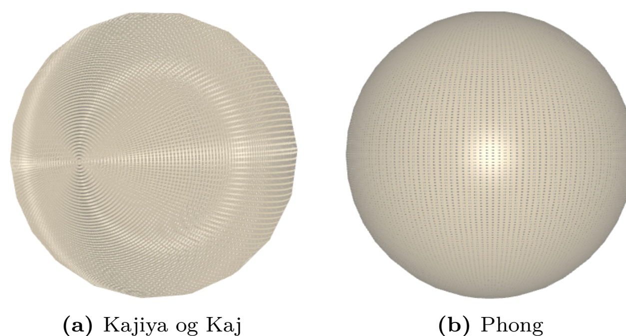


(a) alias:false **(b)** *Culling* **(c)** Korrekt

Figur 5.8: Trådmodellen, hvor hvert billede illustrerer brugen af blending, men uden den nævnte ting, bortset fra det sidste billede, som illustrerer en korrekt blending.

5.3.2 Lysmodel

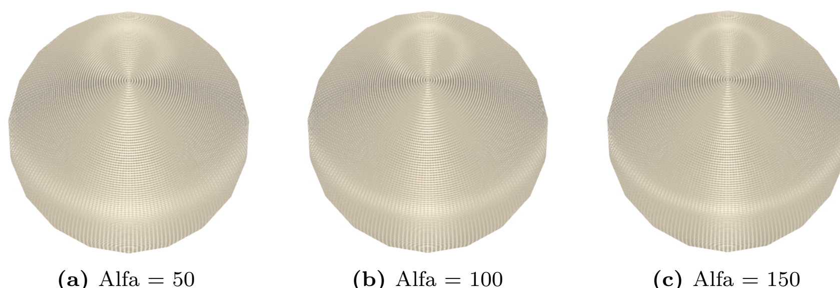
Kajiya og Kaj's lysmodel kan anvendes til at *shade* stofkuglen, således at highlights vil følge fiberstrukturen. I figur 5.9, ses kuglen *shaded* med henholdsvis Kajiyas og Kajs model og en standard Phong model, hvor sidstnævnte resulterer i et plastic-gitter udtryk:



(a) Kajiya og Kaj **(b)** Phong

Figur 5.9: Trådmodellen, *shaded* vha. de 2 modeller. I billedet til venstre ses kuglen fra oven, mens den ses forfra i højre billede .

Der er her anvendt en procedural tekstur, hvor hullet i mellem trådene har en dimension på 0.003x0.003, i mens trådene har en tykkelse på 0.002 hvilket resulterer i de førnævnte moire-mønstre. I figur 5.10, anvendes de samme tekstur-variable, men Phong eksponenten i hår-lysmodellen varieres:



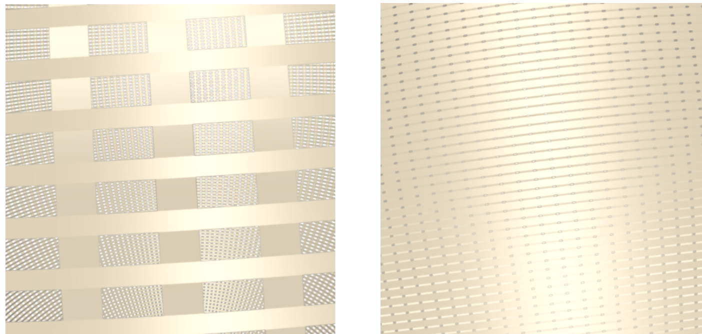
(a) Alfa = 50 **(b)** Alfa = 100 **(c)** Alfa = 150

Figur 5.10: Hår lysmodellen, anvendt med forskellige alfa-værdier (Phong eksponent).

Hvor det som forventet ses at de spekulære highlight bliver mere koncentrerede ved høje alfa-værdier.

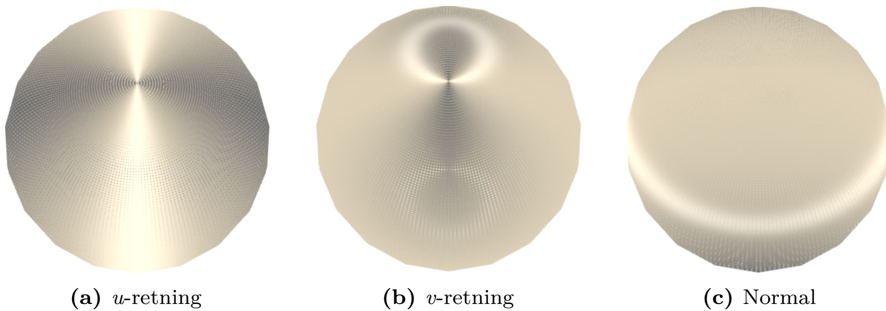
5.3.3 Tangenter

Kajiya og Kaj's model anvender de tangenter som følger fiberstrukturen. Som det ses af figur 5.11, er det i denne procedurale model blot valgt at anvende horisontale tangenter, der hvor trådene mødes:



Figur 5.11: 2 close-ups af trådmodellen.

I nedenstående figur illustreres vigtigheden i at de proceduralt valgte tangenter faktisk følger fiberstrukturen. I de 3 billeder er alle tangenter sat til henholdsvis en tangent der følger u-retningen, v-retningen eller normalen. Hvor man ved at sammenligne med figur 5.10, ser at disse 'kombineres' til det færdige udtryk.



(a) u-retning **(b)** v-retning **(c)** Normal

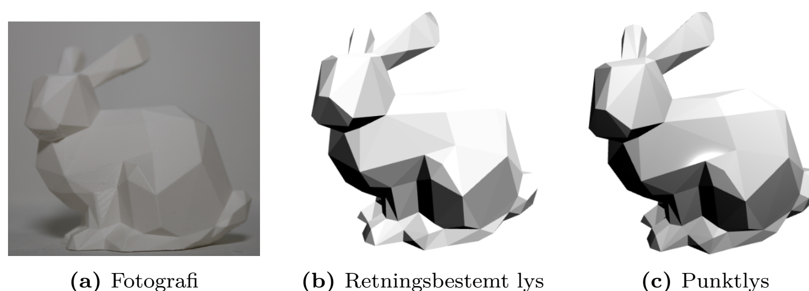
Figur 5.12: 3 tilfælde af trådmodellen, hvor alle tangenter er sat til den beskrevne vektor.

5.4 Lag-inddeling

Dette afsnit vil præsentere den procedurale lag-inddeling/trappestruktur. I første del-afsnit vil det forsøges at et ramme et fotorealistisk udseende (og kvalitativt sammenligne med fotografier, taget i et lyskontrolleret miljø), ved at variere de 2 varianter af Phong reflektionsmodellen. Herefter følger et del-afsnit som vil præsentere eksempler på interaktiv tilpasning.

5.4.1 Fotorealistike renderinger

Nedenstående figur viser et fotografi, taget i et lyskontrolleret miljø, samt 2 3D modeller *shaded* med den modificerede Phong model, med henholdsvis et retningsbestemt og et punktlys. På baggrund af denne figur, besluttes det at lyset i det lyskontrollerede miljø bedst kan beskrives med en retningsbestemt lyskilde.

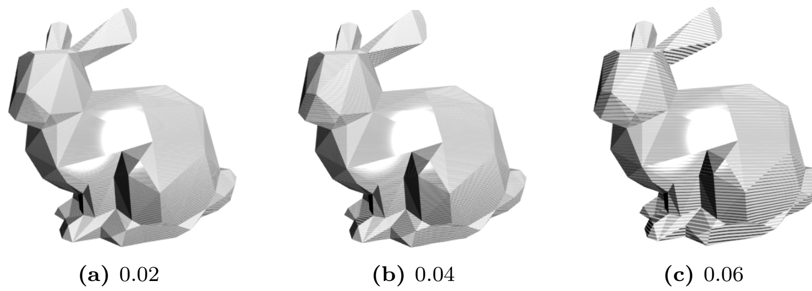


(a) Fotografi **(b)** Retningsbestemt lys **(c)** Punktlys

Figur 5.13: Valg af lyskilde .

I de følgende 2 afsnit vil det forsøges at efterligne udtrykket i figur 5.13a, med henholdsvis den modificerede Phong og Blinn-Phong model:

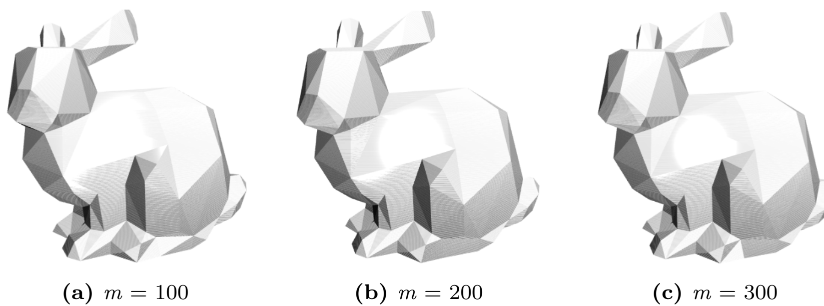
Dem modificerede Phong model
I figur 5.14 søges den optimale lagtykkelse. Det bemærkes at illusionen, af at objektet er opbygget af lag, går i stykker når lagtykkelsen bliver for stor. Tykkelsen 0.02 anvendes i de følgende renderinger.



(a) 0.02 (b) 0.04 (c) 0.06

Figur 5.14: Forskellige lagtykkelser .

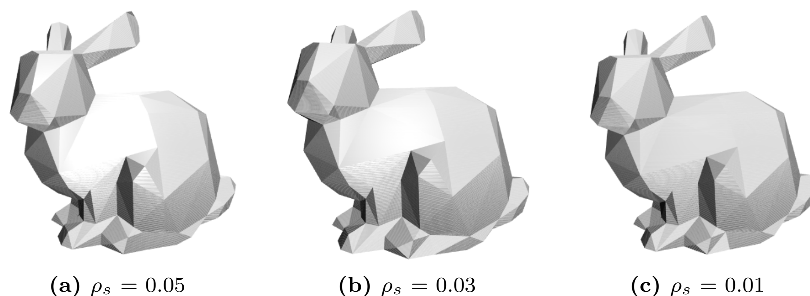
I nedenstående figurer varieres variablene i den modificerede Phong model. I figur 5.15 varieres eksponenten m, som beskriver ruheden af materialet, og styrer hvor koncentreret highlightet er.



(a) $m = 100$ (b) $m = 200$ (c) $m = 300$

Figur 5.15: Forskellige værdier af m.

Da fotografiet i figur 5.13a ikke udviser koncentrerede (cirkulære) highlights, vælges det at gå videre med $m = 100$ i de efterfølgende renderinger. I ovenstående renderinger er der anvendt $\rho_d = 0.9$ og $\rho_s = 0.1$, men ved at sammenligne med fotografiet, bliver det tydeligt at der reflekteres for meget spekulært lys i midten, mens de mindre spekulære reflektioner på hoved og bagende er passende. I figur 5.16 varieres ρ_s derfor.



(a) $\rho_s = 0.05$ (b) $\rho_s = 0.03$ (c) $\rho_s = 0.01$

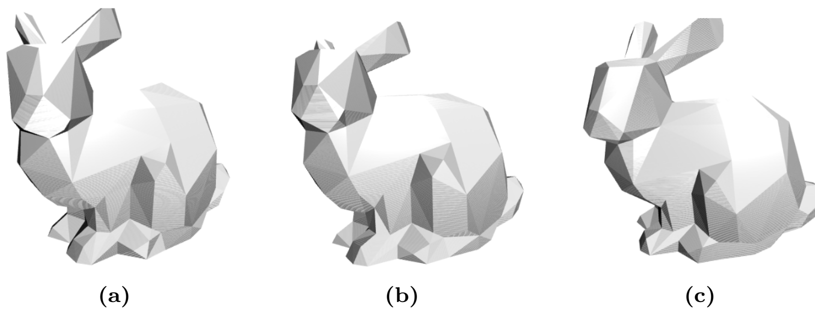
Figur 5.16: Forskellige værdier af ρ_s .

Det bedste bud på en fotorealistisk rendering, med denne model, ses i nedenstående figur, hvor kaninen er roteret, således at highlightet i midten, i stedet er på bagenden. Bemærk at der stadig er tale om et cirkulært highlight:

Figur 5.17: Bedste bud på en fotorealistisk rendering med den modificerede Phong model med $m = 100$ og $\rho_s = 0.05$.

Den modificerede Blinn-Phong model

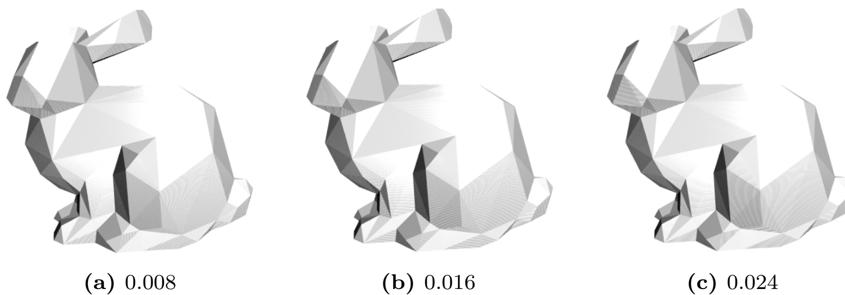
For denne model kan man også variere ruhedsparameteren m, og modsat Phong-modellen er det muligt at modellere ikke cirkulære highlights, som det ses i figur 5.18.



(a) (b) (c)

Figur 5.18: $m = 100$, og $\rho_s = 0.1$, renderinger hvor objektet er roteret.

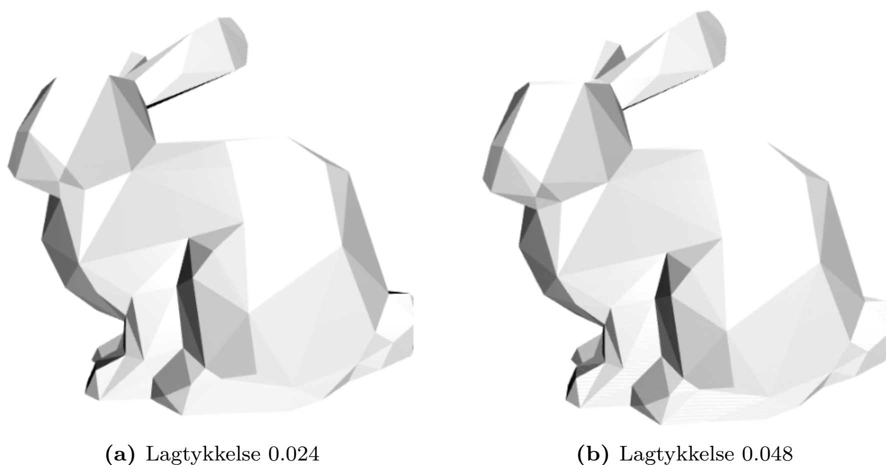
I afsnit 3.2.1 nævnes det at m bør være 4 gange så stor i denne model, som i den forrige for at kunne opnå et lignende resultat. Det vurderes dog at $m = 100$, virker passende, baseret på figur 5.18c. Efter at have fundet en model, som rammer highlightsene relativt godt, re-evalueres lagtykkelsen, hvor forskellige tykkelser kan ses i nedenstående figur. Det bemærkes at disse mere passende lagtykkelser, resulterer i en del moire-mønstre:



(a) 0.008 **(b)** 0.016 **(c)** 0.024

Figur 5.19: Forskellige lagtykkelser.

I et forsøg på at mindske disse moire-mønstre, forsøges det i nedenstående figur at udføre normal-mappingen vha. en passende smoothstep-funktion i stedet for en step-funktion. Det bemærkes at der anvendes lidt tykkere lag, men moire-mønstrene synes også at forsvinde ved de tyndere lag, dog bliver lagstrukturen i disse tilfælde mindre tydelig.



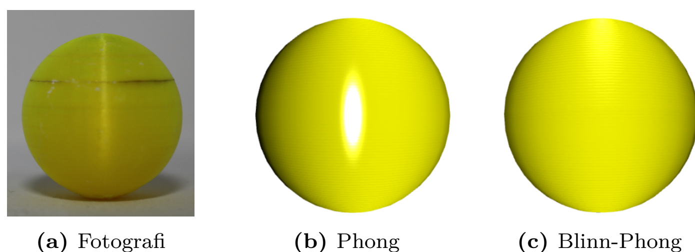
(a) Lagtykkelse 0.024 **(b)** Lagtykkelse 0.048

Figur 5.20: Normalmapping udført med smoothstep(0,0.2,l-t), for 2 forskellige lagtykkelser).

Det bedste bud på en fotorealistisk rendering, med denne model vurderes derfor at være renderingen i figur 5.20b.

Forskellige fotografier

I dette afsnit præsenteres 3 forskellige fotografier af 3D-printede objekter, samt resultatet af at anvende lag-algoritmen med de 2 lysmodeller. Der anvendes de værdier, som i følge ovenstående renderinger, bedst beskriver plastic - for figur 5.21, er der anvendt normal mapping med smoothstep-funktionen, og derfor en lagtykkelse på 0.048. Da de øvrige objekter ikke synes at have noget problem med moire-mønstre, er der blot anvendt step-funktionen, og derfor en lagtykkelse på 0.02.

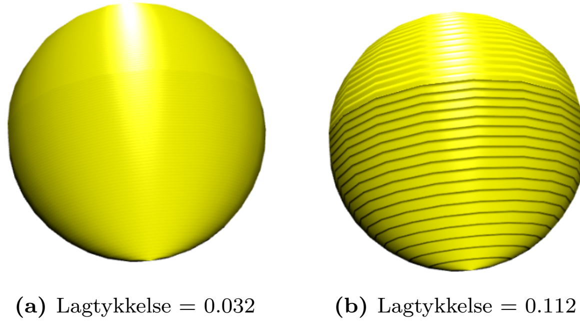


(a) Fotografi **(b)** Phong **(c)** Blinn-Phong

Figur 5.21: Kugle .

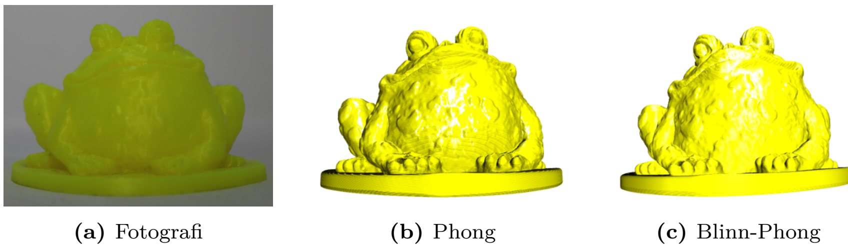
Sfæren er ikke specielt passende til denne procedurale model, hvor det blot er normalerne, og altså ikke geometrien der ændres. I nedenstående figur illustreres problemet

der opstår når man kigger på en sfære, med en vis vinkel fra vandret . Det er også illustreret med større lag i figur 5.22b.



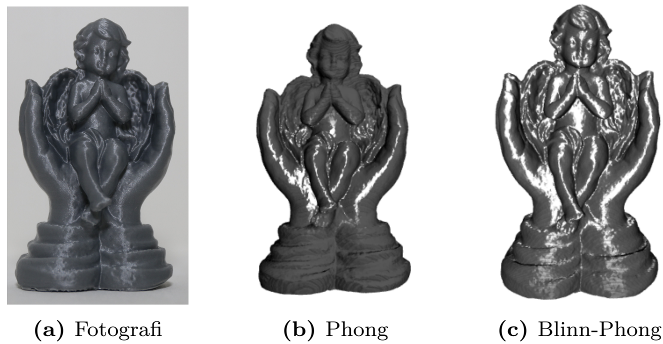
(a) Lagtykkelse = 0.032 **(b)** Lagtykkelse = 0.112

Figur 5.22: Kuglen, set fra en vinkel som ikke er vinkelret på fronten.



(a) Fotografi **(b)** Phong **(c)** Blinn-Phong

Figur 5.23: Frø.



(a) Fotografi **(b)** Phong **(c)** Blinn-Phong

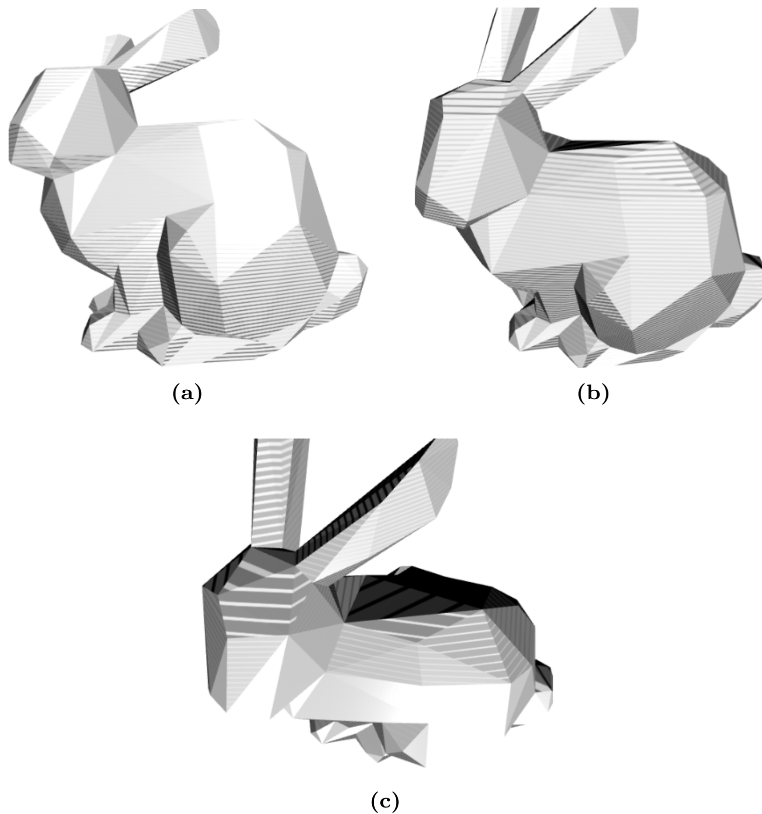
Figur 5.24: Engel .

Til sammenligning vises i figur 5.25, englen *shaded* med Phong modellen, men uden lag-inddelingen:

Figur 5.25: Englen *shaded* med Phong modellen, og uden trappe-effekten .

5.4.2 Interaktiv tilpasning

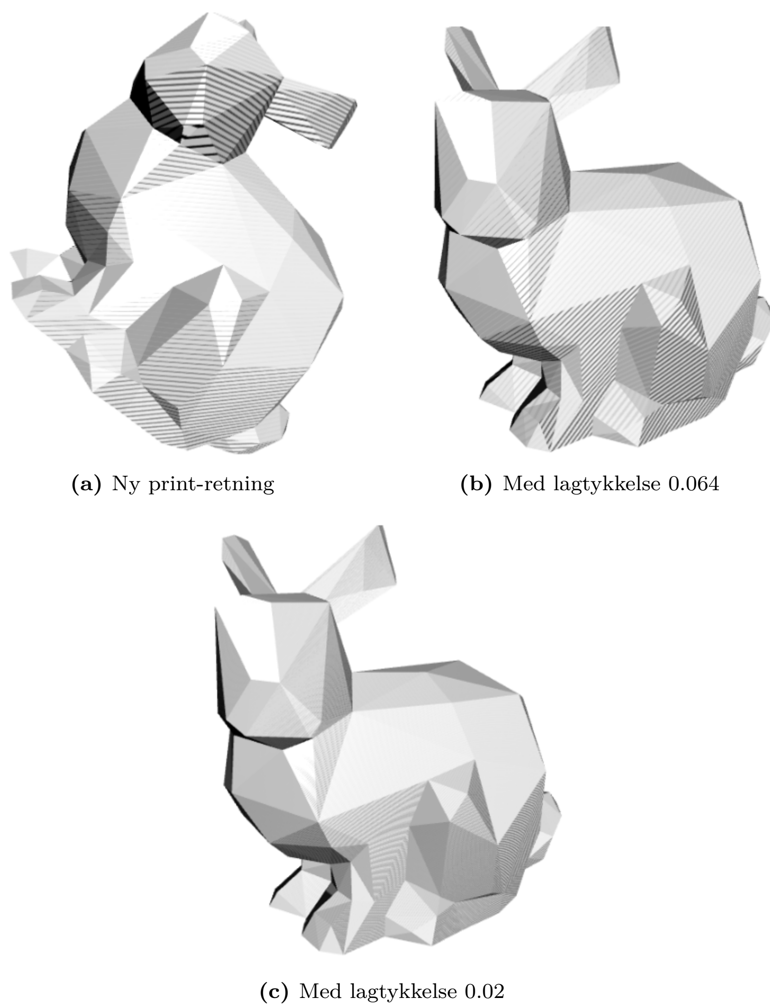
Dette afsnit vil præsentere renderinger, som gør brug af de implementerede interaktive navigationssystemer, og som er af den facetterede kanin. I figur 5.26 visualiseres kamera-navigation, da lag-inddelingens udseende (normalerne) er afhængig af kameraets placering. Der er anvendt en lidt større lag-tykkelse end ellers, for at tydeliggøre lagene, og der er blot anvendt normal mapping med en step-funktion.



(a) **(b)** **(c)**

Figur 5.26: Den facetterede kanin fra 3 forskellige kameravinkler, *shaded* med Blinn-Phong, og med en lagtykkelse på 0.064 .

Med objektnavigation er det muligt at rotere objektet, således at dette får en anden 'print-retning', da print-retningen er fastlås til *world space* y-aksen. I figur 5.27a visualiseres denne rotation med fastlåsning i y-aksen, mens figur 5.27b og c visualiserer brug af denne print-retning hvor kameraet er roteret således at det ligner at objektet er i sin oprindelige position.

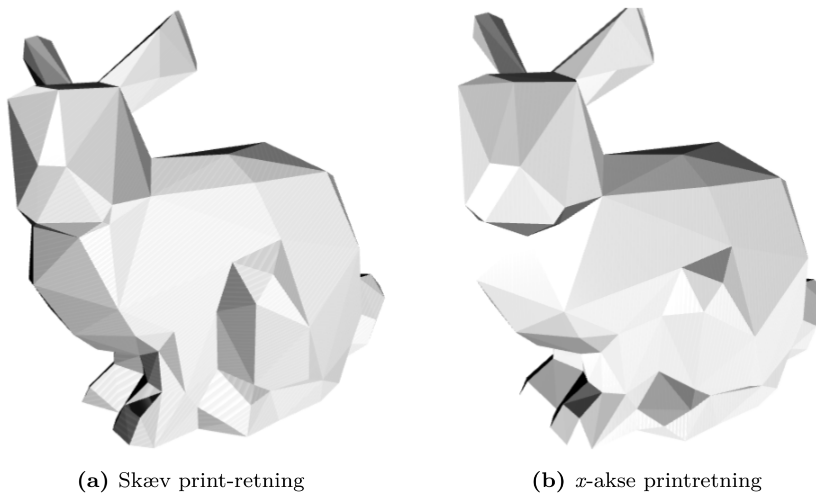


(a) Ny print-retning (b) Med lagtykkelse 0.064

(c) Med lagtykkelse 0.02

Figur 5.27: Ny print-retning. a) og b) har en lidt større lagtykkelse, for bedre at kunne se rotationen af lag.

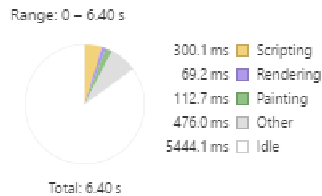
Endelig ses der i figur 5.28 2 renderinger hvor smoothstep-funktionen er anvendt til normal-mappingen. Den første har samme print-retning som i figur 5.27, i mens den anden har x-aksen som print-retning.



(a) Skæv print-retning **(b)** x-akse printretning

Figur 5.28: 2 renderinger med forskellige print-retninger. Der er anvendt smooth-step, og derfor en lagtykkelse på 0.048.

Det bør også her nævnes at det implementerede navigationssystem, samt den procedurale modellering, som ønsket kan renderes i realtid. I nedenstående figur vises en såkaldt *timeline recording*, som kan udføres direkte på websiden (Chrome), for at give et indtryk af hastigheden og hvad tiden bruges på. Det viste diagram beskriver hvad de 6.4s, hvor kameraet og objektet roteres, er blevet brugt på. Derudover kan det også aflæses på websiden at hastigheden er ∼45-55 fps. Det ser dog selvfølgelig lidt anderledes ud når man først renderer store mesh (englen (49998 *vertices*) og frøen (173077 *vertices*)).



Figur 5.29: *Time-line recording* for navigering af den kantede kanin.

KAPITEL 6
Diskussion

Dette afsnit vil diskutere resultaterne set i overstående afsnit, samt reflektere over de anvendte løsninger. Da casen om mursten og tråd, er bygget på et klassisk eksempel, er der ikke helt så meget at diskutere, og fokus vil derfor først og fremmest være på *anti-aliasing*, casen om lag-inddeling og den interaktive kamera- og objektnavigation.

6.1 Anti-aliasing

Der er blevet implementeret en *screen-space* baseret supersamling *anti-aliasing* algoritme, som kan anvendes til realtids renderinger, og altså derfor er passende til det implementerede interaktivitets navigationssystem. Som det ses af figur 5.2, kan algoritmen håndtere, de for raster-displays, velkendte savtakkede overgange, og give et tilfredsstillende resultat ved at anvende 32-50 *anti-aliasing samples*. Ved at reducere dette antal, reduceres render kvaliteten, mens renderingstiden øges.

Algoritmen opnår det bedste resultat ved at sample i et område på 2x2 pixels rundt om den original fragment position, hvilket kan ses i figur 5.3, da dette også mindsker mængden af moire-mønstre, i fht. kun at sample inden for et område på 1x1 pixels. Det ses dog i f.eks. figur 5.10 og 5.14, at moire-mønstre stadig er et problem, for tråd- og lag-modellen da der arbejdes med så fine strukturer. Moire mønstre ses også i virkeligheden, men oftest i mindre grad, da virkeligheden oftest indeholder uregelmæssigheder i regulære strukturer, som kan mindske *grid*-strukturen, hvilket også ses på fotografierne i afsnit 5.4.1. For at kunne reducere mængden af moire-mønstre endnu mere, ville det derfor være nødvendigt at bryde regulariteten, hvilket f.eks. kunne gøres ved at tilføje en *noise*-funktion til den procedurale model. For lag-modellen, ser vi dog i f.eks. figur 5.20, at det også er muligt at gøre dette ved at anvende bløde overgange mellem lagene, frem for den skarpe overgang der opnås ved anvendelsen af en step-funktion. Denne metode kan dog ikke umiddelbart overføres til tråd-modellen.

6.2 Mursten og tråd

Mursten
Det ses af resultaterne i afsnit 5.2, at det ved hjælp af klassisk *bump-mapping* er

muligt at modellere et specifikt udtryk, uden at ændre geometrien. Fokus har været på at modellere disse *bumps*, så det ligner at murstenene stikker frem i fht. mørtlen, hvor et tilfredstillende resultat f.eks. er opnået i figur 5.6. Med denne simple model ser det renderede udtryk, naturligvis ikke specielt fotorealistisk ud, da mursten i virkeligheden har uregelmæssige overflader. Derfor kunne tilføjelsen af en *noise*-funktion til den procedurale model være passende.

Tråd
Det er lykkedes at modellere en stofkugle, ved at modificere den klassiske murstensmodel. Som det ses af figur 5.9, kan man opnå et stof-lignende udtryk, blot ved at anvende en passende lysmodel. Hvor det altså ved at anvende Kajiya og Kaj's lysmodel, er lykkedes at opnå renderinger hvor highligtsene følger fiber-strukturen. Som nævnt viser denne model moire-mønstre, hvilket kunne mindskes ved at tilføje noget af den tilfældighed, som også ses i rigtigt stof. Derudover kunne man også vurdere at, modellere et flettet udtryk, ved ikke blot at sætte krydsningen mellem 2 tråde, til den horisontale tangent hver gang.

6.3 Lag-inddeling

Som det ses i afsnit 5.4, er det lykkedes at efterligne et laginddelt udtryk - altså en trappe-effekt - blot ved at ændre normalen. Dog ses det også i f.eks. figur 5.14, at denne illusion dør når lagtykkelsen bliver for stor. Da det har været hensigten at modellere det udtryk 3D-printede objekter har, er dette dog ikke et problem, da sådanne objekter er opbygget af relativt små lag. Hvis man derimod ønskede at modellere en trappe-effekt med større lag, ville det nok være nødvendigt rent faktisk at ændre geometrien, vha. *displacement mapping.*

I afsnit 5.4.1, forsøges det at ramme udseendet af forskellige fotograferede 3D-printede objekter, ved at variere variablene i de 2 meget anvendte lysmodeller: den modificerede Phong model og den modificerede Blinn-Phong model. Det må ud fra de renderede resultater konkluderes at Blinn-Phong klart kan beskrive udseendet af sådanne objekter bedst. Som det bl.a. ses i figur 5.14 og 5.21, resulterer Phong-modellen i de karakteristiske cirkulære highlights, som altså ikke ses på nogen af de fotograferede objekter. Derudover resulterer denne models brug af reflektionsvektoren, frem for halvvektoren, også i pludselige *cut-offs* i fht. det spekulære bidrag, hvilket især er et problem når der anvendes disse ekstreme *normal-mappede* normaler, som kan bemærkes ved at sammenligne figur 5.24b med figur 5.25.

Med Blinn-Phong modellen er det for den kantede kanin lykkedes at opnå et udtryk, som ligner det fotograferede udtryk relativt godt. Det bedst opnåede resultat ses i figur 5.20b, og ved at sammenligne med fotografiet i figur 5.13a, ses der altså de ønskede spekulære highlights på hovedet og bagenden, og lag-inddelingen kan skimtes der hvor der ikke er så meget spekulær reflektion. Dette udtryk opnås ved at anvende

bløde overgange i mellem lagene, frem for skarpe, da dette mindsker mængden af moire-mønstre drastisk. I fotografiet ses der et uregulært mønster under kaninens hoved, som skyldes at print-retningen er lodret, hvilket betyder at dette stykke printes ud i luften, og således skal nå at tørre. Et sådant udtryk kunne modelleres ved at tilføje en *noise*-funktion.

Da fokus har været på at ramme denne kantede kanins materiale-egenskaber, og så anvende den opstillede algoritme på andre modeller, er renderingerne af de andre modeller ikke helt så fotorealistiske. I sær bemærkes det at frøen og kuglen, synes at bestå af et materiale, det kan være svært at efterligne. Dog ses det af figur 5.24c, 5.23b og 5.23c, at lysmodellerne kan repræsentere de aflange highlights, der ses på laginddelte objekter. Phong-modellen rammer dog kun et lignende udtryk for frøen.

6.4 Interaktiv tilpasning

Implementeringen af den interaktive tilpasning, som gør det muligt at visualisere resultatet af at anvende en anden print-retning, fungerer tilfredsstillende som det ses i figur 5.27. Kamera-navigation og objekt-navigation synes hver for sig at fungere fint, men når disse kobles sammmen, således at man frit kan skifte mellem dem, som er tilfældet i denne implementering, kan navigationen hurtigt blive ulogisk. I appendiks A.1, findes et link til de forskellige måder det er forsøgt at tackle objekt-navigationen på, som alle hver for sig bliver ulogiske i deres navigation efter at have skiftet mellem navigation af kamera og objekt. Det er blevet vurderet at den navigationsmetode, der er mest passende til det givne formål - altså at rotere objektet indtil printretningen er som ønsket, og så rotere kameraet tilbage for at objektet synes at stå i sin default position - er den metode som er beskrevet i afsnit 4.6. Med denne metode risikerer man at rotationen af objektet bliver ulogisk, hvis man først roterer kameraet om på den modsatte side af objektet (hvilket man naturligvis også kan vurdere om brugeren ville have nogen grund til at gøre, da det er anvendt en simpel lysmodel, med en enkelt lyskilde, således at objektet blot er sort bagpå). Dog er det ikke mere ulogisk end at objektet roterer modsat af hvad man ønsker. Ved at anvende samme metode, men med en konjugeret rotationsmatrix, er det muligt at opnå det modsatte: altså objektet drejer som default modsat af hvad musen specificerer, men hvis man roterer kameraet om på den anden side bliver objektrotationen intuitiv. Endelig kan man vælge at udføre objektrotationen i *view space*, hvilket altid resulterer i intuitiv objektrotation, men til gengæld bliver kamerarotationen hurtigt ulogisk. Altså er det umiddelbart tale om en smagssag, hvor alle metoder har sine ulemper.

KAPITEL 7

Konklusion

I et forsøg på at proceduralt modellere regulære strukturer, er det i denne rapport blevet tydeliggjort hvorledes simple funktioner kan kombineres, og procedurale modeller kan modificeres, for at på denne måde opnå forskellige regulære strukturer. 3 forskellige strukturer er blevet modelleret, ved hjælp af kombinationen af simple funktioner, og klasiske procedurale metoder såsom *bump* og *normal mapping*. Derudover er det også blevet tydeliggjort at sådanne regulære strukturer, nemt kan komme til at virke *for* regulære, og at regulariteten bør brødes for at kunne opnå et fotorealistisk resultat.

En simpel procedural model, der kan beskrive den såkaldte trappe-effekt, vha. *normal mapping*, og altså uden faktisk at ændre geometrien er blevet opstillet. Denne model kan som ønsket efterligne 3D-printede objekters udseende, men hvis lag-tykkelsen bliver for stor dør illusionen. For at kunne repræsentere de regulære strukturer på et raster-display, er en såkaldt *anti-aliasing*-algoritme bliver implementeret, som kan håndtere *aliasing*-artifakter, såsom savtakkede farveovergange og moire-mønstre. Det er lykkedes at opnå et relativt fotorealistisk resultat, ved at *shade* modellerne vha. en modificeret Blinn-Phong lysmodel, og ved at anvende bløde overgange mellem lagene, for at bryde den føromtalte regularitet, som kan resultere i overdrevne moire-mønstre ved små lag-tykkelser.

En interaktiv brugerflade er også blevet implementeret, som gør det muligt for brugeren at styre lag-tykkelsen og at skifte mellem navigation af kameraet eller objektet. Når objektet navigeres, er lag-inddelingen låst fast til *world space* y-koordinaten, hvilket gør det muligt at rotere objektet og dermed visualisere brugen af en anden print-retning. Når der skiftes mellem de forskellige navigations-typer kan man risikere at rotationerne kommer til at virke ulogiske. Forskellige måder at udføre objektrotationerne på er blevet implementeret, men alle har deres ulemper, og der er nok mest tale om en smagssag, i fht. hvad man synes virker mest intuitivt.

KAPITEL 8

Referencer

Ahn, S. H. (2013): ’OpenGl Transformation’ [ONLINE], http://www.songho.ca/opengl/gl_transform.html (sidst besøgt d. 7/11-2016)

Akenine-Möller, T., Haines, E., Hoffman, N. (2008): ‘Real-Time Rendering’, A K Peter, Natick, Massachusetts, 3. udgave, side 72-77, 124-134, 217-222

Angel, E., Shreiner, D. (2014): ’Computer Graphics: A Top-Down Approach with WebGL’, 7th edition, Pearson, side 60-62, 361-363, 432-454

Anyurur, A. (2012): ’Professional WebGL Programming - Developing 3D graphics for the web’, John Wiley Sons, Ltd., side 1-19 Blinn, J. F. (1978): ’Simulation of Wrinked Surfaces’, SIGGRAPH’s *Procedings of the 5th annual conference on Computer graphics and interactive techniques*, sider 286-292

Bærentzen, A. J., Nobel-Jørgensen, M., Frisvad, J. R., Christensen, N. J. : ’Hands on real-time graphics’, Kursusnote for kursus i realtidsgrafik på Danmarks Tekniske Universitet, Institut for Matematik og Computer Science, sider 154-156

Ebert, D. S., Musgrave, F. K., Peachey, D., Perlin, K., Worley, S. (2003): ’Texturing and modeling; A procedural approach’, 3. udgave, Morgan Kaufman publishers, sider 25-45

Eiriksson, E. R., Pedersen, D. B., Frisvad, J. R., Skovmand, L., Heun, V., Maes, P., Aanæs, H. (2017): ’Augmented Reality Interface for Additive Manufacturing’, SCIA 2017

Frisvad, J. R., Frisvad, R. R. (2004): ’Real-time simulation of global illumination - using direct radiance mapping’, master thesis, Danmarks Tekniske Universitet, Institut for Matematik og Computer Science, sider 21-23, 172-178

Frisvad, J. R., Wyvill, G. (2007): ’Fast High-quality Noise’, GRAPHITE 07, *Proceedings of the 5th International Conference on Computer Graphics and Interactive Techniques in Australia and Southeast Asia*, sider 244-245

Gigahertz-Optik inc.: 'Calculation of radiometric quantities', [ONLINE], Tutorials, http://light-measurement.com/calculation-of-radiometric-quantities/ (sidst besøgt den 10/1-2017)

He, Y., Xue, G., Fu, J. (2014): 'Fabrication of low cost soft tissue prostheses with the desktop 3D printer', Nature Publishing Group, Scientific reports

Hewitt, W. T., Grave, M., Roch, M. (1991): 'Advances in Computer Graphics IV', 1. udgave, Springer Verlag, side 229

Hughes, J. F., Dam, A. V., McGuire, M., Sklar, D. F., Foley, J. D., Feiner, S. K., Akeley, K. (2014): 'Computer Graphics: Principles and practice', 3. udgave, Addison-Wesley, side 168-171

Khronos group: 'WebGL specification', [ONLINE] version 1.0.3., 27. oktober 2014, https://www.khronos.org/registry/webgl/specs/1.0/ (sidst besøgt den 16/1-2017)

Nielsen, J. B., Frisvad, J. R., Conradsen, K., Aanæs, H. (2014): 'Addressing grazing angle problems in Phong models', In ACM SIGGRAPH Asia 2014 Posters, Article 43

Knuth, D. E. (1969): 'The Art of Computer Programming', vol. 2 *Seminumerical algorithms*, 1. udgave, Addison-Wesley, side 9-21

McReynolds, T. og Blythe, D. (1997): 'Programming with OpenGL - Advanced Rendering', [ONLINE], SIGGRAPH 1997 kursus, afsnit 8.3, https://www.opengl.org/archives/resources/code/samples/advanced/advanced97/notes/ (sidst besøgt d. 8/1-2017)

OpenGL FAQ: 'Transparency, translucency and blending', [ONLINE], FAQ and troubleshooting guide, https://www.opengl.org/archives/resources/faq/technical/#indx0150 (sidst besøgt d. 9/1-2017)

Perea, H. (2012): 'Concept of Shading in Computer Graphics',E-bog, ISBN: 978-81-323-1718-0

Scratchapixel: 'Rasterization: a Practical Implementation' [ONLINE], tutorial webside, https://www.scratchapixel.com/lessons/3d-basic-rendering/rasterization-practical-implementation/ (sidst besøgt d. 1/1-2017)

Shoemake, K. (1985): 'Animating rotation with quaternion curves', SIGGRAPH's *Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, sider 245-254

Shontz, S. M., Nistor, D. M. (2012): 'CPU-GPU algorithms for triangular mesh sim-

plification', *Proceedings of the 21st International Meshing Roundtable*

BILAG A
Appendix

A.1 Kode

Koden der er blevet implementeret i forbindelse med dette projekt kan findes i den uploadede zip-fil, eller på følgende hjemmeside:

`http://www.student.dtu.dk/~s134729/bachelor/`

Der er en undermappe for hver case, og der vil her kort beskrives hvad hver fil indeholder. Hver mappe inderholder par af .html og .js-filer, som har samme navn, så herunder nævnes blot .html-filen.

Mappenavn: Bachelor_brick:
Filerne i denne mappe taler for sig selv, så her vil blot oplistes navnene:

- 2D_brick.html
- 2D_brick_bump.html
- 3D_brick.html
- 3D_brick_bump.html

Mappenavn: Bachelor_fabric:
Denne mappe indeholder kun en enkelt fil, hvor en stofkugle er modelleret, og kameraet kan navigeres: fabric_track.html

Mappenavn: Bachelor_layers:
I fht. normal-mapping og de forskellige lysmodeller er der implementeret forskellige versioner, som her forklares. Alle implementeringer indeholder kamera/objekt-navigation som forklaret i afsnit 3.6.2 og 3.6.3.

- track_object_and_cam_phongblinn.html - Blinn-Phongs modificerede lysmodel + normal mapping med smoothstep
- track_object_and_cam_phong.html - Phongs modificerede lysmodel + normal mapping med step

- track_object_and_cam_phong_many_vertices.html - Blinn-Phongs modificerede lysmodel + normal mapping med step + tilpasset til trekantsmesh med mange *vertices* (garden_toad.obj og angel_1.obj)

I fht. kamera og objektnavigation er der implementeret forskellige versioner, som her vil forklares. De indeholder alle kameranavigation som forklaret i afsnit 3.6.2.

- track_object_and_cam1.html - objekt navigation som forklaret i afsnit 3.6.3
- konjugeret.html - som ovenstående men med konjugeret rotationsmatrix
- view.html - objekt navigation i *view space* som forklaret i [Frisvad, 2004], men med ikke-konjugeret rotationsmatrix

Mappenavn: Common:
Denne mappe indeholder forskellige Javascripts biblioteker. De normalt anvendte vil ikke nævnes her, men der er blandt andet en for kvaternion-operationer og for indlæsning af .obj-filer (en tilpasset store trekantsmesh, og en tilpasset således at et facetteret udtryk kan opnås).

A.2 Projektmotivation og projektplan

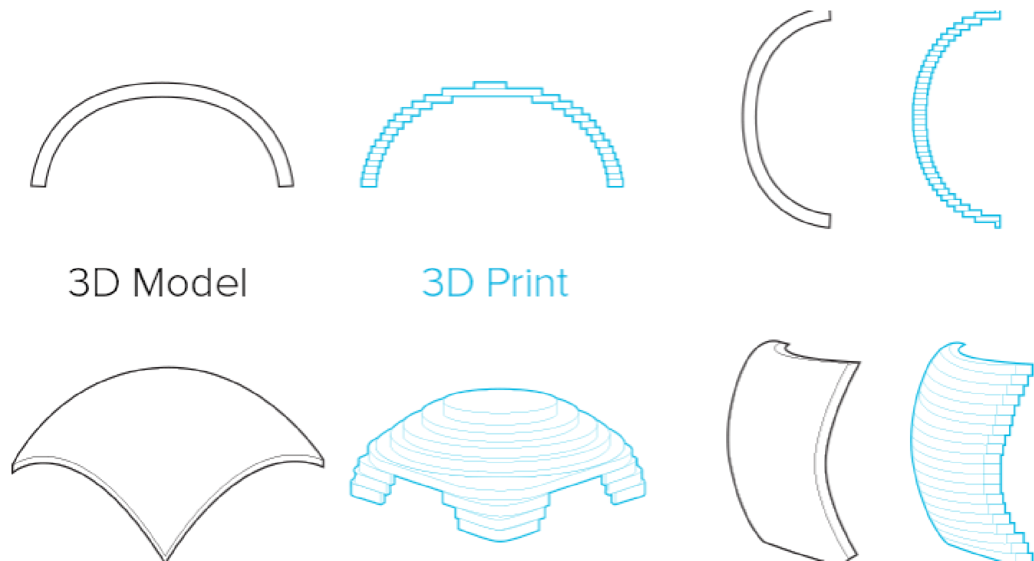
I starten af projektet, blev nedenstående projektmotivation og projektplan udarbejdet. Generelt er projektplanen blevet fuldt rimelig godt, bortset fra at det blot blev valgt at arbejde med real-tid, som det også er forklaret er ønskeligt i projektmotivationen. Derudover er en stor del af tiden også blevet brugt på at implementere interaktiv visualisering (trin 6), så der har ikke været tid til eksempelvis at tilføje støj.

Projektmotivation

3D-printning er en teknologi der har været i udvikling i rigtig mange år, men efter årtusindskiftet kom et boom indenfor teknologien, der bl.a. banede vejen for desktop printere. Desktop printere har gjort det muligt for almindelige forbrugere at 3D-printe, men dog ikke med teknologier på helt samme niveau som der opleves i industriprintere, hvor de mest præcise printere kan printe med lag tyndere end menneskehår.

Som forbruger vil man ofte opleve at resultatet af 3D-printningen og den digitale 3D-model ikke stemmer overens. Dette skyldes at 3D-printere opbygger modeller lag for lag, hvor almindelige forbrugerprintere gerne anvender lag på omkring 0.1 til 0.3 mm. Dette resulterer i en trappe-lignende effekt på kurvede eller hældende overflader, hvor det vil være tydeligst ved små hældninger da afstanden mellem lagene vil være større. Dog vil orienteringen der anvendes ved printningen have en betydning for graden af denne trappeeffekt.

Derfor kunne det være rigtig interessant at opstille et framework, der forsøger at generere en realistisk model af hvordan resultatet af en sådan printning vil se ud, givet en vis digital model. Da der er tale om en regulær struktur kan vi modellere sådan et udseende vha. procedurale teknikker, hvor der anvendes matematiske funktioner og algoritmer, til at beskrive den ønskede overflade-struktur. Desuden kunne det også være ønskeligt at denne model kunne renderes i realtid, således at forbrugeren, før printningen, kan vurdere hvordan resultatet vil blive ved forskellige printnings-orienteringer.



Figur 1: Trappeeffekt ved forskellige printningsorienteringer

Projektplan - Procedural Modellering af Regulære Strukturer

Jobs:

1. Udarbejdelse af projektplan og projektmotivation
2. Litteratursøgning og læsning af baggrundsteori
3. Første eksempel (bump-mapping)
4. Overfladestruktur i lag-retningen
5. Real-tid vs. Offline (valg af renderingsteknik)
6. Tilpas til platform og generaliser/forbedr algoritmen – bl.a. ved at tilføje irregularitet/støj
7. Eksempler/anvend cases/udforsk
8. Færdiggør rapport

Tidsplan:

Job/Uge	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1.																
2.																
3.																
4.																
5.																
6.																
7.																
8.																