Bachelorrapport: Grafiske skygger

Fulde navn: Ulrik Søgaard
KU-nummerplade: lfd450

Fulde navn: Tor Heberg Justesen
KU-nummerplade: lwj858

3. juni, 2013

Abstract

Shadows is an important part of modern computer graphics and many different methods to generate shadows exist. We will in this report work with two of the best known algorithms to generate shadows: the shadow volume algorithm from 1977 [Cro77] and the shadow mapping algorithm from 1978 [Wil78].

In this report we will describe the theory behind these two shadow algorithms and our own implementation based on the algorithms. We will study the two algorithms and describe the pros and cons of each algorithm in the theory part of the report. In addition, we will explain the problems of the algorithms and the possible solutions to said problems.

The report rely on already existing knowledge and published articles in the field. It is beyond the goal of this report to broaden the knowledge and information already present in the field.

Based upon the research we have made we can conclude that neither of the algorithms is better than the other, since both of them has their pros and cons.

Resume

Skygger er en vigtig del af nutidens computergrafik, og der findes derfor forskellige metoder til at generere skygger på. Vi har i denne rapport valgt, at beskæftige os med, to af de mest kendte algoritmer til skyggegenerering. Disse algoritmer er: skyggevolumen (engelsk: "shadow volume") algoritmen fra 1977 [Cro77] samt shadow map algoritmen fra 1978 [Wil78].

Vi vil i denne rapport beskrive teorien bag disse to skyggealgoritmer samt beskrive vores egen implementation af algoritmerne. I teori-delen vil vi undersøge de to algoritmer og beskrive deres fordele og ulemper, desuden vil vi forklare problemer med algoritmerne samt eventuelle løsninger til disse.

Rapporten tager udgangspunkt i allerede eksisterende viden og udgivede artikler indenfor området. Det er udenfor forfatternes mål at udvide den viden og information der allerede findes indenfor området.

Ud fra de undersøgelser der er blevet fortaget i rapporten, er vi kommet frem til at både skyggevolumen algoritmen og shadow mapping algoritmen har sine fordele og ulemper, og at man ikke kan sige at den ene algoritme er bedre end den anden.

Indholdsfortegnelse

1 Introduktion

I dette afsnit vil vi give læseren et overblik over hvad rapporten og projektet indeholder. Desuden vil afsnittet også indeholde de forudsætninger vi forventer af læseren for at kunne forstå projektet samt rapporten. Til sidst i dette afsnit vil vi også komme ind på motivationen for projektet.

I rapporten vil vi gennemgå to algoritmer til at lave skygger, skyggevolumen og shadowmapping. I første del vil vi gennemgå teorien bag algoritmerne og i anden del vil vi forklare vores implementation af de to algoritmer.

I kapitel 2 vil vi gennemgå teorien bag skyggevolumen algoritmen. I denne forbindelse vil vi komme ind på stencil bufferen og silhuetter. Desuden vil vi gennemgå to metoder til at lave skyggerne med skyggevolumen algoritmen, som hedder Z-pass og Z-fail. I kapitel 3 gennemgår vi shadowmapping algoritmen, hvor vi blandt andet kommer ind på percentage closer filtering (PCF) samt moirés pattern og peter panning.
I begge afsnit vil vi desuden også komme ind på bløde skygger med de givne algoritmer. Til sidst i teori-delen vil vi også komme ind på fordele og ulemper ved de to algoritmer.

I kapitel 5 vil vi beskrive hvordan vi har implementeret skyggevolumen algoritmen i vores program. Ligeledes vil vi i kapitel 6 beskrive vores implementation af shadow mapping algoritmen. I kapitel 8 vil vi gennemgå nogle af de forbedringsforslag vi har til vores program. Til sidst, i kapitel 9, vil vi gennemgå de konklusioner vi har draget på baggrund af projektet.

1.1 Krav til læseren

Vi forventer at læseren har kendskab til de basale begreber inden for computergrafik. Dette inkluderer begreber såsom ambient, diffus og reflekterende (specular) lys, dybde test, vertex shader, fragment shader og hvad culling er. Desuden antager vi at læseren ved hvad en bit-maske er samt hvad et koordinat system er og kender til forskellen mellem hårde og bløde skygge.

1.2 Motivation

Vores motivation til at vælge *grafiske skygger* er at vi synes det er et spændende emne, men også et meget relevant emne at beskæftige sig med. Selvom det at lave realistiske skygger er en gammel teknik, er det kun blevet mere relevant i takt med at moderne grafik hardware og dermed også computergrafik bliver mere og mere avanceret og veludviklet. Udviklingen indenfor computergrafik har gjort at man gerne vil kunne lave realistiske skygger hurtigt og effektivt.

Den store relevans indenfor realistiske skygger er ikke den eneste grund til at vi valgte dette emne. Vi var begge meget begejstret for kurset *Introduktion til computergrafik* og blev enige om at skrive bachelor omkring et emne inden for computergrafik.

I Teori

2 Skyggevolumener

I dette afsnit vil vi forklare teorien omkring skyggevolumen (eng: "shadow volume") algoritmen. Vi vil komme ind på hvad stencil bufferen er, samt selve algoritmen. Vi kommer desuden ind på om hvad silhuetter er og hvorfor og hvordan de bliver lavet. Til sidst vil vi desuden kort beskrive hvordan man kan lave bløde skygger med skyggevolumen algoritmen.

2.1 Stencil buffer

Før vi går i gang, med at forklare hvordan man laver skygger ved brug af skyggevolumener, skal man først have en ide om hvad stencil bufferen er, da dette er en essentiel del af algoritmen. Stencil bufferen er en buffer, som indeholder information, i form af et heltal, om hver pixel.
Når man gør brug af stencil bufferen laver man en test for hvert punkt der tegnes for at se om den opfylder visse kriterier. Dette bliver gjort på grafikkortet, og det er også her selve bufferen ligger. Når man laver en stencil-test, betyder det at man tæller værdien for pixels på skærmen op eller ned, hvis de punkter der skulle tegnes i den givne pixel opfylder en eller flere *krav*. De krav der bliver tjekket er: stencil-test, Z-fail, Z-pass. Hvad disse bliver brugt til kommer vi ind på senere.

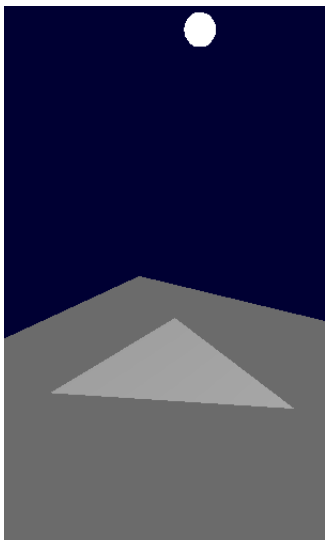
Bufferen bliver hovedsagligt brugt til at lave en bit-maske, der bestemmer hvilke pixels på skærmen der må tegnes på. Det er også på denne måde skyggevolumener bruger bufferen.

2.2 Algoritmen

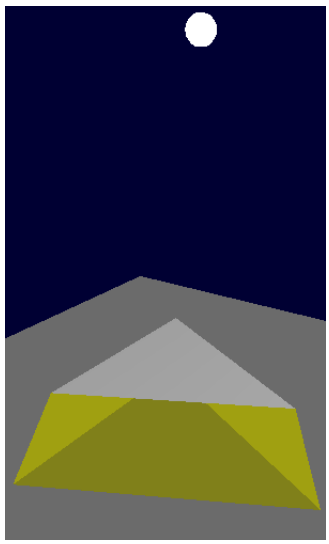
I 1977 fremviste Crow [Cro77] en algoritme til at generere hårde skygger ved brug af stencil bufferen. Denne algoritme kaldes for skyggevolumen (fra engelsk "Shadow Volume") algoritmen. Selve algoritmen fungerer kort sagt ved at man, for hver polygon, udregner et skyggevolumen. De punkter, der ligger inden for disse volumener, vil være i skygge.

Algoritmen fungere ved at man første og fremmest tegner ens scene normalt. Når scenen bliver tegnet bliver der fyldt information ind i dybde bufferen, som er en essentiel for algoritmen.
Når man skal lave selve skyggerne gør man brug af stencil-bufferen. Man laver en stencil-test for at finde ud af hvor mange skyggevolumener et givent punkt ligger i. Dette gør man basalt set ved at tælle hvor mange forsider og bagsider af skyggevolumener et punkt ligger bag. Hvis punktet ligger bag lige mange forsider og bagsider betyder det at punktet ikke ligger i skygge.

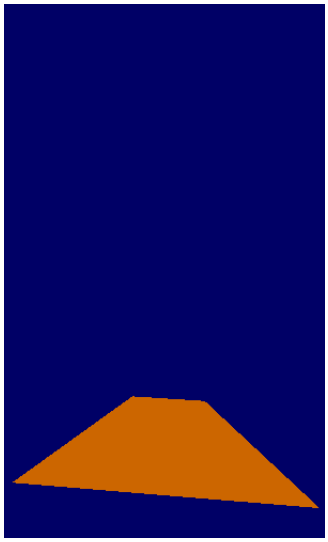
Man bruger somsagt stencil-testen til at tælle hvor mange volumener et punkt ligger i. Her gør man brug af dybde testen. En dybde test sammenligner dybden på et givent punkt, med dybden for samme punkt der ligger i dybde bufferen. Hvis det første punkt ligger tættere på kameraet end det punkt der ligger i dybdebufferen, klarer den dybde testen. Dette er en meget vigtig del af algoritmen, da det er på den måde man finder ud af hvor mange forsider og bagsider af skyggevolumener et punkt ligger bag. Resultatet af stencil-testen (det vil sige hvor mange volumener et punkt ligger i) bliver gemt i stencil bufferen. Når man er færdig med testen tegner man skyggerne. Dette gøres ved at man tegner en semi-transperant polygon over skærmen, i mens man bruger stencil bufferen. Det vil sige, at kun i de punkter i stencil bufferen hvor værdierne er over 0, vil vores semi-transparente polygon bliver tegnet.



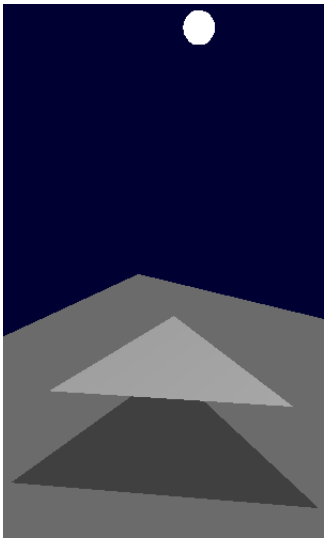
(a) *Dette er vores simple test scene uden skygger.*



(b) *Her kan man se vores skyggevolumen (den gule figur), som er blevet udregnet ud fra lysets position, alt der ligger inden for dette volumen skal ligge i skygge.*



(c) *Her er en visualisering af stencil bufferen for denne scene. De pixels på billedet der er blå, er de pixel hvor stencil bufferens værdi er 0, den orange del er pixels hvor stencil bufferen er over 0 (det vil sige vores skygger).*



(d) *Her er vores test scene med skygger. Bemærk at skyggerne på dette billede stemmer overens med indholdet af stencil bufferen, se 2.1c.*

Figur 2.1: *Disse figurer forklarer hvordan der oprettes skygger ved brug af skyggevolumen algoritmen.*

Ovenstående giver et godt billede af hvordan stencil-bufferen ændrer sig når der laves skygger.

Første billede viser vores simple scene uden skygger. Det næsten billede viser et eksempel på et skyggevolumen, der er udregnet ud fra polygonen på billedet. Det er imens man tegner dette volumen at stencil-testen bliver udført. Det skal siges at man under normale omstændigheder ikke får disse volumener at se.

Når man har lavet stencil testen, vil der ligge en bit-maske i stencil-bufferen. Hvis man tager udgangspunkt i 2.1c vil man som sagt kun have lov til at tegne på den del der er farvet orange. Når man så skal lave skyggerne tegner man som nævnt før en halv-gennemsigtig (alt efter hvor kraftig skyggerne skal være) polygon der dækker hele skærmen, imens man gør brug af bit-masken i stencil bufferen. Dette vil resultere i hvad der ses 2.1d.

Det er vigtigt at slå fast at der ikke bliver tegnet noget synligt på skærmen under selve stencil testen. Der bliver heller ikke skrevet til dybde bufferen under selve stencil-testen, da man ikke skal gemme resultatet af dybde testen der bliver lavet, kun resultatet af stencil testen er vigtigt. Der er to forskellige måder at implementere skyggevolumen algoritmen på. Disse to metoder gør henholdsvis brug af Z-fail eller Z-pass til at bestemme hvornår stencil bufferen tælles op og ned.

2.2.1 Z-pass

Ideen med Z-pass (også kaldet "depth pass") er at tælle værdien i stencil bufferen op når en given pixel består dybde testen. Konstruktionen af stencil-bufferen sker ud fra følgende fremgangsmåde:

1 Skrivning til dybde- og farvebufferen skal deaktiveres, så der ikke bliver skrevet til dybde bufferen eller tegnet noget på skærmen.

2 Brug back-face culling (så det kun er forsiden af polygonerne der bliver tegnet).

  2.1 Tegn skyggevolumenet (da der bruges back-face culling er det kun forsiden af skyggevolumenet der tegnes).

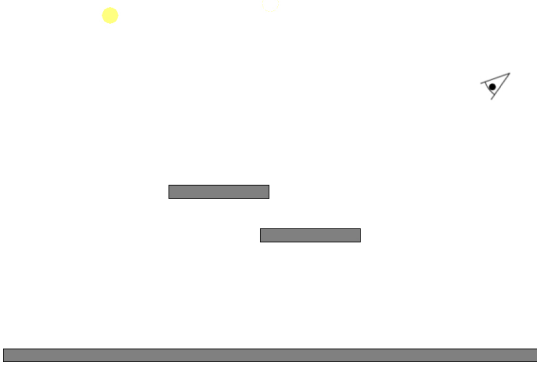
  2.2 Inkrementer værdien i stencil-bufferen når dybdetesten passerer.

3 Brug derefter front-face culling (så kun bagsiden af polygonerne bliver tegnet).

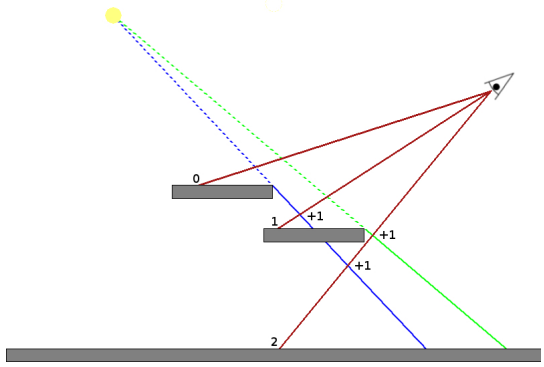
  3.1 Tegn skyggevolumenet (da der bruges front-face culling er det kun bagsiden som tegnes).

  3.2 Dekrementer værdien i stencil-bufferen når dybdetesten passerer.

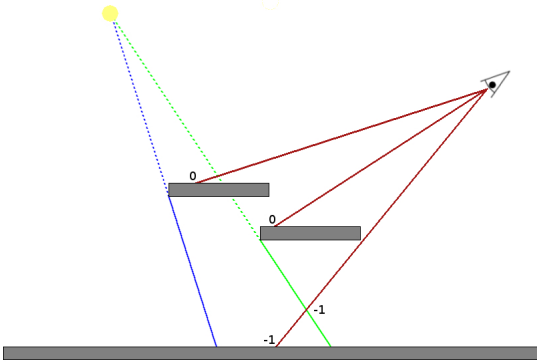
Denne fremgangsmåde skal udføres for hver lyskilde der er i den givne scene. En illustration af denne fremgangsmåde er vist på figur 2.2. Hvis værdien i stencil-bufferen er 0, ligger den givne pixel ikke inden for et skyggevolumen og er derfor ikke i skygge. Hvis værdien derimod er over 0, betyder dette at pixlen ligger i skygge.



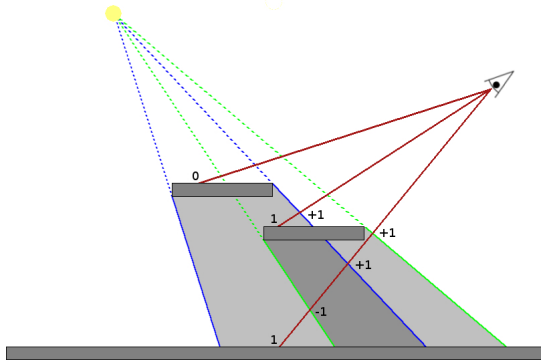
(a) *Denne figur viser scenen uden nogen skygger.*



(b) *Denne figur viser den første gennemkørsel af Z-pass algoritmen, hvor der bruges back-face culling. For hver forside af skyggevolumen linjen går ind i tælles værdien 1 op. Bemærk at det kun er de fuldt optrukne streger der angiver et skyggevolumen.*



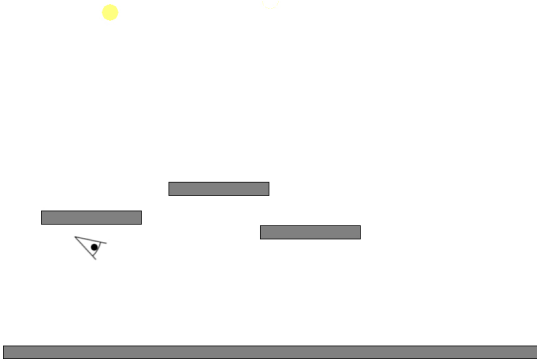
(c) *Denne figur viser anden gennemkørsel af Z-pass algoritmen, hvor der bruges front-face culling. Hver bagside af skyggevolumen linjen går igennem tæller værdien 1 ned.*



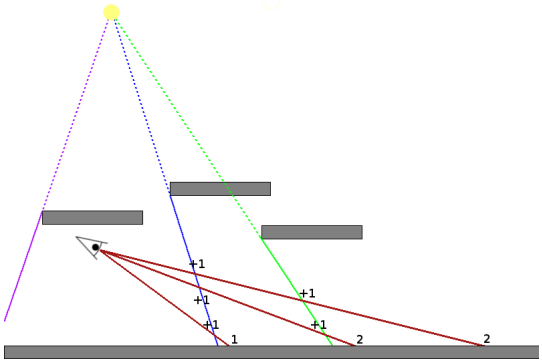
(d) *Denne figur viser det færdige resultat af Z-pass algoritmen.*

Figur 2.2: *Denne figur illustrerer hvordan Z-pass algoritmen fungerer.*

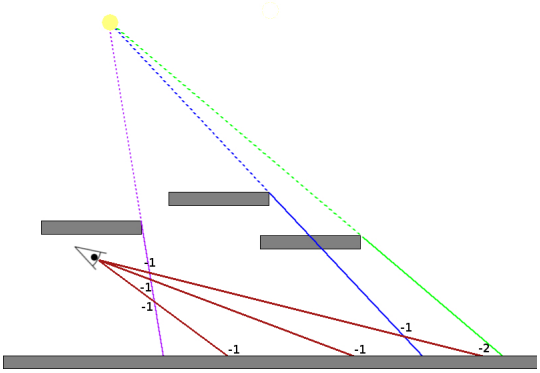
Med denne her fremgangsmetode opstår der dog et stort problem, hvis kameraet er inden i en eller flere skyggevolumener. Da kameraet er inden i en skygge vil stencil bufferen blive fyldt op me ukorrekte værdier. For at løse dette problem, kan man udregne antallet af skyggevolumener, kameraet er inde i og dermed sætte startværdien i stencil-bufferen til dette antal. Denne situation kan ses på figur 2.3. Dette kræver en del udregninger og tager heller ikke højde for situationen, hvor front-clipping planet er delvist indeni et eller flere skyggevolumener.



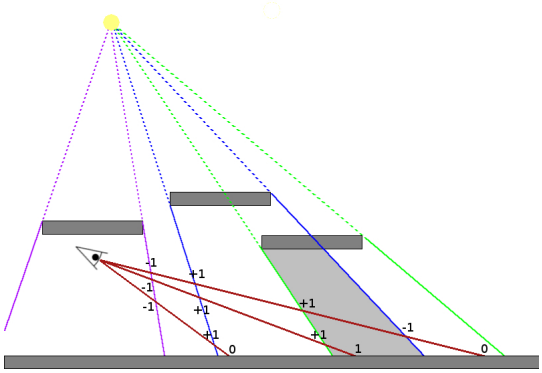
(a) *Denne figur viser scene uden skygger*



(b) *Denne figur viser den første gennemkørsel af Z-pass algoritmen, hvor der bruges back-face culling. For hver forside af skyggevolumen linjen går ind i tælles værdien 1 op.*



(c) *Denne figur viser anden gennemkørsel af Z-pass algoritmen, hor der bruges front-face culling. Hver bagside af skyggevolumen linjen går igennem tæller værdien 1 ned.*



(d) *Denne figur viser det færdige resultat af Z-pass algoritmen, hvor man tydeligt kan se at skyggerne ikke er korrekte.*

Figur 2.3: *Denne figur illustrerer hvorfor Z-pass algoritmen fejler når øjet er inden i et skyggevolumen.*

2.2.2 Z-fail

I Z-pass metoden talte man skygger foran et givent objekt. I Z-fail (også kaldet "depth fail") tæller man skyggerne bag ved et objekt. Selve fremgangsmåden for Z-fail er følgende:

1 Skrivning til dybde- og farvebufferen skal deaktiveres, så der ikke bliver skrevet til dybde bufferen eller tegnet noget på skærmen.

2 Brug front-face culling (så det kun er bagsiden af polygonerne der bliver tegnet).

  2.1 Tegn skyggevolumenet (da der bruges front-face culling er det kun bagsiden af skyggevolumenet der tegnes).

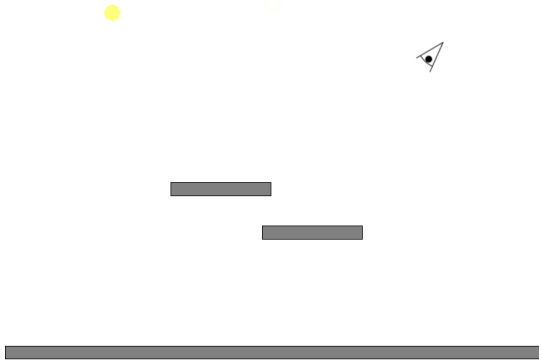
  2.2 Inkrementer værdien i stencil-bufferen når dybdetesten fejler.

3 Brug derefter back-face culling (så kun forsiden af polygonerne bliver tegnet).

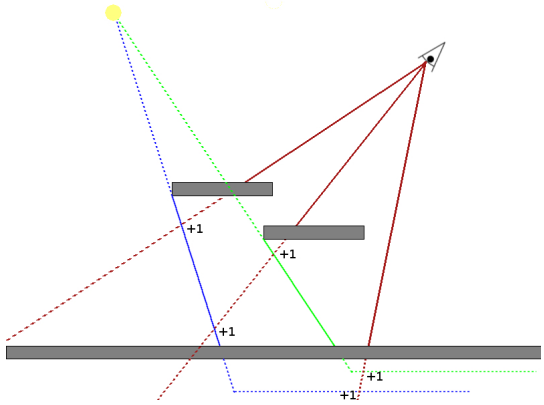
  3.1 Tegn skyggevolumenet (da der bruges back-face culling er det kun forsiden som tegnes).

  3.2 Dekrementer værdien i stencil-bufferen når dybdetesten fejler.

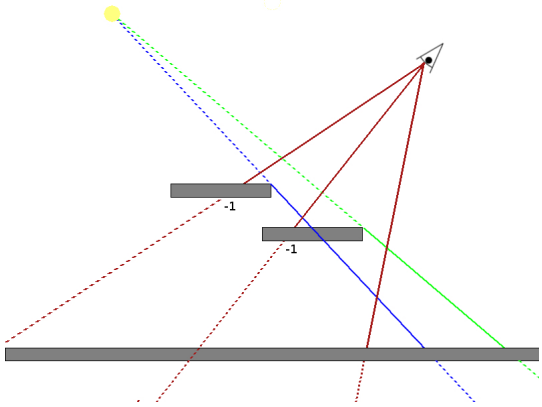
En illustration af denne fremgangsmetode er vist på figur 2.4. Denne her fremgangsmåde kræver dog at skyggevolumenerne er afgrænsede, så man bliver nød til at beskærer bagenden samt toppen af skyggevolumenerne. På figur 2.5 kan man se hvordan Z-fail smukt klarer situationen hvor øjet er inden i skyggevolumen.



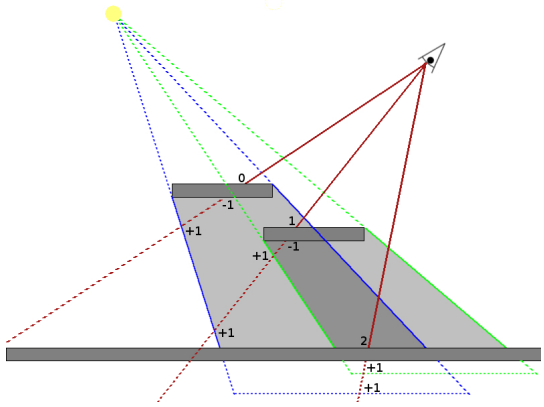
(a) *Denne figur viser hvordan scenen ser ud uden skygger.*



(b) *Denne figur viser første gennemkørsel af Z-fail, hvor der bruges front-face culling. Bemærk at volumenerne skal være aflukket, så derfor vil der være 2 gange +1 fra linjen helt ude til højre. 1 gang for den grønne volumen og 1 gang for bunden af den blå volumen (som linjen vil skærer uden for billedet).*

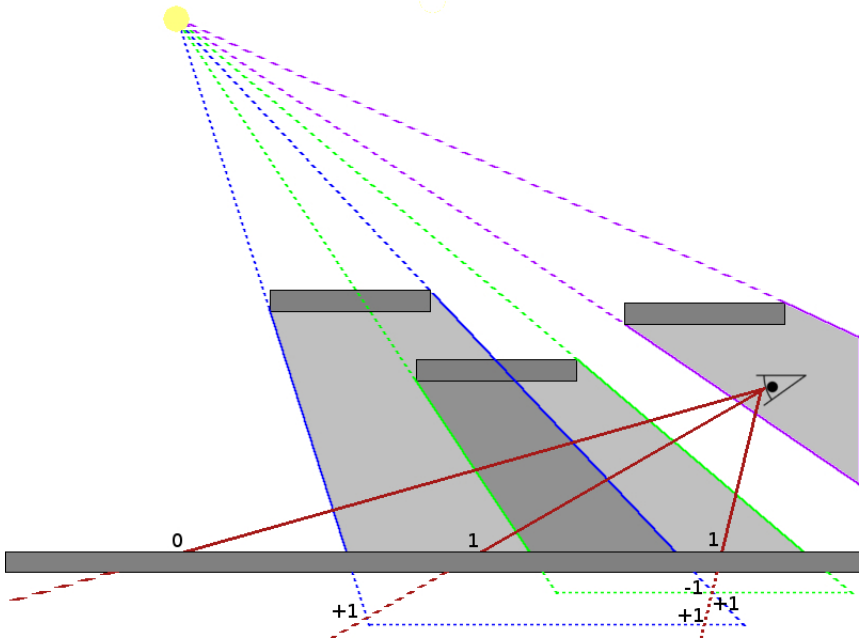


(c) *Denne figur viser anden gennemkørsel af Z-fail, hvor der bruges back-face culling. Bemærk igen at volumenerne skal være aflukket, så toppen af volumenerne vil dekrementere værdien.*



(d) *Denne figur viser det færdige resultat af Z-fail algoritmen.*

Figur 2.4: *Disse figurer viser hvordan Z-fail algoritmen fungerer.*



Figur 2.5: *Her kan man se hvordan Z-fail, helt automatisk, klarer situationen hvor kameraet er indeni et skyggevolumen.*

Det er vigtigt at bemærke at man har afgrænset volumener både i toppen og i bunden.

2.2.3 Sammenligning af Z-pass og Z-fail

Z-pass
Fordele:

- Behøver ikke aflukkede skyggevolumener.
- Mindre der skal tegnes.
- Hurtigere end Z-fail.
- Lettere at implementere end Z-fail, hvis man ser bort fra front-clipping planet.
- Behøver ikke uendelig perspektiv projektion.

Ulemper:

- Ikke en robust løsning på grund af det problem der opstår når kameraet er indeni i et skyggevolumen.

Z-fail
Fordele:

- Robust løsning, da der ikke er noget problem når kameraet er indeni i et skyggevolumen.

Ulemper:

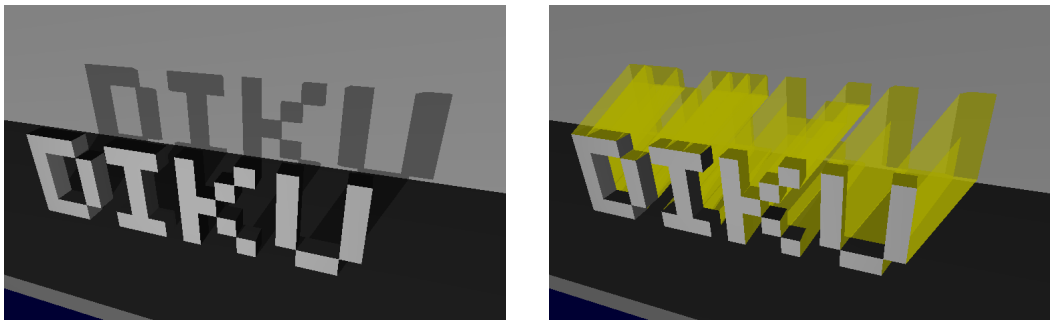
- Kræver aflukkede skyggevolumener.
- Der skal tegnes mere eftersom skyggevolumenerne skal være aflukket.
- Langsommere end Z-pass.
- Sværere at implementerer end Z-pass.
- Uendelige perspektiv projektion er påkrævet, hvis man vil have uendelig lange skygger.

Ud fra dette lader det til at Z-pass er den bedste algoritme, desværre har den dog en meget stor ulempe. Nemlig hvis kameraet er indeni skyggevolumenet, eller hvis kameraet er for tæt på skyggevolumen så front-clipping planet kommer i vejen. Af denne grund bruges ofte Z-fail, da det er en robust løsning, med mindre man kan sikre sig at Z-pass ikke giver problemer (ved at begrænse kameraet)[YK02].

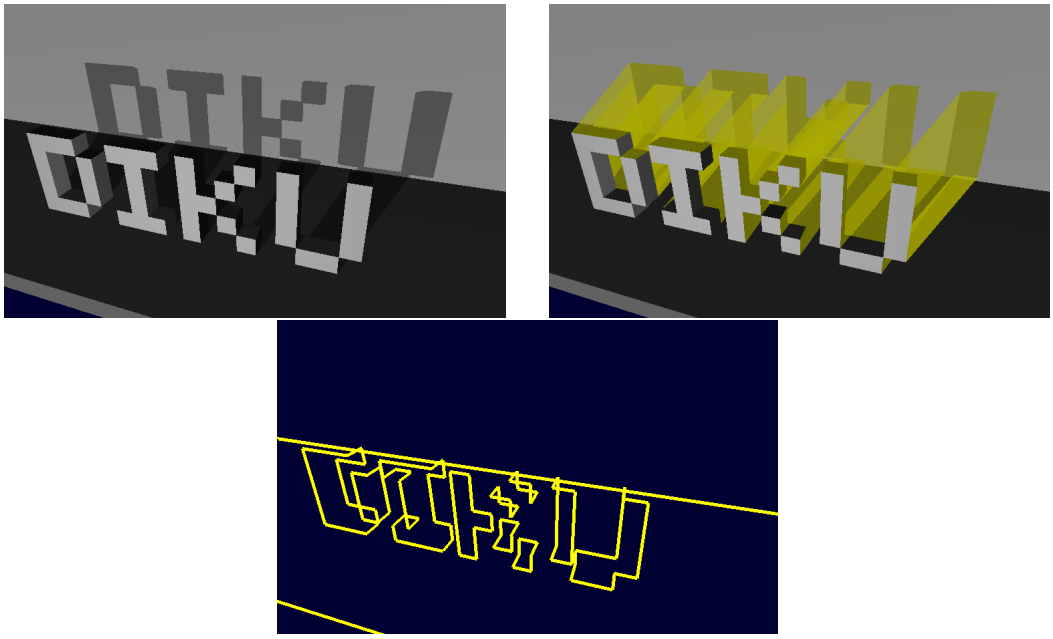
2.3 Silhuetter

Udregningen og renderingen af skyggevolumener kan hurtigt blive meget dyrt (beregningsmæssigt) hvis man laver et volumen per polygon. Derfor gør man brug af en metode til, at danne silhuetter for figurer i en scene.

En silhuet er, som navnet antyder, det som danner skyggen til en given figur. Hvis man ikke gør brug af silhuetter skal der laves skyggevolumener for alle polygoner. Dette betyder at både grafikkortet og cpu'en skal arbejde meget mere end nødvendigt.



Figur 2.6: *I denne scene bliver der ikke brugt silhuetter til at tegne skyggerne. Bemærk det store antal af volumener der bliver brugt for at lave skyggerne.*



Figur 2.7: *Dette er samme scene som før, men her bliver der brugt silhuetter. Som man kan se, er selve skyggerne stort set ens. Ser man derimod på antallet af volumener der bliver brugt, kan man tydelig se en ændring. I det først billede bliver der i alt tegnet 24 volumener, i mens der kun bliver tegnet 11 på andet billede. Bemærk især denne forskel ved D'et og I'et. Det sidste billede viser de silhuetter der bliver genereret ud fra scenen.*

På henholdsvis figur 2.6 og 2.7 kan man se forskellen mellem en scene med og uden silhuetter.

Når man laver en silhuet skal man blot finde ud af om en kant på en polygon, skal bruges til at danne en skyggevolumen. Måden man finder ud af dette er forholdsvis simpel. Dette gør man ud fra følgende fremgangsmetode:

1 Find alle nabo-polygoner til alle polygoner.

2 Find ud af hvilke polygoner der er synlige fra lyskilden.

3 Gennemgå alle polygonerne i figuren.

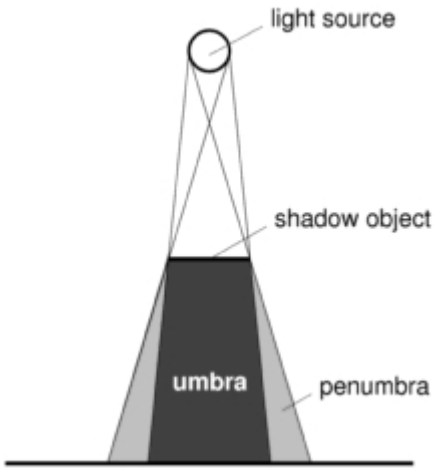
- Hvis en kant til en polygon har en **synlig** nabo, skal den givne kant ikke danne en del af silhuetten.
- Hvis en kant til en polygon ikke har en **synlig** nabo, betyder det at den danner en side af vores silhuet og dermed en side af skyggevolumen.

Man skal dog huske at alle skyggevolumerne skal være aflukket hvis man bruger Z-fail metoden. Det vil sige at alle synlige polygoner stadig skal danne en top og bund på skyggevolumerne.

2.4 Bløde skygger med skyggevolumener

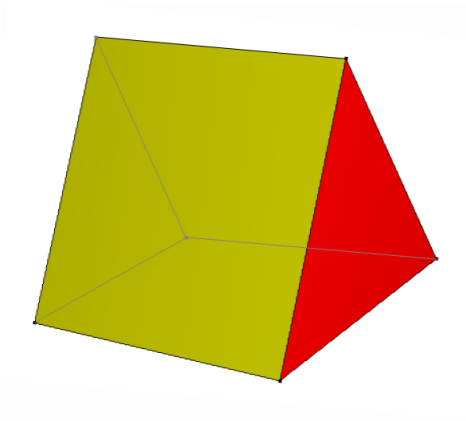
Igennem dette afsnit har vi indil nu beskrevet hvordan man laver hårde skygger med skyggevolumen algoritmen. I denne sektion vil vi kort komme ind på hvordan man kan lave bløde skygger

med skyggevolumen algoritmen. En af de metoder man kan bruge til at generere bløde skygger er penumbra wedges [AMA02]. Denne metode kræver, at man har en lyskilde, som ikke kun er et enkelt punkt. Her kunne man for eksempel bruge et kugleformet objekt som lyskilde. Dette er illustreret på figur 2.8, hvor der er en kugleformet lyskilde, så der dannes både en umbra og en penumbra.

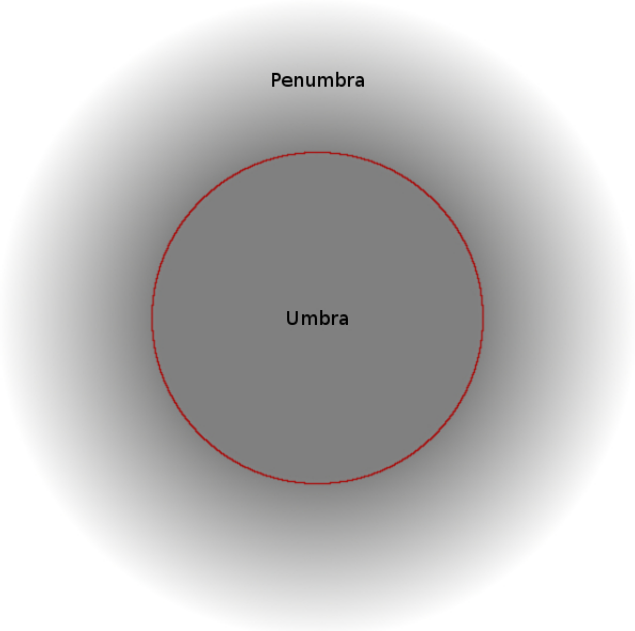


Figur 2.8: *Dette billede er taget fra* `http://www.billcasselman.com/wording_room/antumbra_umbra.htm`.
Denne figur viser forskellen mellem umbra og penumbra. Umbra er den del der er helt i skygge, mens penumbra er den del der kun er delvis i skygge.

Fremgangsmetoden for denne algoritme er at man først genererer de normale hårde skygger. For hver silhuet kant laves en såkaldt kile (eng: wedge) der repræsenterer penumbraen. En kile, er en geometrisk figur der består af 2 trekanter og 3 trapezer, et billede af dette kan ses på figur 2.9. For hver af disse kiler laves så glidende overgang fra de hårde skygger og kilerne (penumbraen fra objektet). For at opnå denne effekt, kan man med fordel bruge alpha blending, for at få den flydende overgang mellem de hårde skygger og penumbraen. Et simpelt eksempel på dette kan ses på figur 2.10.



Figur 2.9: *Dette billede er taget fra* http://upload.wikimedia.org/wikipedia/commons/e/ea/Triangular_prism_wedge.png.
Dette viser hvad en kile er. Den består af 2 trekanter samt 3 trapezer.



Figur 2.10: *Denne figur viser et simpelt eksempel på hvordan penumbra wedges fungerer. Her kan man se at Umbraen ligger helt i skygge. Penumbraen falder i intensitet jo længere væk den kommer fra selve skyggen, hvilket giver den flyende overgang.*

3 Shadow mapping

I dette afsnit vil vi komme ind på teorien omkring shadow mapping algoritmen. Vi vil komme ind på selve algoritmen og nogle af de problemer der kan opstå med algoritmen, blandt andet moire pattern og peter panning. Til sidst vil vi komme ind på bløde skygger med shadow mapping algoritmen, ved brug af percentage closer filtering.

3.1 Algoritmen

Princippet bag *shadow mapping* er i sig selv rimelig simpel. Det er selve implementeringen, samt de problemer der generelt opstår med shadow mapping, der gør den avanceret (dette kommer vi ind på senere).

Algoritmen er som følger:

1 Transformer kameraet op i lyset.

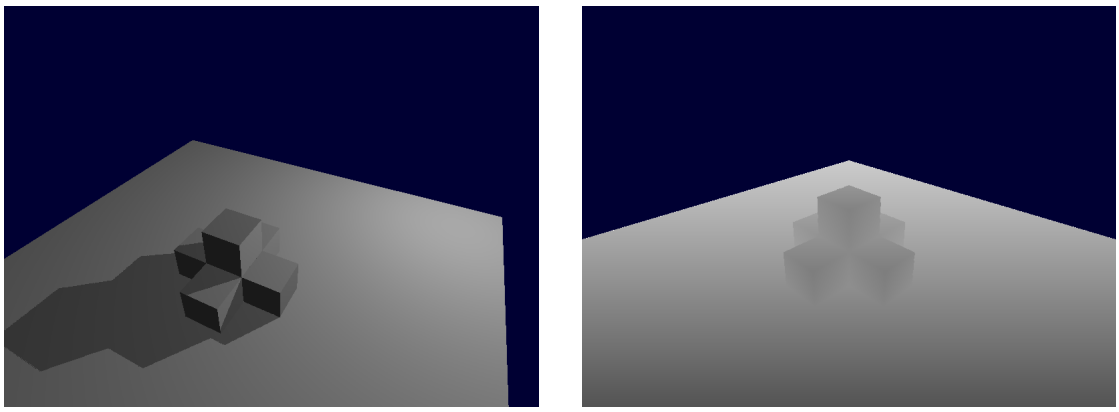
Det skal helst pege på det samme punkt som det gjorde før transformationen.

2 Lav et dybde map af scenen ud fra lystets position.

3 Transformer kammeret tilbage til den oprindelige position.

4 Render scenen, men med en ekstra test:

- Sammenlign afstanden fra punkterne man vil tegne ud fra lyset, med værdien udregnet for samme punkt i dybde-mappet (fra punkt 2).
- Hvis afstanden er højere, betyder det at punktet ligger i skygge og skal derfor farves som værende i skygge.



Figur 3.1: *Denne figur viser den reelle scene med skygger (venstre) og et eksempel på et dybde map udregnet fra lyset (højre).*

Sammenligningen mellem dybdeværdierne fra punkterne set fra kameraet og dybdeværdierne i shadowmappet, sker i fragment-shaderen (mere om dette i afsnittet omkring vores implementation). Hvis dybdeværdien i shadowmappet er lavere end dybdeværdien for punktet man er i gang med tegne (i forhold til lyset), betyder det at punktet ligger i skygger. Det er også denne sammenligning der giver anledning til en række problemer.

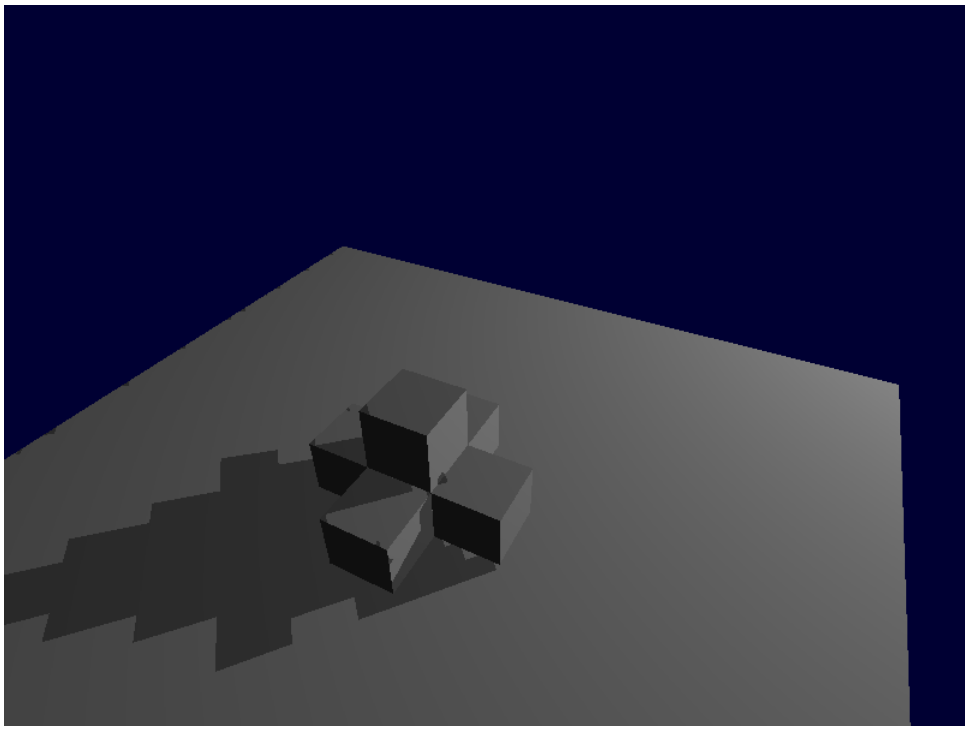
En anden vigtige del af algoritmen er opslaget af et punkt i shadowmappet. Dette gøres normalt ved at man først transformerer et givent punkt fra kamera koordinater tilbage til verdens koordinater og derefter transformere man punktet op i lyset. Dette betyder, at man skal gemme den transformation matrix man brugte, da man i første omgang transformerede kameraet op i lyset.

3.2 Problemer

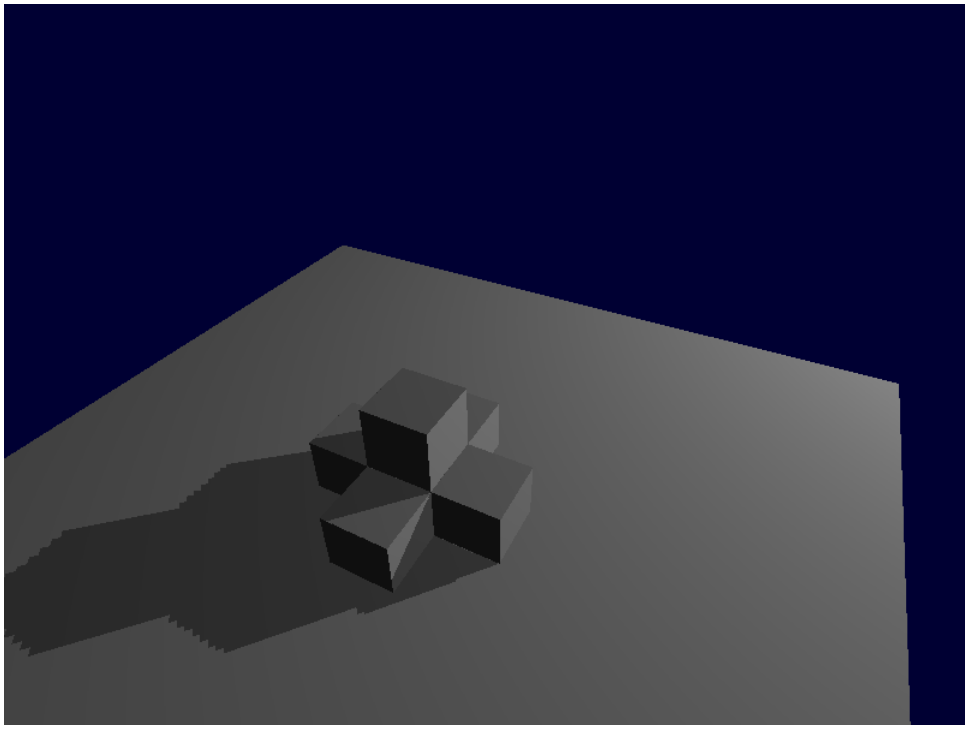
Der er en række forskellige aspekter og problemer man skal tage højde for når man arbejder med shadowmapping. I dette afsnit vil vi komme ind på de vigtigste af disse. Vi vil gennemgå vigtigheden af opløsningen på shadow mappet, samt hvad moire's pattern er og hvordan det løses. Til sidste i dette afsnit vil vi komme ind på et andet problem, kaldet peter panning, som en af løsningerne på moire's pattern kan forsage.

3.2.1 Opløsning på shadowmappet

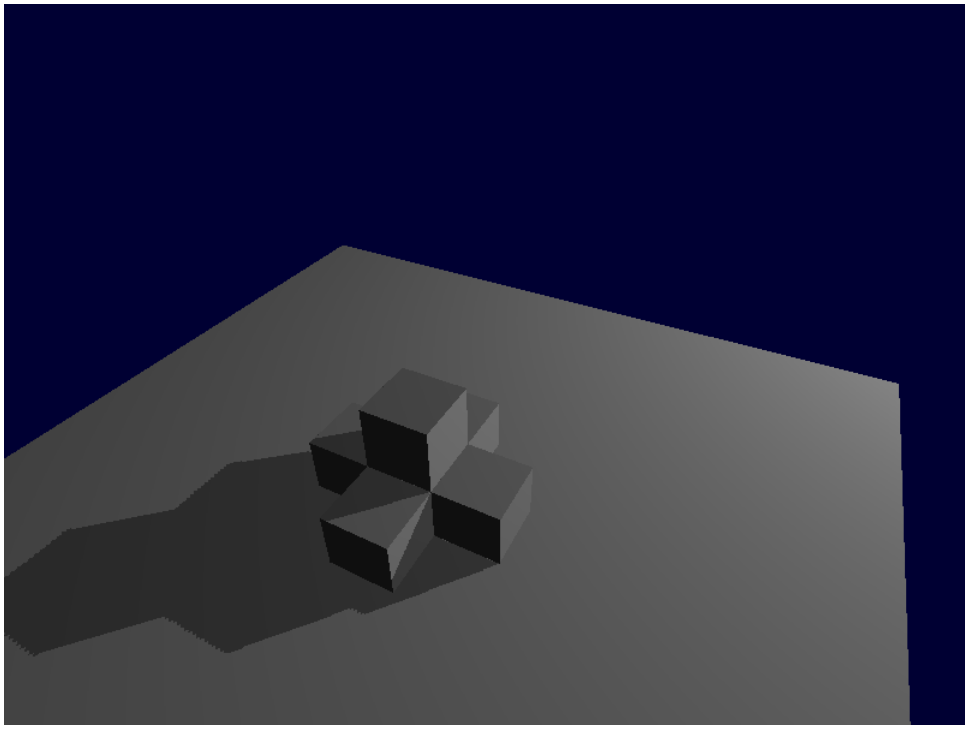
Det er vigtigt at tage højde for opløsningen på shadowmappet, for at få ordenlige skygger. En for lav opløsning vil forsage ujævne og ukorrekte skygger. Dette skyldes at shadowmappet vil blive mere og mere upræcist jo mindre opløsningen bliver. Følgende billeder viser en scene med forskellige opløsninger af shadowmappet.



Figur 3.2: *Shadowmap med en opløsningen* 80×60



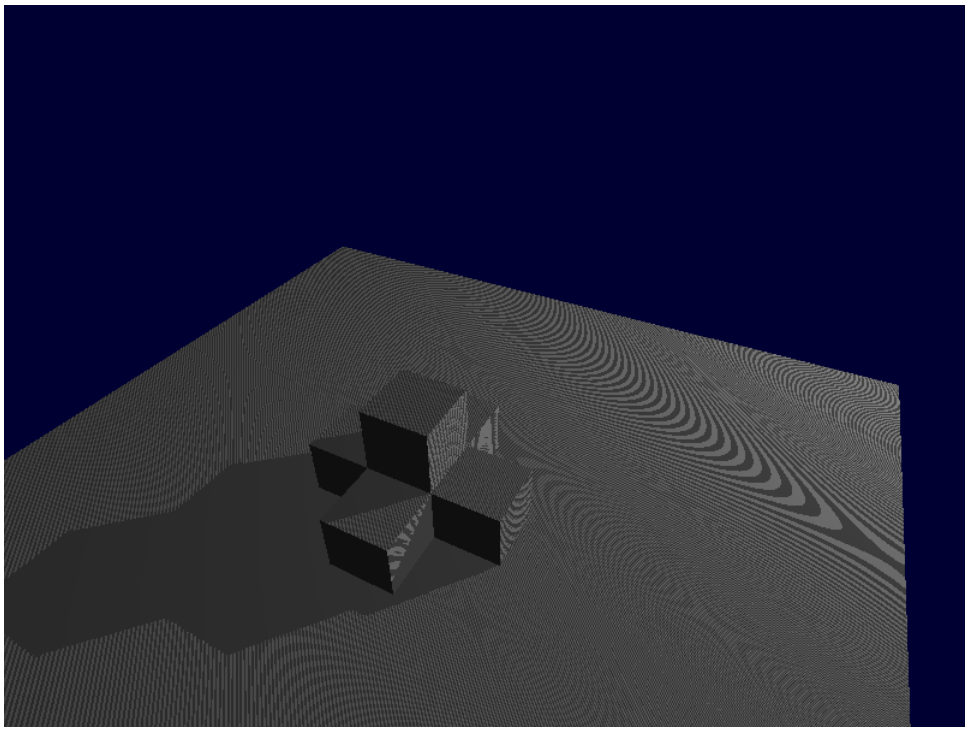
Figur 3.3: *Shadowmap med en opløsningen* 800×600



Figur 3.4: *Shadowmap med en opløsningen* 1600×1200

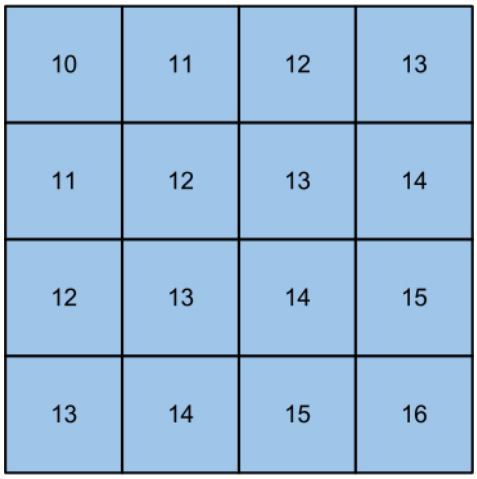
Som man kan se så afhænger skyggernes udseende meget af opløsningen på shadowmappet. Problemet med de kantede skygger opstår hvis der er flere punkter (set fra kameraet) der mapper til det samme punkt i shadowmappet. Det er derfor det er vigtigt at shadowmappet har en tilstrækkelig opløsning. I sidste af de viste tilfælde er opløsningen dobbelt så stor som opløsningen af kameraet. Det er meget vigtigt at huske at opløsningen også kan blive unødvendigt høj. Dette vil betyde at grafikkortet arbejder meget mere end nødvendigt, og resultatet bliver ikke nødvendigvis tilsvarende bedre.

3.2.2 Moiré pattern

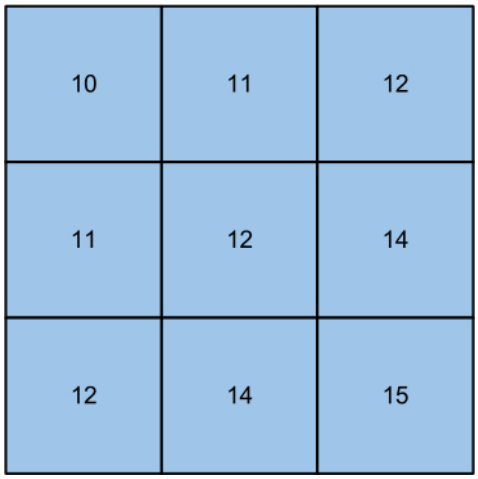


Figur 3.5: *Denne figur viser et eksempel på Moirés pattern fra vores program.*

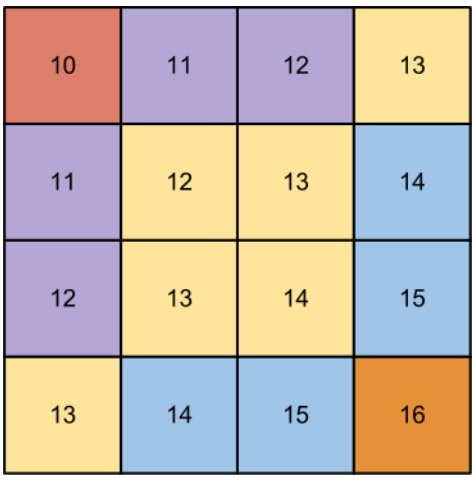
Som set på figur 3.5, ser scenen ikke korrekt ud. Som man kan se er selve skyggerne korrekte, men der er derudover også et uønsket mønster. Dette mønster kaldes Moirés pattern (andre kilder refererer dog til det som *shadow-acne[OT13]*). Problemet opstår ved at et givent punkt, set fra kameraet, har større dybde (i forhold til lyset), end værdien punktet mappes til i shadow mappet, selvom punktet ikke ligger i skygge.
Følgende forklaring illustrere problemet:

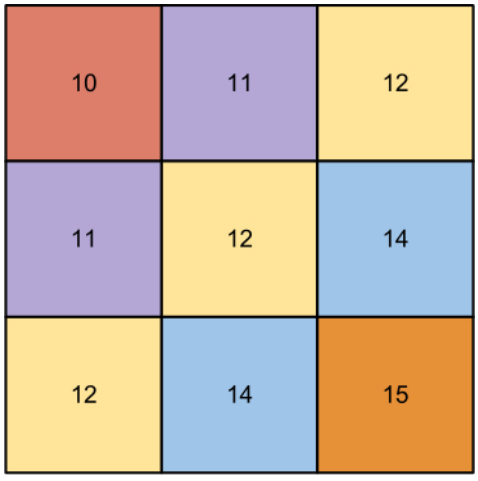


(a) *Dette er dybdeværdier (i forhold til lyset) af nogle punkter set fra kameraet, bemærk at ingen af punkterne ligger i skygge*

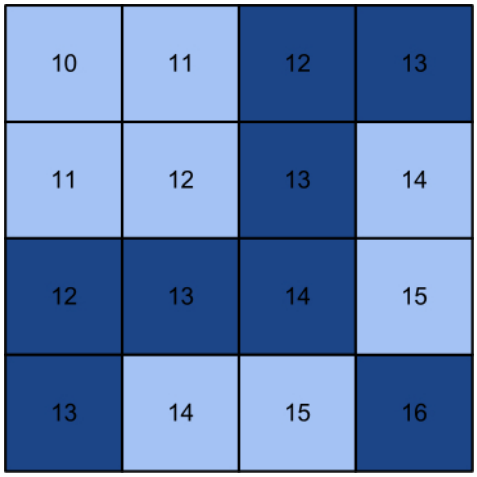


(b) *Dette er dybdeværdier af nogle punkter set fra lyset, det vil sige vores shadowmap*





(c) *Her kan man se hvordan diverse punkter fra kameraet, mapper til værdierne i shadowmappet. De røde pixels på på venstre figur mapper til de røde figure på højre osv. Hvis værdien på en given pixel fra kameraet (venstre figur) er højere end værdien i shadowmappet (højre figur), så ligger punktet i skygge.*

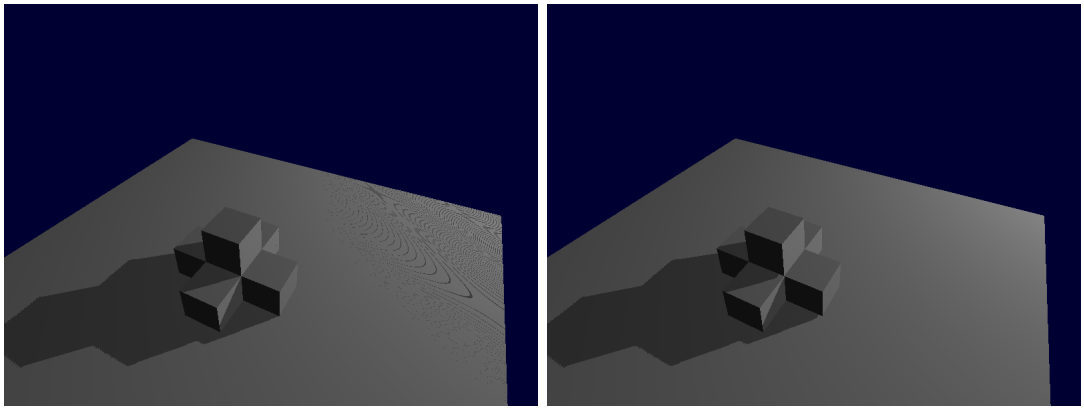


(d) *Her kan man så se nogle af pixelsne ligger i skygge selv om de ikke burde (Husk at ingen af punkter lå i skygge i første figur). Denne figur er et eksempel på moire's pattern når man er helt tæt på.*

Figur 3.6: *Disse figurer viser dybdeværdier i et shadowmap, som forsager moirés pattern.*

Overstående situation sker fordi de givne punkter set fra kameraet fylder 4x4 pixels, imens selv samme punkter set fra lyset fylder 3x3 pixels. Så når man laver sammenligningerne vil nogle af punkterne blive sammenlignet med en ukorrekt værdi i shadow mappet og dermed være i skygge selvom der ikke er noget der blokerer lyset.

Man kan løse dette ved at sætte opløsningen på shadowmappet meget højt op, men dette er meget upraktisk og det kan ikke anbefales, da det er meget krævende for grafikkortet. En anden løsning på dette er, at introducere en såkaldt "bias", det vil sige ved sammenligningen ligger man en bias til værdierne i shadowmappet. Dette vil betyde at dybdeværdierne i shadowmappet, vil være højere end dybdeværdierne for samme punkter set fra kameraet og derfor vil punkterne blive tegnet korrekt (det vil sige ikke i skygge).



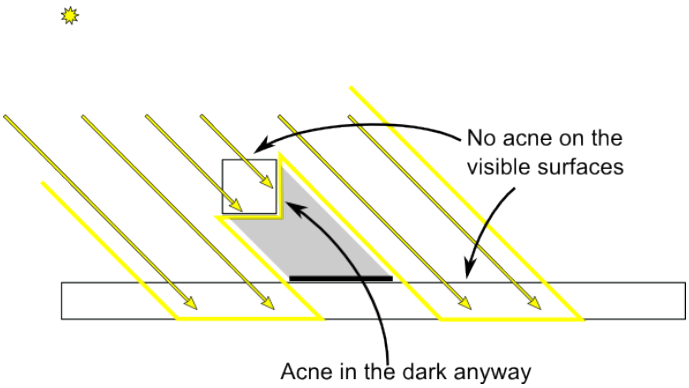
(a) *I dette billed bliver der brugt en bias konstant på 0.00001. Som man kan se giver det et væsentligt bedre billed, men det er stadig ikke nok til at fjerne problemet helt.*

(b) *Her bliver der brugt en bias konstant på 0.0001. Bemærk at alle de uønskede mønstre nu er væk*

Figur 3.7: *Denne figur viser hvilken effekt forskellige værdier på biasen gør. Hvis biasen er for lille bliver moire's pattern ikke fuldstændigt fjernet.*

Som man kan se på figur 3.7 så løser tilføjelsen af en bias problemet. Man skal dog være opmærksom på at der ikke findes *én* korrekt bias-værdi. Værdien for ens bias afhænger af scenen og opløsningen på shadowmappet. Desuden introducerer bias løsningen et nyt problem, som kaldes for "Peter panning"(Dette bliver dækket i følgende afsnit).

Der er også et alternativ til bias-metoden. Denne metode involvere at man culler front-faces når man laver selve shadow-mappet og culler backfaces når man tegner selve scenen.



Figur 3.8: *Billeder er taget fra OpenGL-tutorial [OT13].*
Her kan man se en demonstration af shadowmapping med culling-metoden, som man kan se flyttes moirepattern nu over på bagsiden er af figuren. det vil sige den del hvor figuren skygger for sig selv.

Denne metode sørger for at overflader i lys aldrig vil kaste skygge på sig selv, da dybdeværdierne for disse overflader (det vil sige front-faces) er mindre end dybdeværdien på de overflader der vender væk fra lyset (det vil sige back-faces).

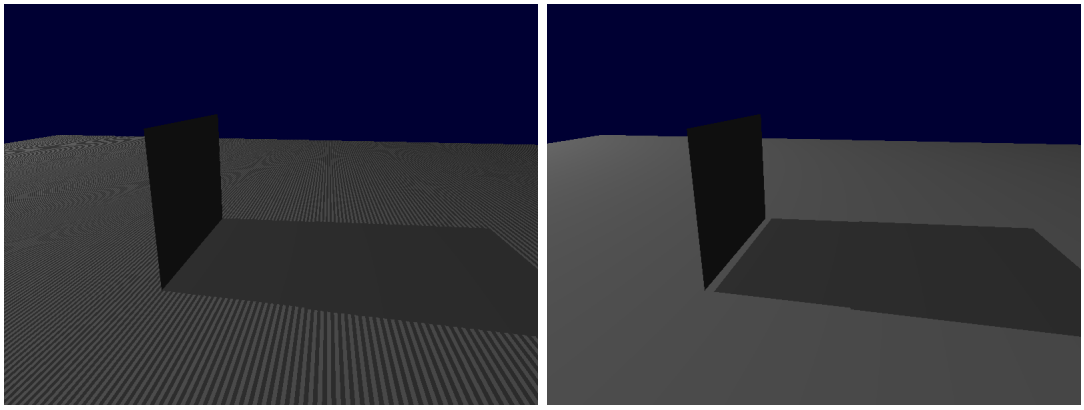
Tilgengæld flytter dette, moirés pattern over på back-facesne af vores figurer. Det vil sige der hvor figuren kaster skygge på sig selv. Dette kan man afhjælpe ved at antage, at hvis et punkts normal peger væk fra lyset, så ligger punktet i skygge. Dette gør man blot ved at finde prikproduktet mellem punktets normal og retning fra punktet til lyset. Hvis prikproduktet er negativt (det vil sige vinklen mellem de to er over 90), så vil punktet være i skygge.

Denne metode kræver dog, at man kun har lukkede figurer. For eksempel vil en enestående polygon der vender mod lyset ikke blive renderet i nogle tilfælde, da vi culler back-faces. Hvis polygonen vender mod lyset, vil den heller ikke blive medtaget i shadowmappet, da den vil forsvinde når man ser på den fra lyset (da vi laver front-face culling når shadowmappet laves).

Bemærk at man også kan bruge en bias når man bruger culling metoden i stedet for at tjekke normalerne.

3.2.3 Peter-panning

Som nævnt tidligere forsager introduktionen af en bias et problem kaldet *peter-panning*[1].



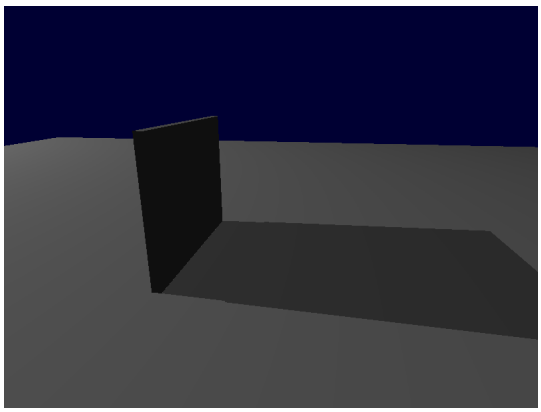
(a) *En test scene uden bias. Her er der ingen peter panning og et meget tydeligt moires-pattern*

(b) *Sammen Scene som før, men nu uden moires-pattern. Her er det tydeligt at se hvilken effekt tilføjelsen af en bias har. Fænomentet hvor skyggen først starter et stykke efter figuren kaldes som sagt peter panning.*

Figur 3.9: *Denne figur viser hvordan man kan fjerne moirés ved at bruge bias.*

Fænomenet opstår på grund af den bias man har sat ind (Hvis man burger culling metoden hvor man tjekker normalerne, vil der ikke opstå peter-panning). Da man lægger biasen til dybdeværdierne i shadowmappet, vil dette resulterer i, at visse punkter som burde være i skygge, ender med ikke at være det. Dette resulterer i, at punkterne der er tæt på en skygges start, ender op med ikke at være i skygge.

Løsningen på dette, er blot at undgå helt flad geometri. Grunden til at dette løser problemet, er at biasen (burde) være lille nok til ikke at gøre en synlig forskel, når man har med ikke-flad geometri at gøre.



Figur 3.10: *Her kan man se at en ikke-flad version af geometrien fra før, har løst problemet*

[1] Opengl-tutorial kalder fænomenet peter-panning, da det kan få objekter til at se ud som om de svæver

3.3 Bløde skygger med shadow mapping

Indtil videre har vi kun arbejdet med hårde skygger med shadow mapping. I denne sektion vil vi komme ind på hvordan man kan lave bløde skygger. Der findes mange forskellige metoder til at lave bløde skygger med shadow mapping algoritmen, men vi vil kun komme ind på én metode. Den metode vi vil omtale hedder Percentage-Closer Filtering.

3.3.1 Percentage-Closer Filtering (PCF)

Percentage-Closer Filtering går kort sagt ud på at man finder ud af hvor kraftig skyggen er for hvert enkelt punkt. Dette finder man ud af ved at se på de omkring liggende punkter for punktet man slår op.
Måden man finder de omkring liggende punkter er blot at tage ligge et offset til det opslåede punkt i hver retning. Størrelsen på offsetet er baseret på shadow mappets størrelse:

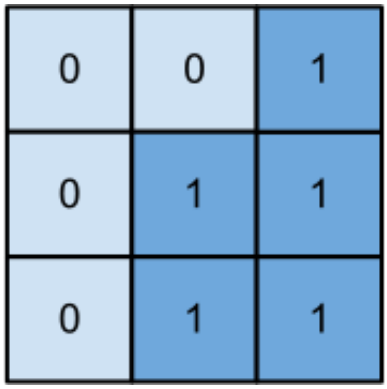
$$offsetX = \frac{1}{Shadowmap\ bredte}$$

$$offsetY = \frac{1}{Shadowmap\ højde}$$

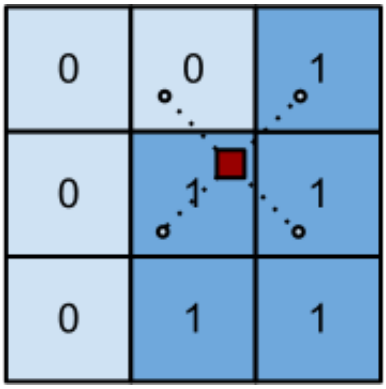
Når man taler om PCF bliver der ofte talt om *PCF 2x2*, *PCF 3x3* osv. Dette beskriver blot hvor

mange samples der bliver brugt, det vil sige hvor mange punkter der bliver sammenlignet med.

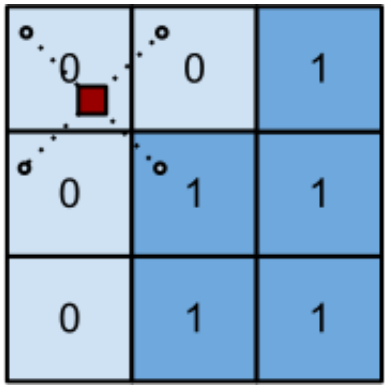
Med *PCF 2x2* sammenligner man kun med hjørne punkterne som set her:



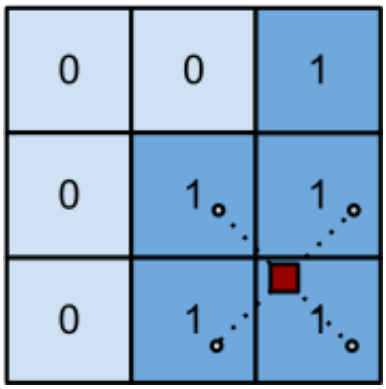
(a) *Dette er et eksempel på et udsnit af et shadowmap. Der hvor værdierne er 1 er der hvor der skal være skygge i scenen set fra kameraet.*



(b) *I dette tilfælde vil det opslåede punkt ligge* $\frac{3}{4}$ *skygge. Det vil sige skyggen kun er* $\frac{3}{4}$ *så kraftig, som den ville have været, hvis punktet havde ligget fuldkommen i skygge.*



(c) *Her ligger punktet kun* $\frac{1}{4}$ *i skygge.*

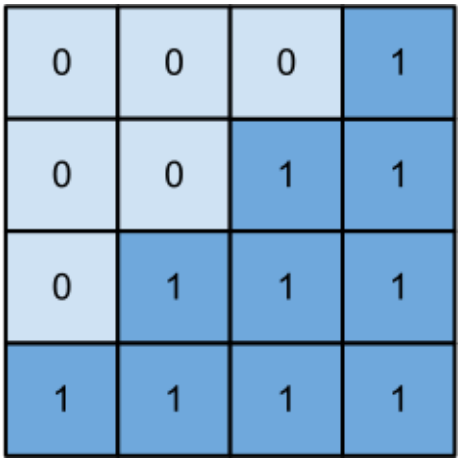


(d) *Her ligger punktet fuldkommen i skygge.*

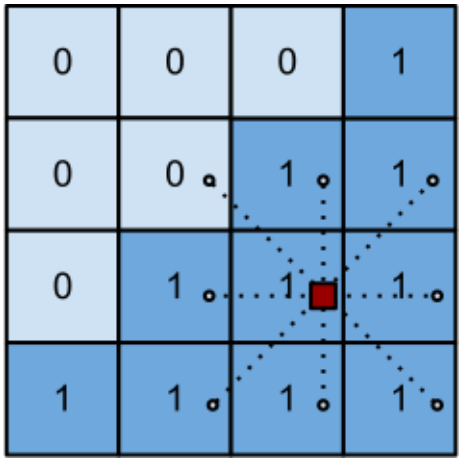
Figur 3.11: *Demonstration af PCF 2x2.*

Som man kan se vil man i de fleste tilfælde slå op i samme punkt i shadowmappet som det opslåede punkt ligger i, samt 3 omkring liggende punkter.

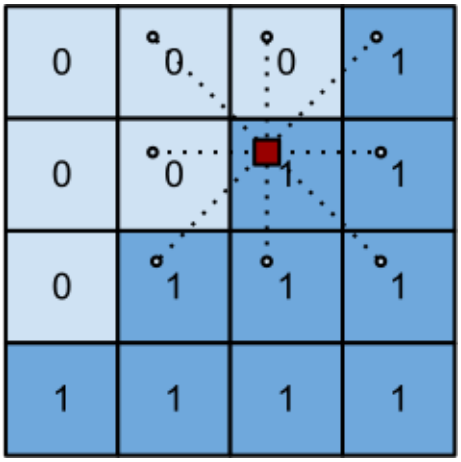
Når man derimod bruger *PCF 3x3* sammenligner man med alle omkring liggende punkter:



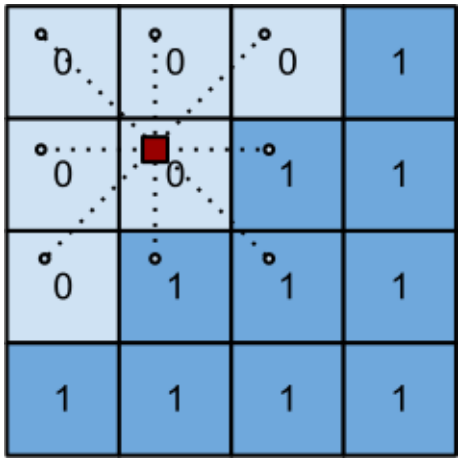
(a) *Ligesom før ser vi på et eksempel på et udsnit af et shadowmap. Der hvor værdierne er 1 er der hvor der skal være skygge i scenen set fra kameraet.*



(b) *Punktet ligger i* $\frac{8}{9}$ *skygge.*



(c) *Her er skyggens kun* $\frac{6}{9}$ *så kraftig som fuldkommen skygge.*



(d) *Her er skyggens kun* $\frac{3}{9}$ *så kraftig som fuldkommen skygge.*

Figur 3.12: *Demonstration af PCF 3x3. Bemærk at man også slår op i selve punktet (den røde firkant)*

Hvis man bruger denne fremgangs metode for alle punkter der renderes vil man få en blød overgang i skyggerne. Jo flere samples man bruger jo blødere bliver skyggerne, da man sammenligner med flere og flere punkter og får derfor en mere og mere glidende overgang. Det bliver samtidig også mere og mere krævende af grafikkortet.

II Implementering

4 Generelt om implementationen

I dette afsnit vil vi forklare de vigtige dele af vores implementation af begge algoritmer. Vi vil også kort komme ind på den overordnede struktur i vores program, samt de vigtigste klasser vi har lavet.

4.1 Object

I vores implementation bruger vi en klasse, *Object*, til at håndtere en samling af polygoner der tilsammen bliver en figur.

En polygon i vores implementation er en klasse som indeholder al brugbar information om en enkelt polygon. Den indeholder alle informationer der er relevant for enten *skyggevolumener*- eller *Shadow mapping* algoritmen.

4.2 Scener

En scene i vores program, er basalt set en klasse hvori alt vores geometri bliver beskrevet. Vi har valgt at beskrive vores geometri direkte i klasser i stedet for i eksterne filer, da vi syntes det var lettere.

4.3 Renderes

Vi gør brug af forskellige *rendere*-klasser som vi har implementeret. Disse klasser gør brug af en *basic-rendere*-klasse som indeholder funktionalitet til at tegne en scene. Hvis en scene skal tegnes på en specifik måde, f.eks. med *Shadowmapping* algoritmen nedarver vi blot fra denne klasse og laver de relevante ændringer/tilføjelser.

Vi har følgende nedarvninger fra denne klasse som vi gør brug af, alt efter hvilke af de 2 algoritmer en scene skal tegnes med:

- ShadowVolumeRendere, *renderer scenen med standard skygge volume algoritmen.*
- ShadowVolumeStencilRendere, *tegner indeholdet af stencil bufferen for en given scene.*
- ShadowVolumeSilhuetsRendere, *renderer scenen med skygge volume algoritmen og gør brug silhuetter.*
- ShadowVolumeActualSilhuetsRendere, *renderer scenen hvor man kun kan se figurenes silhuetter.*
- ShadowMapRendere, *renderer scenen med shadowmapping algoritmen.*
- ShadowMapContentRendere, *tegner indeholdet af shadowmappet for en given scene.*
- ShadowMapBiasRendere, *renderer scenen med shadowmapping algoritmen der bruger bias.*
- ShadowMapBiasContentRendere, *tegner indeholdet af shadowmappet, der bruger bias, for en given scene.*

5 Skyggevolumen

I dette afsnit vil vi gennemgå de vigtigste punkter af vores implementation af skyggevolumener. Vi har i vores implementation brugt Z-fail variationen af algoritmen.

Efter geometrien i en scene er blevet renderet, bliver skyggerne tilføjet. Vores implementation af selve genereringen af skygger følger rimelig tydeligt algoritmen beskrevet tidligere:

```
1  glPushAttrib(GL_ALL_ATTRIB_BITS);{
2    glDisable(GL_LIGHTING); // Disable lighting
3
4    // Disable color mask
5    glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);
6    // Disable depth mask
7    glDepthMask(GL_FALSE);
8
9    glEnable(GL_STENCIL_TEST); // Enable stencil test
10   // Set stencil function to always be used.
11   // The second and third parameter does not matter,
12   // since we set GL_ALWAYS in the first parameter
13   glStencilFunc(GL_ALWAYS, 0, 0xff);
14   glEnable(GL_CULL_FACE); // Enable culling
15
16   glCullFace(GL_FRONT);
17   glStencilOp(GL_KEEP, GL_INCR, GL_KEEP);
18   this->MakeShadowVolume(scene);
19
20   glCullFace(GL_BACK);
21   glStencilOp(GL_KEEP, GL_DECR, GL_KEEP);
22   this->MakeShadowVolume(scene);
23
24   glDisable(GL_CULL_FACE);
25   glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
26   glDepthMask(GL_TRUE);
27
28   this->DrawShadows();
29 }glPopAttrib();
```

Her kan man tydeligt se at vi først slår lyset samt skrivning til farve- og dybdemasken fra. Derefter slår vi stencil-testen til og sætter selve stencil-testen til altid at blive brugt (ved at bruge GL_ALWAYS flaget).
Dette gør vi, da vi gerne vil have at stencil-testen bliver udført på alle punkter. Det er dog værd at nævne, at der ikke bliver testet på punkter der bliver skåret væk via clipping (Hvilket er derfor det ikke er lige til at lave uendelig lange skygge med Z-fail).
Vi slår derefter *culling* af *front-faces* til og sætter derefter stencil bufferen til at inkrementere, hvis dybde testen fejler for en given pixel. Dette svarer til at vi tæller hvor mange skygge-bagsider, som et punkt ligger foran.

Derefter *culler* vi *back-faces* i stedet og sætter stencil bufferen til at dekrementere, hvis dybde testen fejler. Dette svarer til, at tælle hvor mange skygge-forsider punktet ligger bagved.

"MakeShadowVolume()"er den funktion der "rendere"vores volumener. Det er når disse volumener bliver tegnet, at stencil-testen bliver udført.
Det er kun polygoner der er synlige fra lyset, der bliver lavet volumener for, da det kun er dem der vil kaste skygge. En polygon ses som synlig fra lyset, hvis minimum en af dens normaler, i mindst et af dens punkter, peger mod lyset.

De mest interessante dele af MakeShadowVolume() er udregningen af silhuetterne og volumenerne. Følgende kode viser vores implementation af funktionen der "tegner"skyggevolumenerne.

```
1  void ShadowVolumeRendere::MakeShadowVolume(Scene* scene){
2    Polygon polygon;
3    Object object;
4    Position lightPos;
5
6    lightPos = scene->GetLightSource().GetPosition();
7
8    //scene->GetCurrentSize() returns the number of objects in the
       scene.
9    for(int o = 0; o < scene->GetCurrentSize(); o++){
10     object = scene->GetObject(o);
11     //object.GetCurrentSize() returns the nummber of polygons in the
         object.
12     for(int i = 0; i < object.GetCurrentSize(); i++){
13       polygon = object.GetPolygon(i);
14
15       if (polygon.visible){
16         //Top plane of volume
17         glBegin(GL_POLYGON);{
18           Vertex vertex;
19           for(int q = 0; q < polygon.GetVerticesCount(); q++){
20             vertex = polygon.GetVertex(q);
21             glVertex3fv(vertex.array);
22           }
23         }glEnd();
24
25
26         //Bottom plane of volume
27         glBegin(GL_POLYGON);{
28           Vertex vertex;
29           for(int q = polygon.GetVerticesCount() - 1; q >= 0; q--) {
30
31             vertex = polygon.GetVertex(q).Copy();
32             vertex -= lightPos;
33
34             Normalize(vertex);
35             vertex *= Vertex(VOLUME_LENGTH, VOLUME_LENGTH,
                   VOLUME_LENGTH);
36             vertex += lightPos;
37
38             glVertex3fv(vertex.array);
39           }
40         }glEnd();
41
42         for (int j = 0; j < polygon.GetVerticesCount(); j++ )
```

```
43          {
44            // Get The Points On The Edge
45            Vertex v1;
46            Vertex v2;
47
48            v1 = polygon.GetVertex(j).Copy();
49            v2 = polygon.GetVertex((j + 1) %
                 polygon.GetVerticesCount()).Copy();
50
51            // Calculate The Two Vertices In Distance
52            Vector v3 = v1.Copy(),
53                   v4 = v2.Copy();
54
55            v3 -= lightPos;
56            Normalize(v3);
57            v3 *= Vertex(VOLUME_LENGTH, VOLUME_LENGTH, VOLUME_LENGTH);
58            v3 += lightPos;
59
60            v4 -= lightPos;
61            Normalize(v4);
62            v4 *= Vertex(VOLUME_LENGTH, VOLUME_LENGTH, VOLUME_LENGTH);
63            v4 += lightPos;
64
65            glBegin(GL_QUADS);{
66              glVertex3fv(v2.array);
67              glVertex3fv(v1.array);
68              glVertex3fv(v3.array);
69              glVertex3fv(v4.array);
70            }glEnd();
71          }
72        }
73      }
74    }
75  }
```

I denne stump kode går vi alle kanter i vores geometri igennem og tegner så vores volumener ud fra dem. Der bliver som sagt kun set på polygoner der er synlige fra lyset. Først bliver toppen og bunden af vores skyggevolumen tegnet, derefter siderne. Om polygonen er synlig fra lyset eller ej, bliver udregnet når man skifter til scenen eller skifter rendere, det vil sige at det bliver udregnet en gang og ikke i hvert skærmbillede. En polygon anses for at være synlig hvis minimum en af dens normaler ikke peger væk fra lyset.

Det er vigtigt at vide at VOLUME_LENGTH, er en værdi der er tilstrækkelig stor, men ikke stor nok til volumenerne går ud over vores far-plane. Dette skyldes at hvis bunden af volumenerne ligger uden for far-planet vil den aldrig blive "tegnet", da den bliver clippet af far-planet. Dette betyder at stencil testen aldrig bliver udført på den. Dette leder til, at visse punkter ikke ligger i skygge, selvom de burde. Hvis man ønsker at have volumener af uendelig længde, kan man eventuelt clampe værdierne for bunden af vores skyggevolumener til far-plane. Dette har vi dog ikke gjort og det er ikke nødvendigvis så trivielt at implementerer, som man skulle tro.

Efter at man har "tegnet" og udført tests for både front- og back-faces, tegner man sin skygge polygon over hele skærmen. Dette bliver gjort i "DrawShadows()":

```
void ShadowVolumes2::DrawShadows(float shadowIntensity) {
  glPushAttrib(GL_ENABLE_BIT | GL_COLOR_BUFFER_BIT);{
    glPushMatrix();{
      glLoadIdentity();
      glMatrixMode(GL_PROJECTION);
      glPushMatrix();{
        glLoadIdentity();
        glOrtho(0, 1, 1, 0, 0, 1);
        glDisable(GL_DEPTH_TEST);

        glEnable( GL_BLEND );
        glBlendFunc( GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA );

        glColor4f(0.0f, 0.0f, 0.0f, shadowIntensity);
        glBegin(GL_QUADS);{
          glVertex2i(0, 0);
          glVertex2i(0, 1);
          glVertex2i(1, 1);
          glVertex2i(1, 0);
        }glEnd();

        glEnable(GL_DEPTH_TEST);
      }glPopMatrix();
      glMatrixMode(GL_MODELVIEW);
    }glPopMatrix();
  }glPopAttrib();
}
```

Her kan man se at vi først går over i et Orthogonalt-view. Hvilket vil sige vi går fra at arbejde i en 3D-verden til et 2D-verden. Kort sagt bliver alle koordinater i 3D konverteret til 2D-koordinater som ligger mellem 0 og 1. Derefter slår vi dybde testen fra, så vores polygon bliver tegnet på skærmen uanset dens dybde. Vi slår blending til for at kunne lave semi-transperante polygoner (så vores skygge ikke bliver helt sort). Vi tegner derefter vores polygon som dækker hele skærmen, men fordi vi i starten af funktionen har:

```
  glStencilFunc(GL_NOTEQUAL, 0x0, 0xff);
  glStencilOp(GL_REPLACE, GL_REPLACE, GL_REPLACE);
```

Vil der nu kun bliver tegnet der hvor stencil-bufferen indeholder værdier over (eller under) 0. Til sidst bliver dybde testen slået til igen og vi går derefter tilbage til model-viewet.

5.1 Silhuetter

Det første man skal gøre når man skal lave en silhuette, er at finde alle nabo-polygoner til alle polygonerne i den givne scene. Dette er rimelig beregningstungt og bør derfor kun blive gjort når det er nødvendigt.

Man gør dette, ved at sammenligne alle kanterne i ens geometri. Hvis to polygoner deler kant betyder det, at de er naboer.

På grund af vores program struktur blev vi nød til først at fjerne polygon-dubletter, før vi udregnede nabo-polygonerne. Dette skyldes, at under visse omstændigheder, kunne en kant have mere end en nabo-polygon (f.eks. to 3D bokse placeret lige op ad hinanden). Vores løsning, var blot at skjule dubletterne, ved at sige de ikke er synlige.

I vores implementation, skjuler vi dubletter og udregner naboerne umiddelbart efter vi har udregnet hvilke polygoner, der er synlige fra lyset. Dette betyder, ligesom før, at det kun sker en gang.

Det er under renderingen af skyggevolumenerne, at vi gør brug af de fundene naboer.

Når vi er ved at tegne en kant til volumen af en polygon, tjekker vi om polygonen har en nabo på denne kant. Hvis den har, skal man normalt tjekke om denne nabo-polygon er synlig fra lyset af.

Ændringen til den oprindelige kode der tegner skyggerne er derfor blot tilføjelsen af:

```
1 if(polygon.GetNeighbor(j) == 0 || !polygon.GetNeighbor(j)->visible){
2   ...
3 }
```

omkring den del der tegner siderne på vores skyggevolumen. Denne lille stump kode, ser om den j'te kant i den nuværende polygon, har en nabo-polygon. Hvis den ikke har returneres 0 og derfor er kanten, en del af vores silhuette og bliver derfor tegnet.

Toppen og bunden af volumenerne, hvor der bruges silhuetter, bliver tegnet på normal vis.

6 Shadow mapping

Vi har implementeret shadow mapping på begge måder (som beskrevet i teori-delen). Det vil sige at vi har implementeret både shadowmapping, hvor der bliver gjort brug af en bias, og en hvor der bliver gjort brug af front- og back-face culling.

Begge implementationer har visse ting tilfældes. Opsætningen af selve dybde-mappet er for eksempel helt ens. Den store forskel ligger i vertex- og fragment-shaderne der bliver brugt.

Vi har i disse shadere også implementeret "phongs lys-model", men vi vil dog ikke forklare nærmere om dette. For at undgår forvirring, har vi også klippet alle udregninger der er relateret til phongs lysmodel ud af eksemplerne.

6.1 Generel implementation af shadowmapping

Som nævnt tidligere er der visse ting, de to implementationer har tilfældes. Mest nævneværdig er funktionerne:

- InitShaders
- InitShadowFBO
- SetupTextureMatrix

6.1.1 Initialisering af shaderne

InitShaders() funktionen bruges til initialisere vores shadere og binde dem til et *handle*. Funktionen ser således ud:

```
1  GLhandleARB vertexShaderHandle;
2  GLhandleARB fragmentShaderHandle;
3
4  vertexShaderHandle   = this->LoadShader(this->vertexShaderPath,
       GL_VERTEX_SHADER);
5  fragmentShaderHandle = this->LoadShader(this->fragmentShaderPath,
       GL_FRAGMENT_SHADER);
6
7  // Creates the shader that is going to be used to apply shadows
8  this->shadowShaderId = glCreateProgramObjectARB();
9
10 // Attaches the two shaders loaded previously
11 glAttachObjectARB(shadowShaderId, vertexShaderHandle);
12 glAttachObjectARB(shadowShaderId, fragmentShaderHandle);
13
14 // Links our shader
15 // This makes a executeable useable by our GPU
16 glLinkProgramARB(shadowShaderId);
17
18 //Gets the handle for out shadowmap (used to load the shadow map
       texture to shaders)
19 this->shadowMapUniform = glGetUniformLocationARB(shadowShaderId,
       "ShadowMap");
```

LoadShader er blot en funktion, der giver et handle tilbage som vores shader kan ligge i. Det skal dog siges at den i denne sammenhæng, *test-compiler* vores shadere og melder fejl hvis der er noget galt.

6.1.2 Initialisering af ShadowFBO

FBO står for "Frame buffer object", hvilket basalt set er et objekt der repræsentere en skærm man kan tegne på. Det er i dette object vores shadowmap bliver lagt. Initialiseringen af shadowmap FBO'et bliver gjort på følgende måde:

```
1  void ShadowMapRendere::InitShadowFBO() {
2    // The Width and Height of out shadowmap
3    // DEFAULT_WIDTH = 800, DEFAULT_HEIGHT = 600, SHADOW_MAP_RATIO = 2
4    int shadowMapWidth = DEFAULT_WIDTH * SHADOW_MAP_RATIO;
5    int shadowMapHeight = DEFAULT_HEIGHT * SHADOW_MAP_RATIO;
6
7    // Used to store the current status of our FBO
8    // Mainly used for errordetection
9    GLenum status;
10
11   //Creates a one layered texture and places the handle in
        depthTextureId
12   glGenTextures(1, &depthTextureId); //Generates a texture and stores
        it in depthTextureID
13   glBindTexture(GL_TEXTURE_2D, depthTextureId); //Binds
        depthTextureId as a 2D Texture
14
15   //Sets up the parameters for the shadow texture.
16   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
17   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
18   glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP );
19   glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP );
20
21   //Creates the 2D image that's going to contain our shadowmap.
22   glTexImage2D( GL_TEXTURE_2D,      // The target Texture
23           0,            // The texture level
24           GL_DEPTH_COMPONENT, // Internal Format
25           shadowMapWidth,     // Width
26           shadowMapHeight,    // Height
27           0,          // this value must be zero
28           GL_DEPTH_COMPONENT, // Format of the pixel data
29           GL_UNSIGNED_BYTE,   // data-type for the pixel data
30           0);         // Pointer to the image data in the memory
31
32   // Binds the image to our texture
33   glBindTexture(GL_TEXTURE_2D, 0);
34
35   // Creates a the framebuffer that is going to be used to generate
        our shadowmap.
36   glGenFramebuffersEXT(1, &fboId);
37   glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fboId);
38
39   // Instruct openGL that we won't bind a color texture with the
        currently binded FBO
```

```
40    glDrawBuffer(GL_NONE);
41    glReadBuffer(GL_NONE);
42
43    // Attach the texture to FBO depth attachment point
44    glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
45                  GL_DEPTH_ATTACHMENT_EXT,
46                  GL_TEXTURE_2D,
47                  depthTextureId,
48                  0);
49
50    // Check FBO status
51    status = glCheckFramebufferStatusEXT(GL_FRAMEBUFFER_EXT);
52    if(status != GL_FRAMEBUFFER_COMPLETE_EXT){
53      std::cout << "GL_FRAMEBUFFER_COMPLETE_EXT failed, CANNOT use
          FBO\n";
54      exit(-1);
55    }
56
57    // Switch back to window-system-provided framebuffer
58    glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);
59  }
```

Denne funktion laver først og fremmest en texture og sætter den til at være en 2DTexture. Derefter laver man et 2D billede som vi sætter til, kun at indeholde dybde data, hvorefter den bliver bundet til vores 2D texture. Basalt set betyder det at vores texture nu er et 2D billede, som i stedet for farve-værdier i hver pixel, indeholder dybdeværdier. Vi sætter derefter vores texture til blive brugt af vores FBO (som blev oprettet tidligere), og binder billedet til dybde laget af FBO'en (det vil sige den del af FBO'en der indeholder alle dybdeværdierne). Dette betyder at når der bliver udregnet og gemt dybdeværdier i vores FBO, vil de blive gemt i vores texture.

6.1.3 SetupTextureMatrix

For at man kan slå et givent verdenskoordinat op i vores dybde map, er det vigtigt at finde ud af hvordan man transformerer det til et koordinat i vores shadowmap. Dette gør man ved at gemme transformation matrixen, som bliver brugt når vi tegner scenen fra lyset af. Ved at gange denne matrix med et verdenskoordinat får man punktets position set fra lyset af. Denne matrix blive gemt som en texture matrix (en matrix der normalt bruges til at transformere texture koordinater).

```
1   void ShadowMapRendere::SetupTextureMatrix(void) {
2     float modelView[16];
3     float projection[16];
4
5     // This matrix is used to convert our depth matrix (which is
          homogeneous ([-1, 1]) into
6     // Texture coordinates which is in the range [0, 1]
7     const GLFloat bias[16] =
8     {
9         0.5, 0.0, 0.0, 0.0,
10        0.0, 0.5, 0.0, 0.0,
11        0.0, 0.0, 0.5, 0.0,
12        0.5, 0.5, 0.5, 1.0
13    };
14
```

```
15    // Grab modelview and transformation matrices
16    glGetFloatv(GL_MODELVIEW_MATRIX, modelView);
17    glGetFloatv(GL_PROJECTION_MATRIX, projection);
18
19    glMatrixMode(GL_TEXTURE);
20    glActiveTextureARB(GL_TEXTURE7);
21
22    //Loads the identity matrix into GL_TEXTURE7
23    glLoadIdentity();
24
25    glLoadMatrixf(bias);
26
27    // concatating all matrice into one.
28    glMultMatrixf(projection);
29    glMultMatrixf(modelView);
30
31    // Go back to normal matrix mode
32    glMatrixMode(GL_MODELVIEW);
33 }
```

Denne funktion gemmer den transformationsmatrix vi skal bruge, til at transformere punkter op i lyset, i GL_TEXTURE7. Det mest bemærkelsesværdige ved denne funktion, er "bias matrixen" (ikke at forveksle med biasen beskrevet i teorien). Hvis vi på normal vis gemmer vores modelview-projection matrix fra lyset og ganger den på et koordinat, vil man ende med punkter i intervallet -1 *til* 1. Dette er et problem da vi skal bruge koordinater fra 0 *til* 1, for at slå op i vores shadowmap. For at konvertere en 4-vektor med koordinater i intervallet -1 *til* 1 til en 4-vektor med intervallet 0 *til* 1 ganger vi med bias-matrixen. Man kunne også uden større problemer tage højde for dette i shaderne i stedet.

6.2 Shadowmapping med bias

Shadowmapping med bias gør, som nævnt tidligere, brug af en bias konstant for at tage højde for moire's patterns. Denne bias er normalt meget lav, og for at få gode resultater skal den også være varierende alt efter vinklen et punkts normal har i forhold til lyset.
Når en scene bliver renderet med vores shadowmap-rendere (som gør brug af bias), bliver det gjort således:

```
1    // Use the FBO generated earlier (by InitShadowFBO) to render our
         shadowmap onto
2    glBindFramebufferEXT(GL_FRAMEBUFFER_EXT,this->fboId);
3
4    // Use the standard rendering programs
5    glUseProgramObjectARB(0);
6
7    // Sets the view port of the current fbo, to be the size of our
         shadow map.
8    glViewport(0,0,DEFAULT_WIDTH * SHADOW_MAP_RATIO,DEFAULT_HEIGHT *
         SHADOW_MAP_RATIO);
9
10   // Clear previous frame values
11   glClear(GL_DEPTH_BUFFER_BIT);
12
13   //Disable drawing to the color buffer
```

```
14    glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);
15
16    // Transforms the camera to the lightsource
17    // LightPos being the world-coordinates of the light
18    // LightRot being the direction of the light
19    this->TransformCamera(lightPos, lightRot, scene);
20
21    // Culling switching, rendering only backface, this is done to
          avoid self-shadowing
22    this->DrawScene(scene);
23
24    // Saves the transformation matrix
25    this->SetupTextureMatrix();
26
27    // Use the original framebuffer FBO
28    glBindFramebufferEXT(GL_FRAMEBUFFER_EXT,0);
29
30    // Sets the viewport to the default values
31    glViewport(0, 0, DEFAULT_WIDTH,DEFAULT_HEIGHT);
32
33    //Enabling color write again
34    glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
35
36    // Clear previous frame values
37    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
38
39    //Using the shadow shader
40    glUseProgramObjectARB(this->shadowShaderId);
41
42    //Loads 7 into the shadowMapUniform value in our shaders
43    glUniform1iARB(this->shadowMapUniform, 7);
44
45    // Activates GL_TEXTURE7 and binds our shadowmap to it.
46    glActiveTextureARB(GL_TEXTURE7);
47    glBindTexture(GL_TEXTURE_2D, depthTextureId);
48
49    // Transform the camera back to the original position
50    // cameraPos being the cameras Position (world cooridinates)
51    // cameraRot being the cameras direction
52    this->TransformCamera(cameraPos, cameraRot, scene);
53
54    // Draws the scene
55    this->DrawScene(scene);
```

Koden ovenfor følger den oprindelige algoritme. Det er dog ikke nødvendigvis tydeligt hvornår skyggerne bliver lavet. Shadowmappet bliver genereret ved *DrawScene*. Selve skyggerne bliver tegnet samtidig med alt andet, når *DrawScene* bliver kaldt for anden gange. Om et punkt ligger i skygge eller ej, bliver udregnet i vores vertex og fragment shadere.

6.2.1 Vertex shaderen

Vores vertex shader ser således ud:

```
1  // Used for shadow lookup
2  varying vec4 ShadowCoord;
3  varying vec3 N;
4  varying vec3 v;
5
6  void main()
7  {
8    // our vertex in out modelview (seen from the camerea)
9    v = vec3(gl_ModelViewMatrix * gl_Vertex);
10   N = normalize(gl_NormalMatrix * gl_Normal);
11
12   // The cooridinate of the vertex from our light source
13   // gl_Vertex is our untransformed vertex
14   ShadowCoord = gl_TextureMatrix[7] * gl_Vertex;
15
16   // The position of the vertex after all transformation has been
       applied
17   gl_Position = gl_ProjectionMatrix * gl_ModelViewMatrix * gl_Vertex;
18
19   // The color of Front face
20   gl_FrontColor = gl_Color;
21 }
```

Koden for vores vertex-shader er rimelig lige til. Som man kan se så gemmer vi en vektor, v, som indeholder den nuværende vertex i vores ModelView. Normalen, N, for den nuværende vertex bliver også gemt. Disse bliver brugt til at justere den givne bias konstant (dette bliver gjort i fragment shaderen).

Som nævnt i teorien er det vigtigt at kunne transformere et givent punkt op i lyset. Da *gl_Vertex* er verdens koordinatet for det nuværende punkt, behøver vi ikke først at transformere punktet fra kameraet til verdens koordinater. ShadowCoord er den nuværende vertex position transformeret til lyset.

6.2.2 Fragment shaderen

Det intressante sker i fragment shaderen:

```
1  uniform sampler2D ShadowMap;
2  uniform float b;
3  varying vec4 ShadowCoord;
4  varying vec3 N;
5  varying vec3 v;
6
7  void main()
8  {
9    // The direction from the vertex (v) to the light
10   vec3 L;
11
12   // The vector containing the shadow color.
13   vec4 shadow;
```

```
14
15    // The coordinates in our shadowmap.
16    vec4 shadowCoordinateWdivide;
17
18    // The distance from the current fragment to the light
19    float distanceFromLight;
20
21    float bias; // The bias to be applied
22    float cosTheta; // the dot product of N and L
23
24    // L is a normalized vector pointing from out vector to out
          lightsource
25    L = normalize(gl_LightSource[0].position.xyz - v);
26
27    //The angle between the the normal of the vertex and the vector
          pointing toward the light.
28    cosTheta = clamp( dot( N,L ), 0,1 );
29
30    // Bias is based on the slope of the surface
31    bias = b * tan(acos(cosTheta));
32    bias = clamp(bias, 0, 0.005);
33
34    shadow = vec4(1.0, 1.0, 1.0, 1.0);
35
36    shadowCoordinateWdivide = ShadowCoord / ShadowCoord.w ;
37    distanceFromLight =
          texture2D(ShadowMap,shadowCoordinateWdivide.st).z;
38
39    if (distanceFromLight < shadowCoordinateWdivide.z - bias){
40      shadow[0] = 0.5;
41      shadow[1] = 0.5;
42      shadow[2] = 0.5;
43    }
44
45    //The final color
46    gl_FragColor = gl_FrontLightModelProduct.sceneColor + Iamb + Idiff
          * shadow + Ispec * shadow;
47  }
```

Det skal endnu engang bemærkes at alt der har med vores lysmodel at gøre, er blevet skåret ud af eksemplet, da dette er irrelevant for forklaringen. Det første der bliver udregnet her er størrelsen på biasen i det givne punkt. Dette bliver gjort ved først at tage vinklen mellem en vektor der peger mod lyskilden og normalen for den givne vektor. Ud fra dette bliver vores bias udregnet. Værdien b bliver sat ude i vores rendere-klasse og ikke i selve shaderene. Vi begrænser biasen til intervallet 0 *til* 0.005, da det generelt var det der gav det bedste resultat. Vi slår derefter vores punkt op i shadowmappet og sætter *distanceFromLight* til at være denne værdi. Hvis afstanden fra vores nuværende punkt til lyset minus vores bias (beskrevet ved: *shadowCoordinateWdivide.z - bias*) er højere end den opslåede værdi, betyder det at det nuværende punkt ligger i skygge.

Iamb, *Idiff* og *Ispec* er alle værdier, som er blevet udregnet i vores lysmodel. De er henholdsvis tilskudet fra ambient, diffus og reflekterende lys.

I koden kan man se at biasen er afhængig af en konstant b. Her bliver man nød til at prøve sig frem for at finde et såkaldt "sweetspot", for ens scener. En for høj bias giver, som nævnt i teori-delen, peter-panning, i mens en forlav bias ikke vil være nok til at fjerne moire's pattern.

6.3 Shadowmapping med culling

I vores implementation af shadowmapping som bruger culling, er der, som nævnt tidligere, ikke nogen forskel i måden shadowmappet bliver opsat på. Der er dog en lille forskel i både shaderne og måden scenerne bliver renderet på. Scenerne bliver renderet på følgende måde, når man gør brug af culling:

```
1  glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, this->fboId);  //Rendering
       offscreen
2
3  //Use the standard rendering programs
4  glUseProgramObjectARB(0);
5
6  //Sets the view port of the current fbo, to be the size of our
       shadow map.
7  glViewport(0,0,DEFAULT_WIDTH * SHADOW_MAP_RATIO,DEFAULT_HEIGHT*
       SHADOW_MAP_RATIO);
8
9  // Clear previous frame values
10 glClear(GL_DEPTH_BUFFER_BIT);
11
12 //Disable drawing to the color buffer
13 glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);
14
15
16 // Transforms the camera to the lightsource
17 // LightPos being the world-coordinates of the light
18 // LightRot being the direction of the light
19 this->TransformCamera(lightPos, lightRot, scene);
20
21 // Enables face culling
22 glEnable(GL_CULL_FACE);
23
24 // Culling switching, rendering only backface, this is done to
       avoid self-shadowing
25 glCullFace(GL_FRONT);
26 this->DrawScene(scene);
27
28 //Save modelview/projection matrice into texture7, also add a biais
29 this->SetupTextureMatrix();
30
31 // Now rendering from the camera POV, using the FBO to generate
       shadows
32 glBindFramebufferEXT(GL_FRAMEBUFFER_EXT,0);
33
34 glViewport(0, 0, DEFAULT_WIDTH,DEFAULT_HEIGHT);
35
36 //Enabling color write (previously disabled for light POV z-buffer
       rendering)
```

```
37   glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
38
39   // Clear previous frame values
40   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
41
42   //Using the shadow shader
43   glUseProgramObjectARB(shadowShaderId);
44   glUniform1iARB(shadowMapUniform, 7);
45   glActiveTextureARB(GL_TEXTURE7);
46   glBindTexture(GL_TEXTURE_2D, depthTextureId);
47
48   // Transform the camera back to the original position
49   // cameraPos being the cameras Position (world cooridinates)
50   // cameraRot being the cameras direction
51   this->TransformCamera(cameraPos, cameraRot, scene);
52
53   glCullFace(GL_BACK);
54   glColor3f(0.5, 0.5, 0.5);
55   this->DrawScene(scene);
```

Som man kan se så minder implementationen meget om shadowmapping med bias. Den eneste forskel er:

```
1   glCullFace(GL_FRONT);
2   this->DrawScene(scene);
```

og

```
1   glCullFace(GL_BACK);
2   glColor3f(0.5, 0.5, 0.5);
3   this->DrawScene(scene);
```

Som nævnt i teori delen så flytter culling metoden problemet med moire-pattern til de steder hvor figurene kaster skygger på sig selv. Dette problem bliver løst i shaderne.

6.3.1 Vertex shaderen

Vertex shaderen for shadowmapping med culling, er identisk til vertex shaderen for shadow mapping med bias.

6.3.2 Fragment shaderen

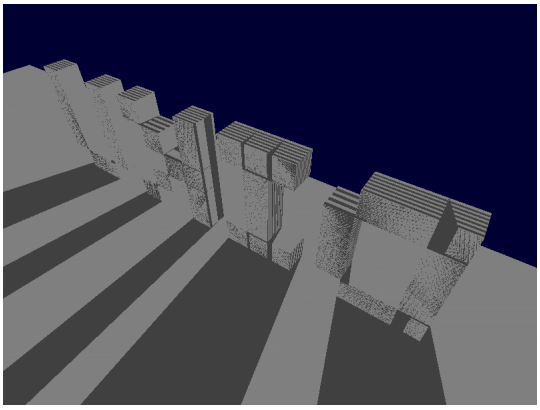
```
1   vec3 L; // The direction from the vertex (v) to the light
2
3   // The vector containing the shadow color.
4   vec4 shadow;
5
6   // The coordinates in our shadowmap.
7   vec4 shadowCoordinateWdivide;
8
9   // The distance from the current fragment to the light
```

```
10    float distanceFromLight;
11
12    L = normalize(gl_LightSource[0].position.xyz - v);
13
14    //The shadow color is set to black
15    shadow = vec4(1.0, 1.0, 1.0, 1.0);
16
17    shadowCoordinateWdivide = ShadowCoord / ShadowCoord.w ;
18    distanceFromLight =
          texture2D(ShadowMap,shadowCoordinateWdivide.st).z;
19
20    if (distanceFromLight < shadowCoordinateWdivide.z || dot(N,L) <=
          0.0){
21      shadow[0] = 0.5;
22      shadow[1] = 0.5;
23      shadow[2] = 0.5;
24    }
25
26    //The final color
27    gl_FragColor = gl_FrontLightModelProduct.sceneColor + Iamb + Idiff
          * shadow + Ispec * shadow;
28  }
```

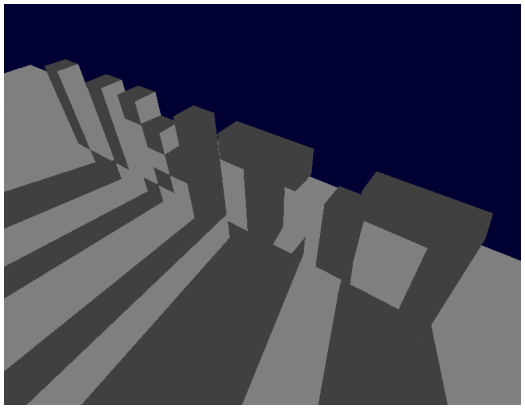
Endnu engang minder de to implementationer meget om hinanden. Den store forskel i dette tilfælde, er at der ikke bliver gjort brug af bias. Tilgengæld laver vi et nyt tjek for se om et punkt ligger i skygge.

```
1     if (distanceFromLight < shadowCoordinateWdivide.z || dot(N,L) <=
          0.0){
2   ...
3     }
```

Ud over at vi sammenligner dybderne fra vores shadowmap og det nuværende punkt i forhold til lyset, tjekker vi også om prikproduktet mellem punktets normal og en vektor, der peger fra punktet mod lyset er negativt. (det vil sige om vinklen mellem N og L er over 90). Hvis dette er tilfældet, konkludere vi at den overflade, som det nuværende punkt er en del af, vender væk fra lyset og dermed ligger i skygge. Hvis vi ikke har dette tjek, opstår der moires-pattern i skyggerne, som man kan se på følgende billeder:



(a) *Her kan man se at de er moire's pattern der hvor der burde være skygge på figurene selv. Selve de skygger som figurene kaster er der ingen problemer med.*



(b) *Her kan man se at moire's pattern er forsvundet efter vi har tilføjet tjeket på normalerne.*

Det skal siges i billederne ovenfor har vi slået vores lysmodel fra, da visse udregninger i modellen gjorde, at man ikke kunne se det. Det var hovedsagligt udregningerne af det diffuse lystilskud, der forsagede dette da den ville være 0 de steder hvor $dot(N, L) <= 0.0$ er sandt.

Hvis man ikke vil til at tjekke på normalerne, kan man også gøre brug af bias i stedet. Det vil sige man både bruger culling og tilføjer en bias.

7 Sammenligning af algoritmerne

Vi er kommet frem til af begge af algoritmerne har visse fordele og ulemper. Før vi sammenligner algoritmerne er det vigtigt at gennemgå hvad disse er.

Fordele og ulemper for skyggevolumen algoritmen:

- Fordele
  - Implementation er ikke særlig avanceret.
  - Omni-directional lys.
  - Kræver ikke specialiseret shadere
- Ulemper
  - Kan blive meget CPU intens hvis der er mange polygoner
  - Skal kende til alt geometri i scenen.
  - Kræver man laver skyggevolumener for scenen 2 gange.
  - Kræver man laver udregningerne for hver lyskilde.

Fordele og ulemper for shadowmapping algoritmen:

- Fordele
  - Uafhængig af scenens geometri
  - Størstedelen af udregningerne bliver udført på grafikkortet.
  - Kan lettere skaleres, da kvaliteten afhænger af variabler man kan indstille på.
- Ulemper
  - Kræver specialiseret shadere.
  - Ikke omni-directional fra starten.
  - Kræver bredt kendskab til OpenGL's API.
  - Kan kræve en masse fin justeringer.
  - Man skal lave et shadow map pr. lyskilde
  - Kræver man skal tegne scenens geometri mindst 2 gange.

Shadowmapping-algoritmen er god hvis man gerne vil have størstedelen af udregningerne over på grafikkortet. Desuden er den ikke direkte afhængig af en scenes kompleksitet. Den er generelt god at bruge, hvis man har kompleks geometri eller dynamisk lys, da der ikke er nogen ekstra udregninger i forbindelse med ændringer i en scene.

Tilgengæld kræver shadowmapping en del forskellige funktionaliteter af grafikkortet. Man skal først og fremmest sørge for at grafikkortet er kraftigt nok. Dette skyldes blandt andet at shadowmapping er, som nævnt før, meget afhængig af opløsningen på scenen, da opløsningen på shadowmappet baseres ud fra den. Jo højere opløsning scenen er, jo mere bliver der krævet af grafikkortet.
Derudover skal grafikkortet understøtte de funktioner der bliver brugt.

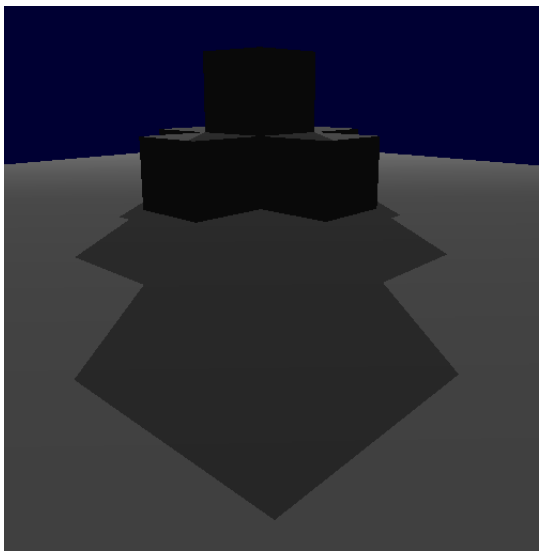
Skyggevolumen algoritmen er god hvis man har nok CPU kræft, og hvis man ikke vil flytte for mange udregninger til grafikkortet. Man skal heller ikke lave nogle ekstra tiltag for at få omnidirectionelt lys. Derudover så er selve implementationen rimelig simpel, da det i modsætning til

shadowmapping, ikke kræver at man laver specialiserede shadere. Den er dog ikke særlig god til dynamisk lys, da man skal lave en masse udregninger, hver gang enten lyset eller geometrien i en scene har ændret sig.
Den kræver, imodsætning til shadowmapping, dog ikke nogle fin justeringen (så som brug af bias i shadowmapping). Den er heller ikke særlig afhængig af hvad grafikkortet understøtter.

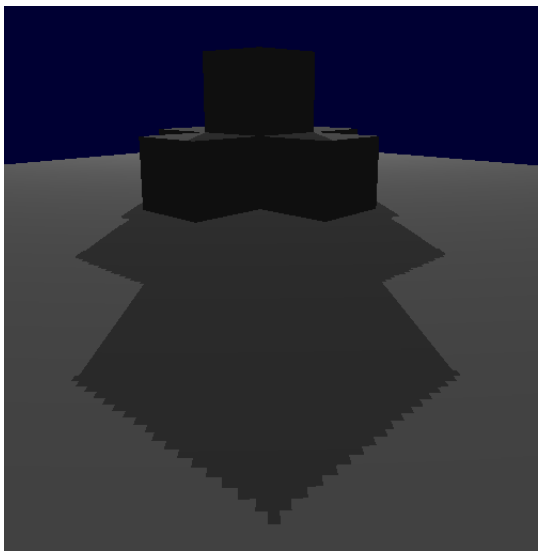
Alt i alt så er shadowmapping god hvis man går efter at få skyggerne hurtigt. Skyggevolumen algoritmen er tilgengæld god, hvis man går efter kvalitet frem for hastighed, hvilket kan ses i følgende afsnit.

7.1 Sammenligning af skyggerne

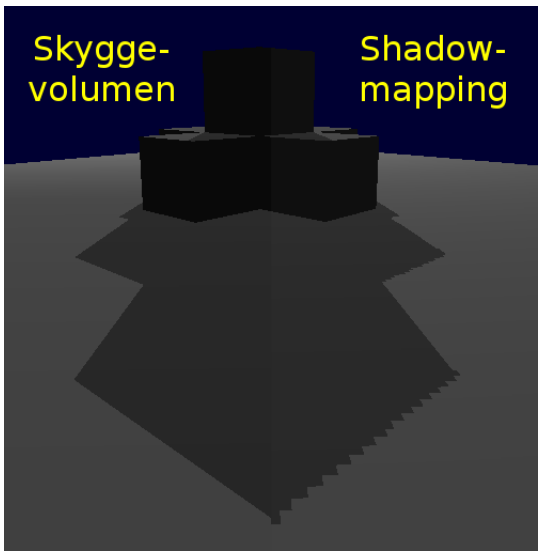
Ud over fordele og ulemper ved algoritmerne, er det også vigtigt at huske at de to algoritmer producerer forskellige resultater. På afstand ser skyggerne meget ens ud, men når man går lidt tættere på er forskellen tydelig.



(a) *Her bliver skyggevolumen algoritmen brugt*



(b) *Her bliver shadow mapping algoritmen brugt*



(c) *En side om side sammenligning af de resulterende skygger for de 2 algoritmer*

Bemærk at skyggerne lavet med shadow mapping kommer til at ligne mere og mere skyggerne lavet af skyggevolumener, jo længere væk fra kameraet skyggen er.

III Afslutning

8 Forbedringsforslag

I denne sektion vil vi beskrive de forbedringsforslag vi har til vores program. Desuden vil vi også kort komme ind på en metode til at kombinerer de 2 skygge algoritmer, som vi har forklaret igennem rapporten, for at effektivisere genereringen af skyggerne. Denne metode er baseret på en artikel skrevet af Michael D. McCool i år 2000 [McC00].

Metoden går kort sagt ud på at man bruger shadow mapping algoritmen til at lave et shadow map. Ud fra dette shadow map laves silhuetter, som så bruges til skyggevolumenerne. Med denne fremgangsmetode behøver man derfor ikke kende til geometrien i scenen og man behøver heller ikke at lave fin justeringer i shadow mapping algoritmen for at få pæne skygger.

8.1 Forbedringsforslag til vores implementationen

Selvom begge vores implementationer af algoritmerne fungerer, er der stadig en del der kan forbedres. Vi vil i dette afsnit gennemgå nogle af de forbedringer man kunne tilføje til vores implementation af algoritmerne, for at gøre dem bedre og hurtigere. Disse forbedringer nåede ikke med i det færdige produkt, enten på grund af mangel på tid, eller også var dette simpelthen ikke en del af vores mål med dette projekt.

8.1.1 Optimering af Silhuetter

Vores implementation af skyggevolumener med silhouetter, er for eksempel meget langsom i forhold til hvad den kunne være. Dette skyldes ikke mindst, at vi skal gennemløbe alle polygoner i alle objekter flere gange for først at fjerne dublet polygoner (det vil sige polygoner der deles af flere objekter) og derefter udregner vil silhouetterne. Ved at bruge en bedre data struktur til opbevaring af kanterne på polygoner, ville vi kunne lave silhuetterne meget hurtigere.

8.1.2 OpenGL transformationer

På nuværende tidspunkt kan vi ikke bruge OpenGL's rotation, translation og skalerings funktioner sammen med skyggevolumener, da skyggevolumener kræver at vi hele tiden har styr på verdenskoordinaterne for geometrien i vores scener. For at løse dette problem, kan vi enten implementerer vores egne transformations funktioner eller hente transformations-matricerne fra OpenGL og så gange dem på punkterne når det er relevant. Problemet med begge løsninger er, at det giver endnu mere arbejde for CPUen og at vi skal gemme transformationsmatricen for hvert punkt.

8.1.3 Shadowmap fitting

Et af de problemer der er med shadowmapping er, at sørge for at den dække så meget at ens scene som muligt. Den del som shadowmappet dækker er begrænset af hvad kameraet kan se fra lyset. I vores implementation har vi ikke taget højde for dette. Vi har manuelt placeret og valgt hvilken vej lyset peger.

Man skal helst sørge for at alt hvad kameraet kan se bliver dækket af shadowmappet. At gøre dette kan være meget beregningskrævende, da det kræver at man analyserer scenens geometri og peger lysets kamera derhen hvor shadowmappet dækker mest muligt af hvad kameraet kan se.
For at løse dette problem kunne man gøre så lyskilden laver skygger i alle retninger. Altså ved at lave shadowmappet omni-directional (det vil sige at den peger i alle retninger).

8.1.4 Skygger på omni-directional lys med shadowmapping

Vores implementation er på nuværende tidspunkt begrænset til den retning som vores lyskilde peger. Det vil sige at et lys i vores shadowmap ikke danner skygger i alle retninger på alle objekter i en scene, men kun på de objekter der kan ses fra lyset. Dette betyder at hvis der er geometri i ens scene som lyskilden ikke kan se, vil de ikke danne skygger. Dette kan løses ved at man laver et såkaldt cubemap. Det vil sige at man laver et shadowmap for alle sider af shadowmappet, i stedet for blot et enkelt map for den retning lyset peger. Dette betyder dog at hver lyskilde har 6 shadowmaps, i stedet for kun 1. Hvilket samtidig betyder at grafikkortet skal arbejde en del mere.

Det er værd at nævne at skyggevolumen algoritmen ikke har problemer med fitting, eller at danne skygge i alle retninger.

8.1.5 Bløde skygger til begge algoritmer

I begge vores implementationer har vi ikke tilføjet bløde skygger. Hvis vi skulle implementere bløde skygger (som beskrevet i et tidligere afsnit) til skyggevolumen, ville det først og fremmest være en god ide at optimere vores nuværende implementation af silhuetter, som nævnt tidligere. Implementationen af bløde skygger til shadowmapping burde ikke kræve så meget hvis vi blot holder os til PCF-metoden (med enten 2x2 samples eller 3x3 samples). Det skal dog siges, at der er andre måder at lave bløde skygger på som er væsentligt mere avanceret, men også giver et bedre resultat.

9 Konklusion

Efter at have læst om og arbejdet med de to algoritmer i længere tid, er vi kommet frem til at begge algoritmer har både fordele og ulemper. Derudover er der flere forskellige variationer af begge algoritmer, som desuden også har deres egne fordele og ulemper. Vi kan konkludere at shadowmapping kan betale sig at bruge i nogle tilfælde, imens man i andre tilfælde, med fordel, kan bruge skyggevolumener.

Skyggevolumener giver uden tvivl de pæneste hårde skygger af de 2 algoritmer. Den kræver dog at man har styr på verdenskoordinaterne på alt ens geometri, da det er ud fra disse koordinater skyggevolumenerne bliver konstrueret. Desværre sker størstedelen af udregningerne på CPU'en hvilket betyder at den kan blive langsom hvis geometrien er meget kompleks. Man kan dog effektivisere algoritmen ved at bruge silhouetter.

Vi kan dog konkludere, at det er vigtigt at huske at hvis man ikke implementerer silhuetterne korrekt, så kan dette gøre algoritmen endnu langsommere (hvilket desværre var tilfældet for vores implementation).
Hvis man derudover ønsker en robust løsning, skal man bruge Z-fail variationen af algoritmen. Kan man leve med den ikke robuste løsning, kan man få skyggevolumener der er uendelige lange og samtidig også hurtigere at lave.

Vi mener ud fra vores arbejder, at skyggevolumener er bedst hvis man vil have pæne skygger, men da den er meget beregningskrævende kan den bedst betale sig at bruge til pre-rendering, eller hvis man allerede har meget arbejde på grafikkortet og gerne vil flytte arbejde over på CPU'en.

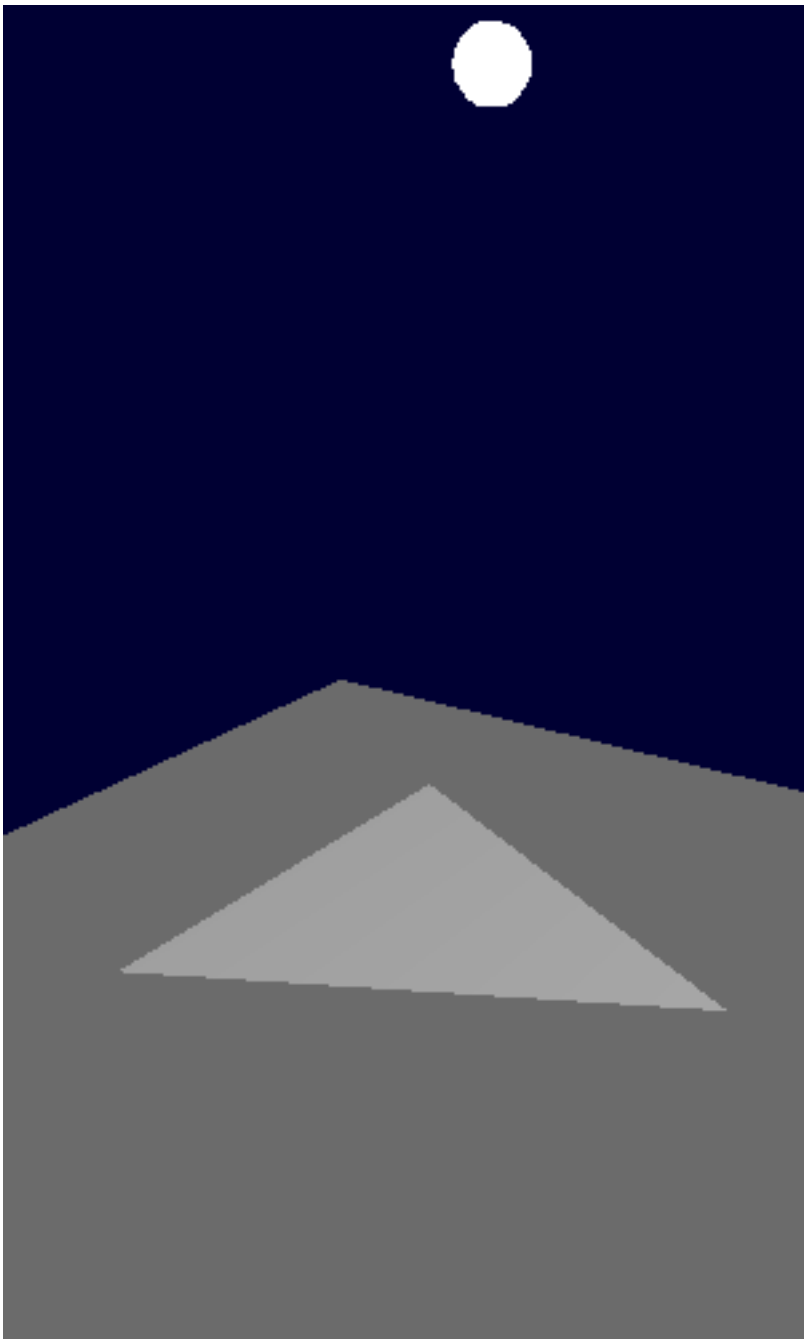
Shadowmapping kan bedst betale sig når man har med dynamisk lys og komplekse scener at gøre, da algoritmen ikke bliver langsommere fordi der for eksempel er flere polygoner på skærmen. Vi kom dog også frem til at shadowmapping ikke giver lige så pæne skygger som skyggevolumener. Dette skyldes at shadowmapping er meget afhængig af opløsningen på shadowmappet.
Hvis man vælger at gøre brug af bias, skal man desuden også tage højde for peter-panning. Løsningen vi fandt på peter-panning betød dog, at vi i de fleste tilfælde, lige så godt kunne bruge culling-metoden og så lade helt hver med at bruge bias, og i stedet tjekke normalernes retning i forhold til lyset.

Vi kan, på nuværende tidspunkt, ikke med sikkerhed bedømme hvilke af de to algoritmer der takler bløde skygger bedst, da vi ikke har beskæftiget os med implementationen af dette. Det er dog værd at nævne, at det er langt lettere at finde metoder til at lave bløde skygger med shadowmapping end med skyggevolumen algoritmen.
Kort sagt: Hvis man vil have hastighed frem for kvalitet skal man bruge shadowmapping. Hvis man derimod protiere kvalitet højere end hastighed kan man med fordel bruge skyggevolumener. Alt i alt kan vi konkludere at hver algoritme giver forskellige slags udfordringer, som man skal tage højde for, og at de hver især kan betale sig at bruge på forskellige tidspunkter. Det er desuden værd at nævne, at der findes flere end disse to algoritmer til at lave skygger, samt mange variationer af de algoritmer vi har beskæftiget os med i dette projekt.

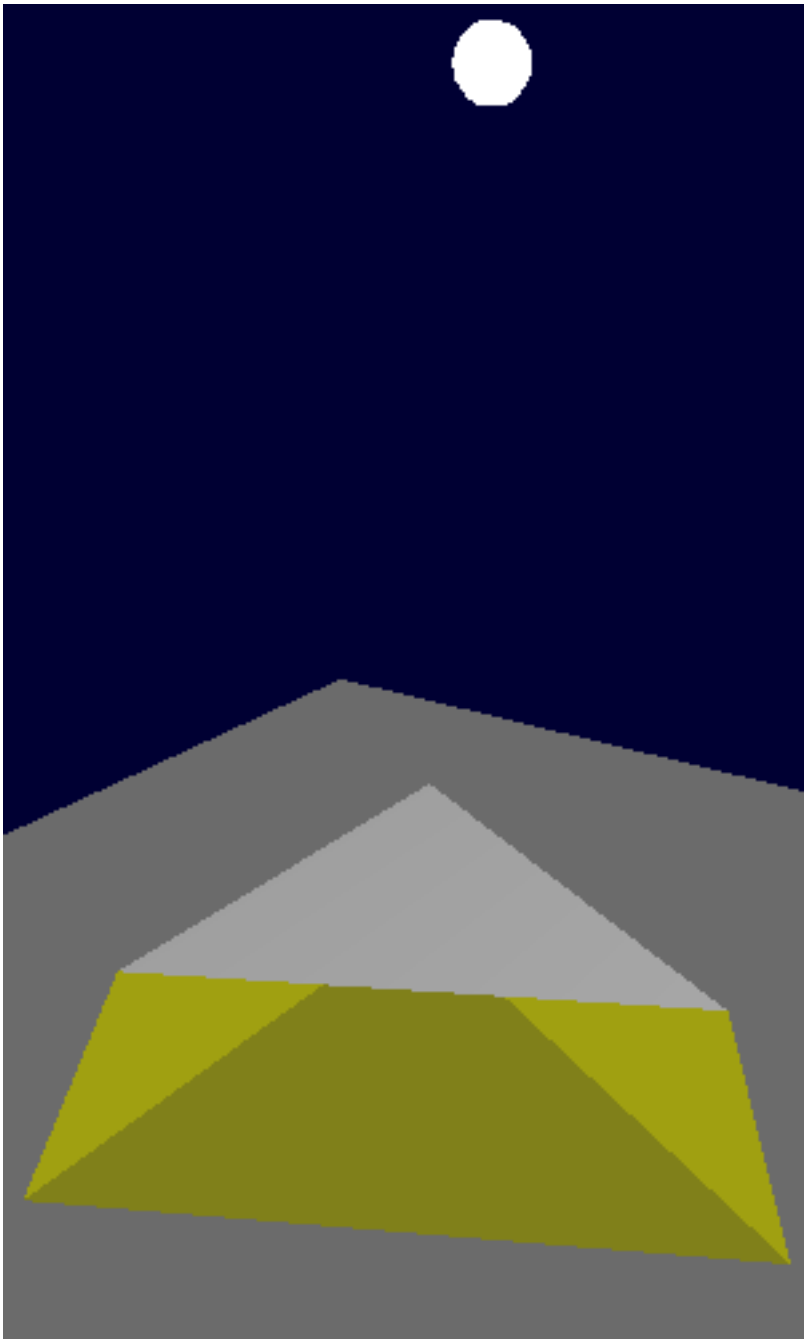
IV Appendiks

A Billeder i stort format

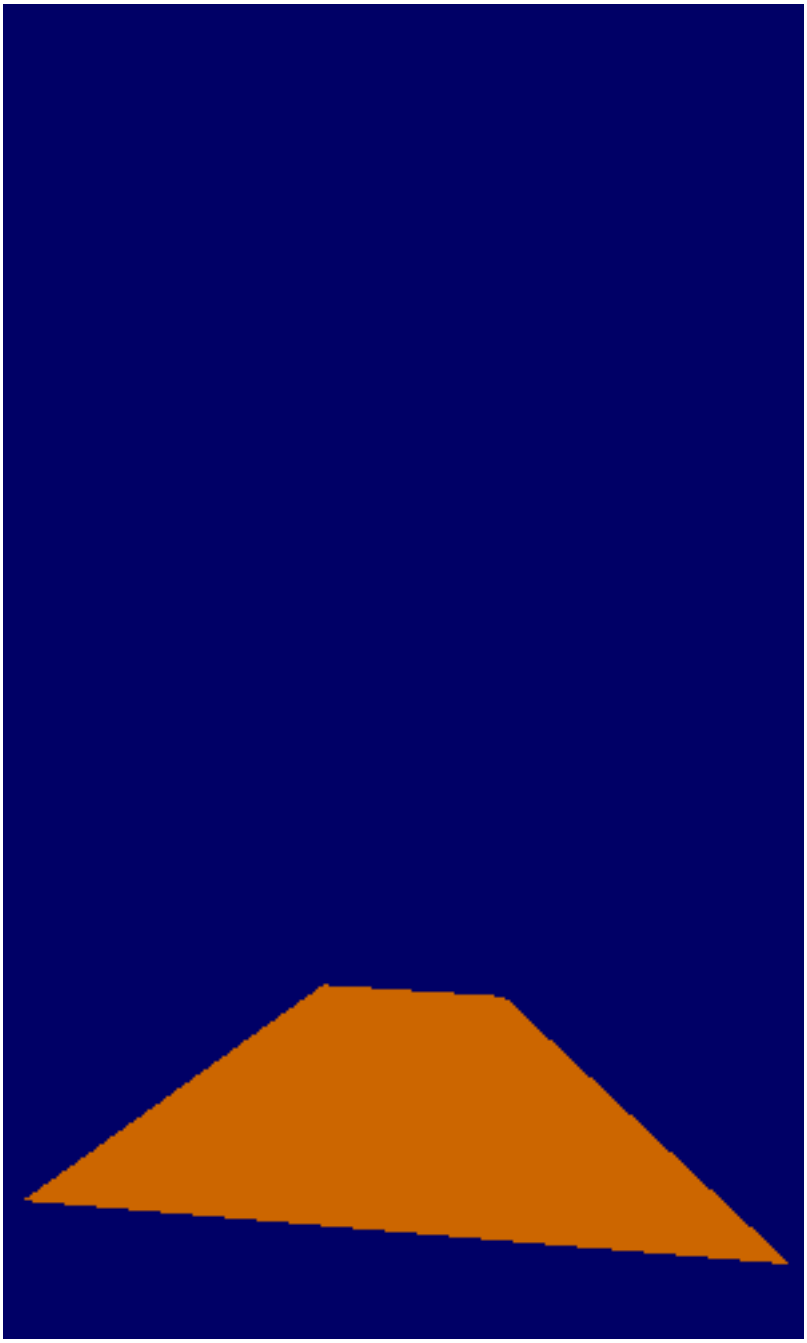
A.1 Billeder til kapitel 2



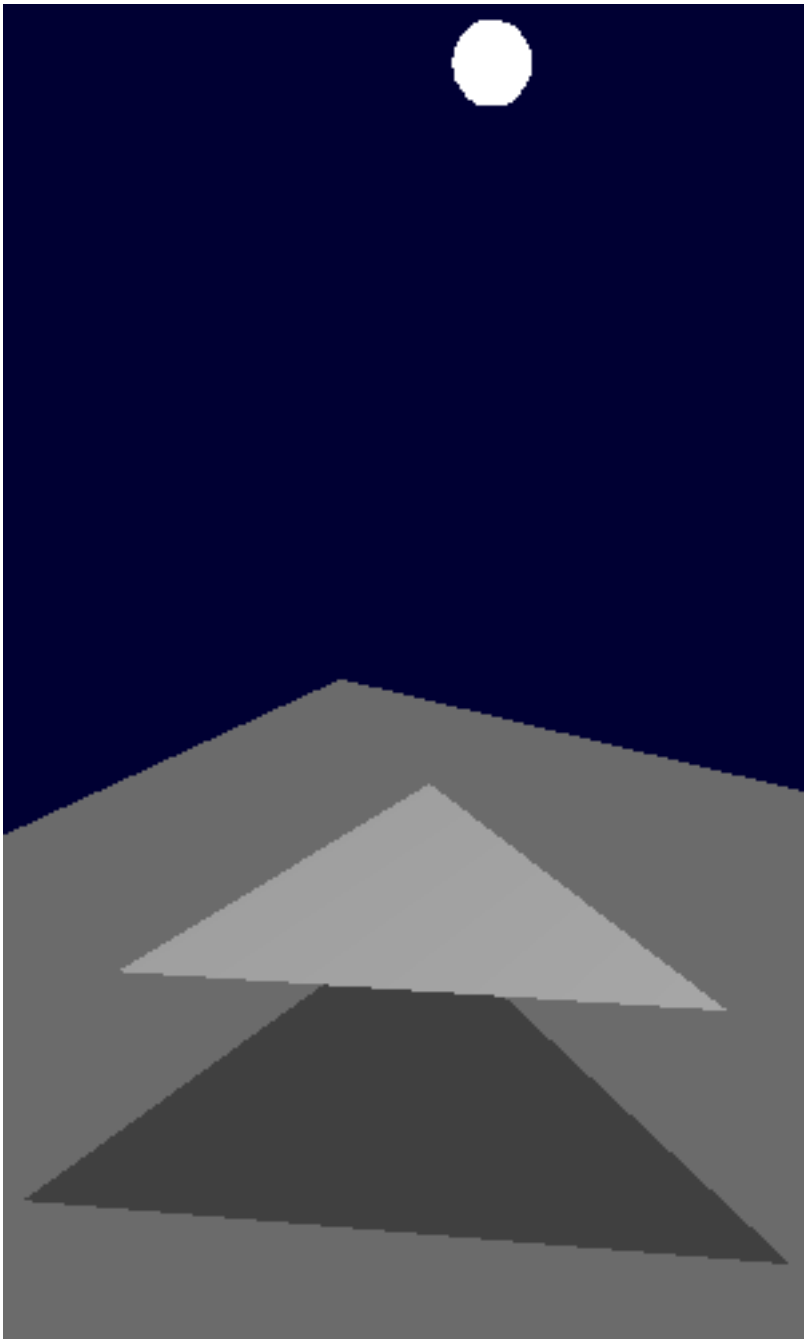
Figur A1: *Dette er vores simple test scene uden skygger.*



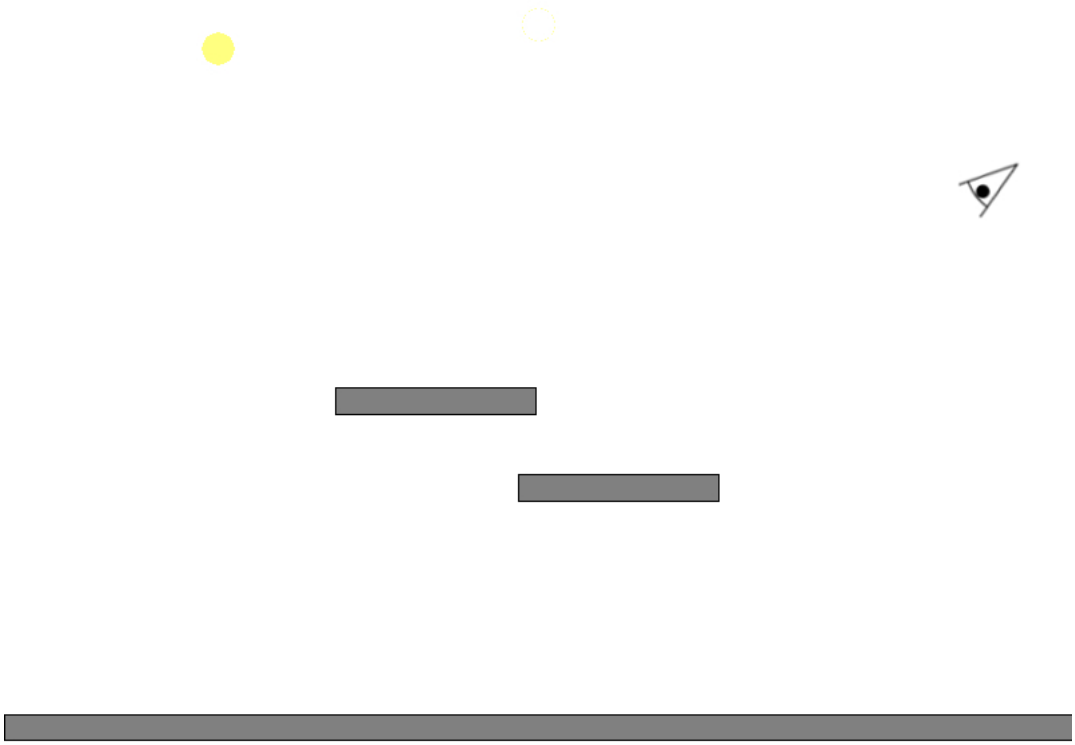
Figur A2: *Her kan man se vores skyggevolumen (den gule figur), som er blevet udregnet ud fra lysets position, alt der ligger inden for dette volumen skal ligge i skygge.*



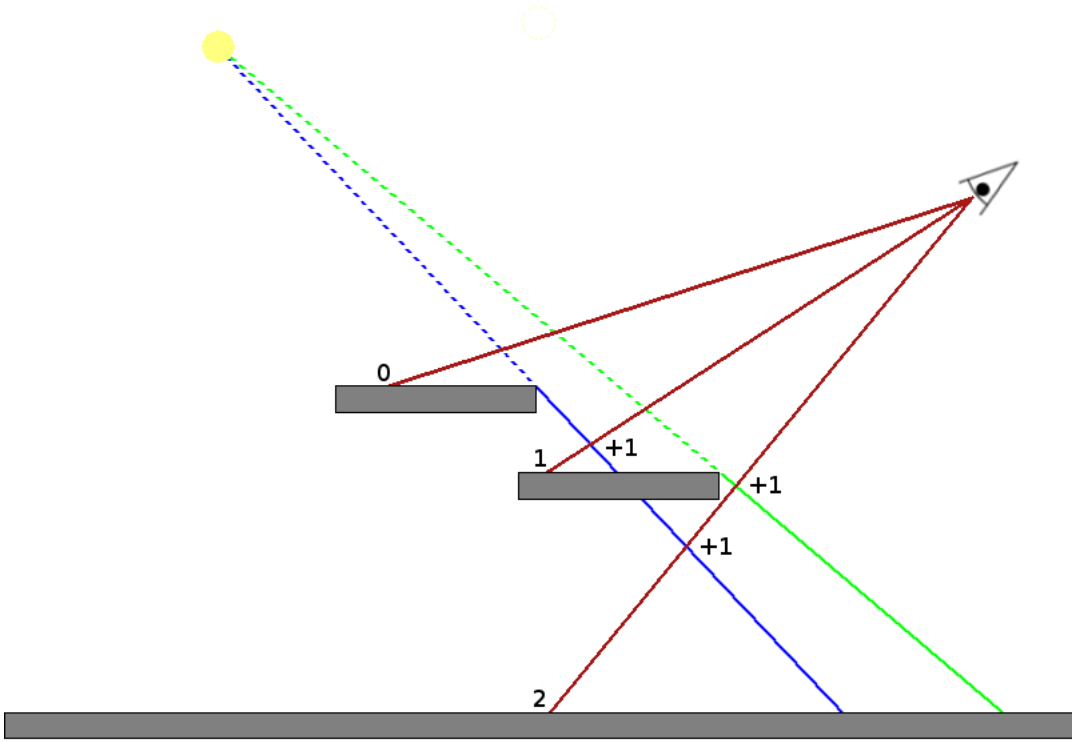
Figur A3: *Her er en visualisering af stencil bufferen for denne scene. De pixels på billedet der er blå, er de pixel hvor stencil bufferens værdi er 0, den orange del er pixels hvor stencil bufferen er over 0 (det vil sige vores skygger).*



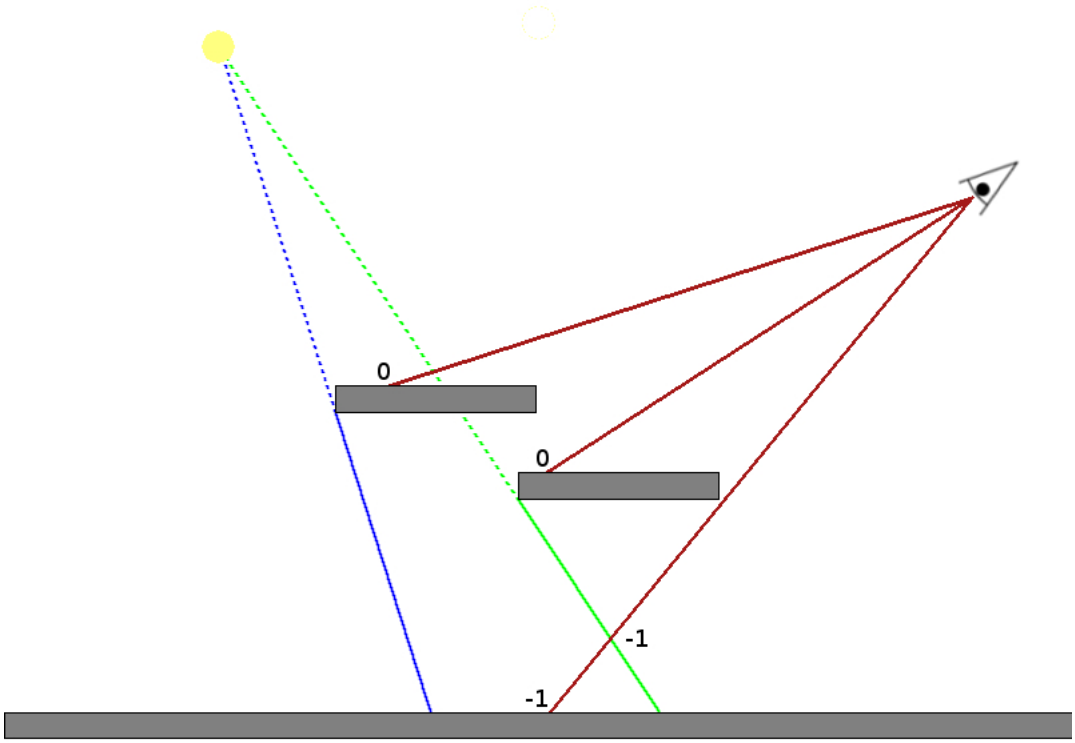
Figur A4: *Her er vores test scene med skygger. Bemærk at skyggerne på dette billede stemmer overens med indholdet af stencil bufferen, se 2.1c.*



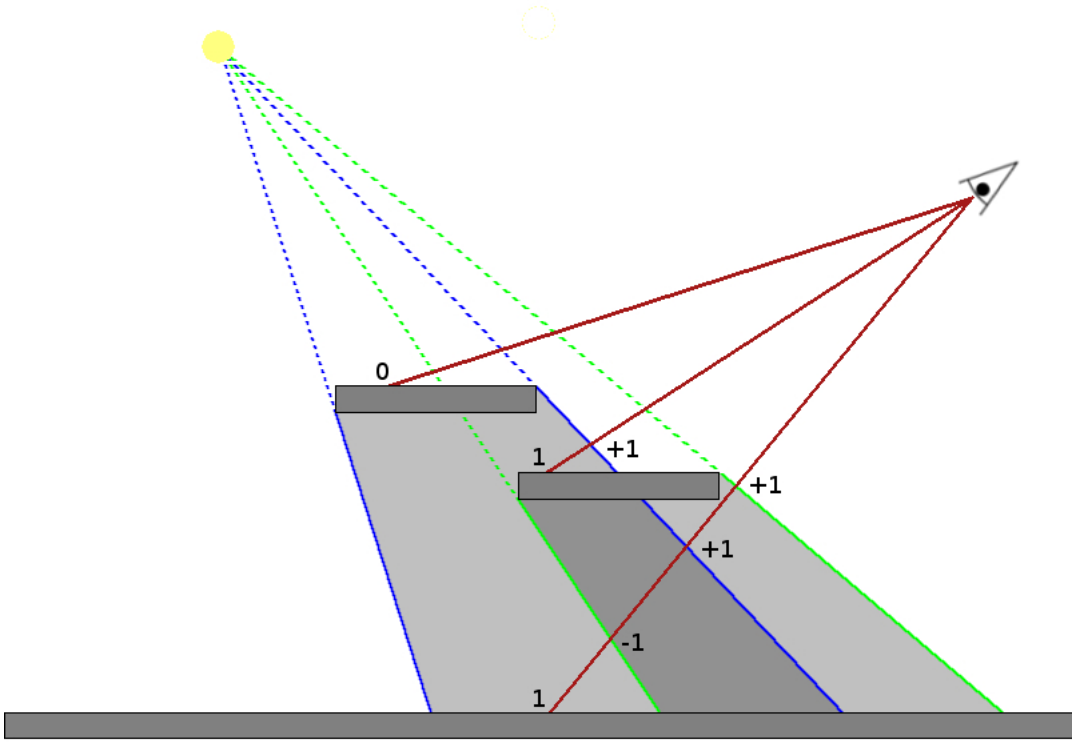
Figur A5: *Denne figur viser scenen uden nogen skygger.*



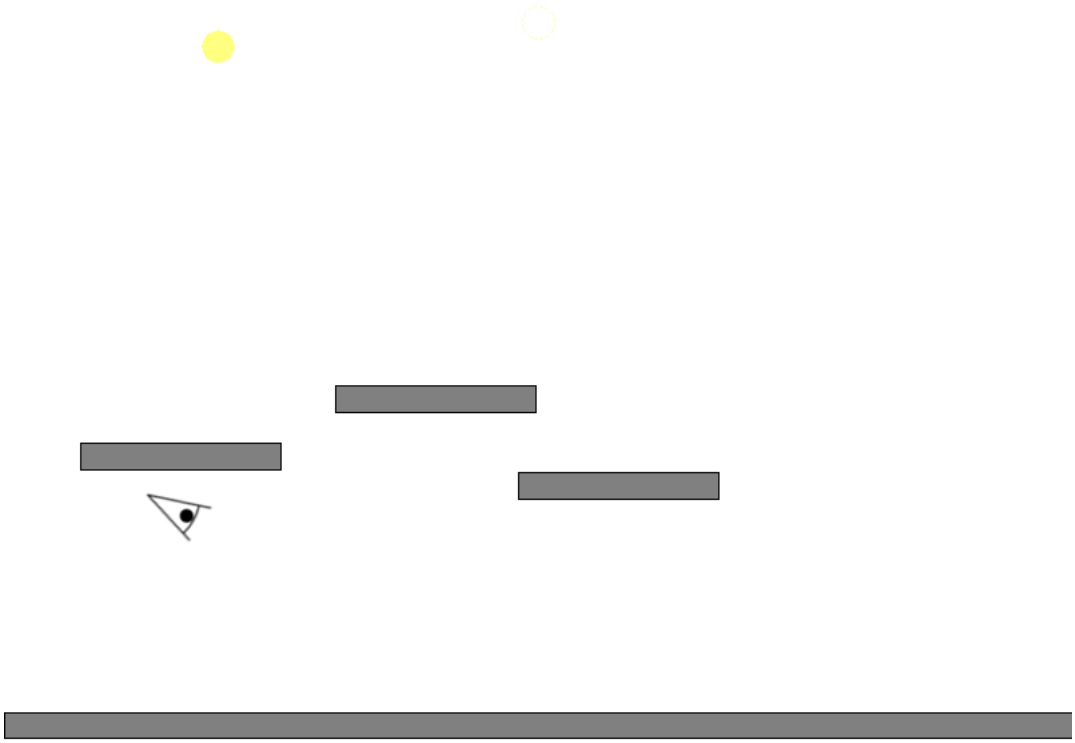
Figur A6: *Denne figur viser den første gennemkørsel af Z-pass algoritmen, hvor der bruges back-face culling. For hver forside af skyggevolumen linjen går ind i tælles værdien 1 op. Bemærk at det kun er de fuldt optrukne streger der angiver et skyggevolumen.*



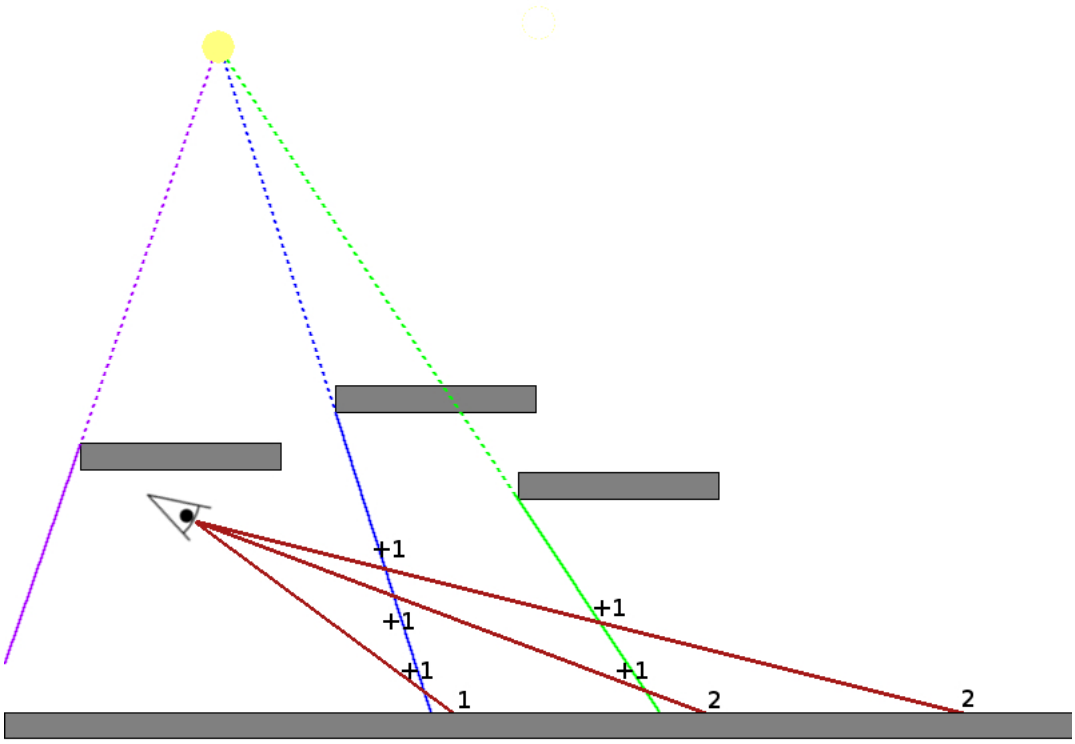
Figur A7: *Denne figur viser anden gennemkørsel af Z-pass algoritmen, hor der bruges front-face culling. Hver bagside af skyggevolumen linjen går igennem tæller værdien 1 ned.*



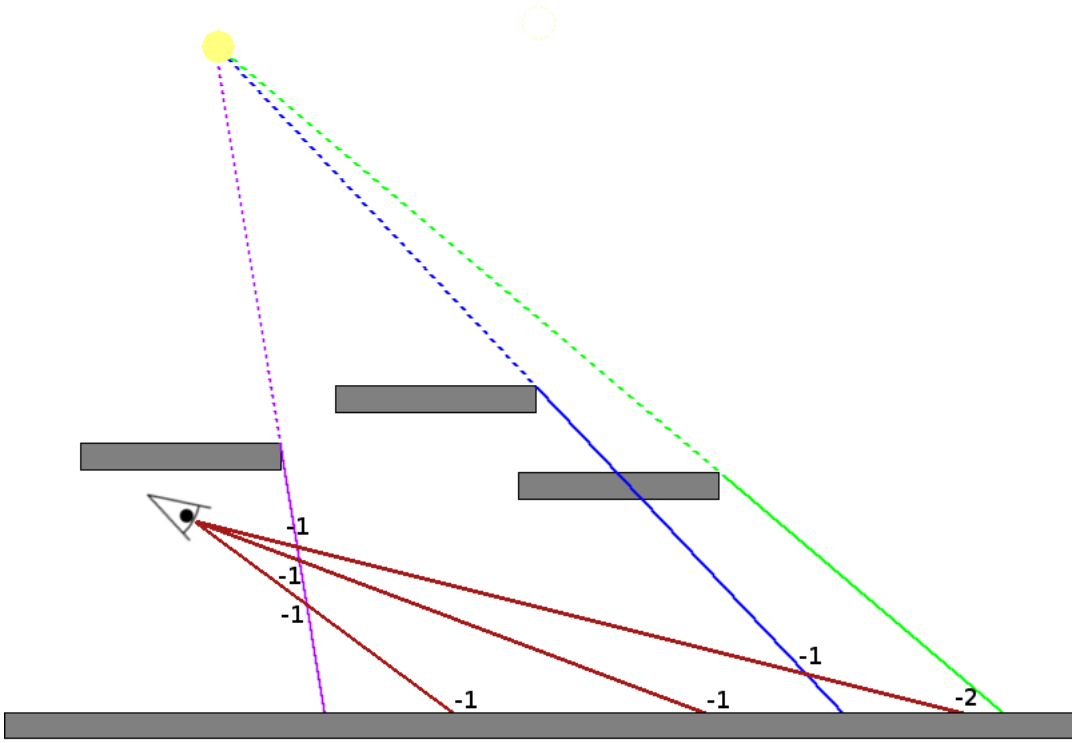
Figur A8: *Denne figur viser det færdige resultat af Z-pass algoritmen.*



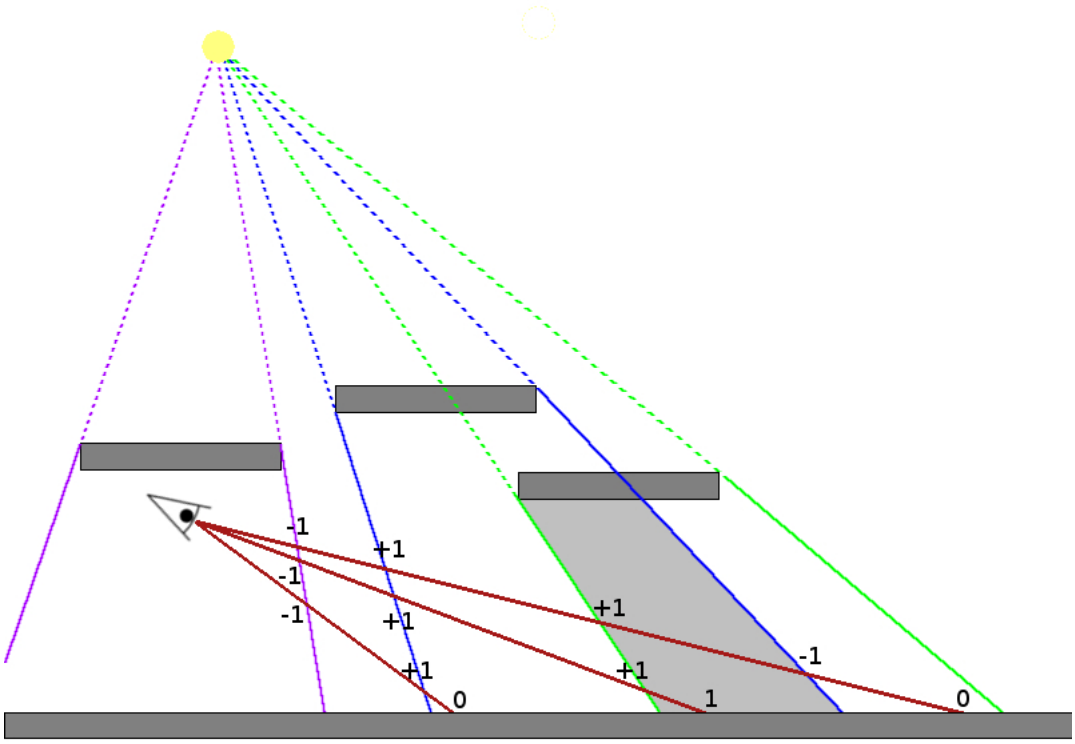
Figur A9: *Denne figur viser scene uden skygger.*



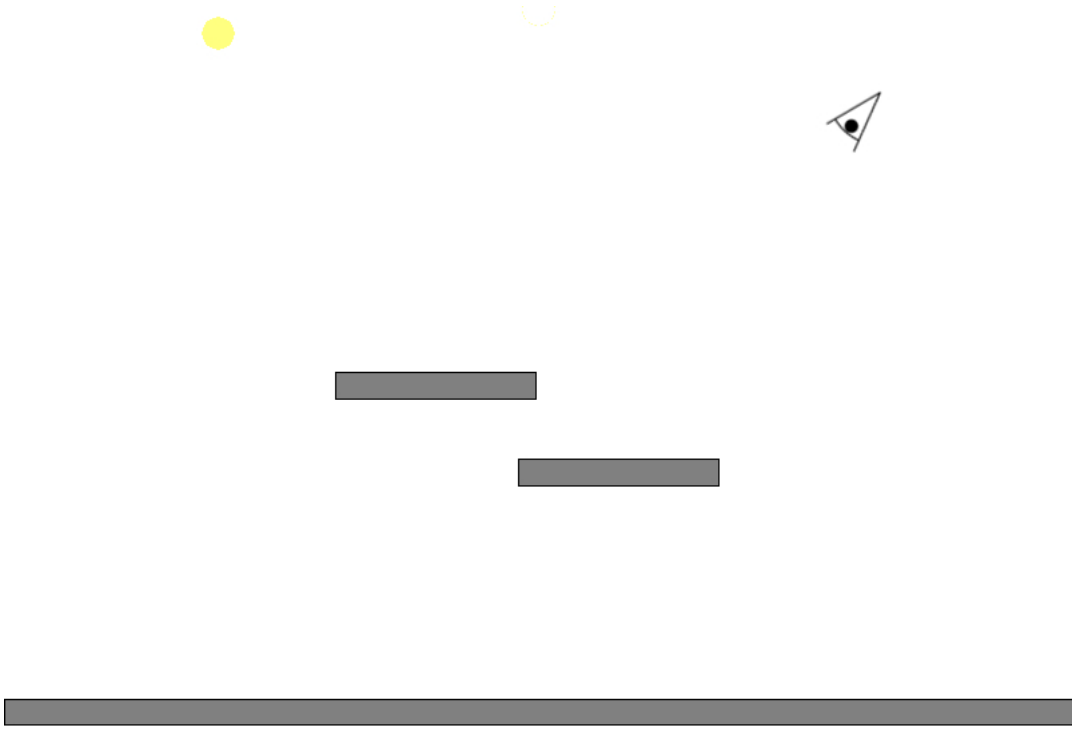
Figur A10: *Denne figur viser den første gennemkørsel af Z-pass algoritmen, hvor der bruges back-face culling. For hver forside af skyggevolumen linjen går ind i tælles værdien 1 op.*



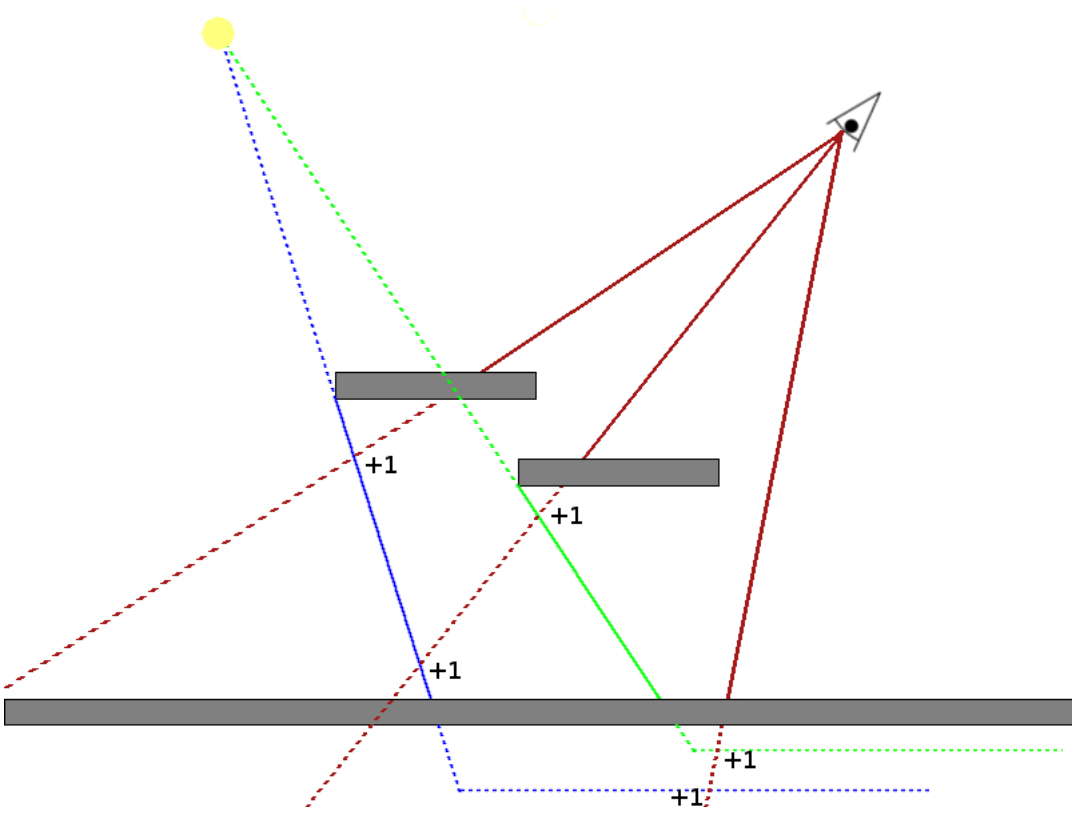
Figur A11: *Denne figur viser anden gennemkørsel af Z-pass algoritmen, hor der bruges front-face culling. Hver bagside af skyggevolumen linjen går igennem tæller værdien 1 ned.*



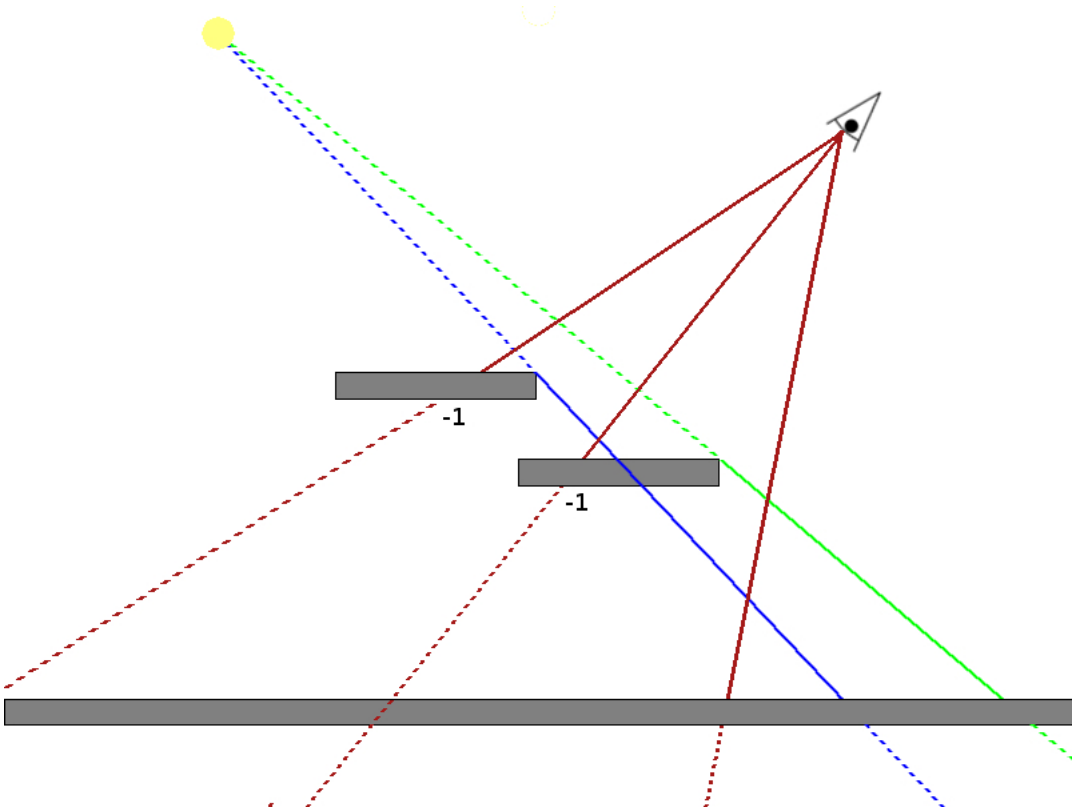
Figur A12: *Denne figur viser det færdige resultat af Z-pass algoritmen, hvor man tydeligt kan se at skyggerne ikke er korrekte.*



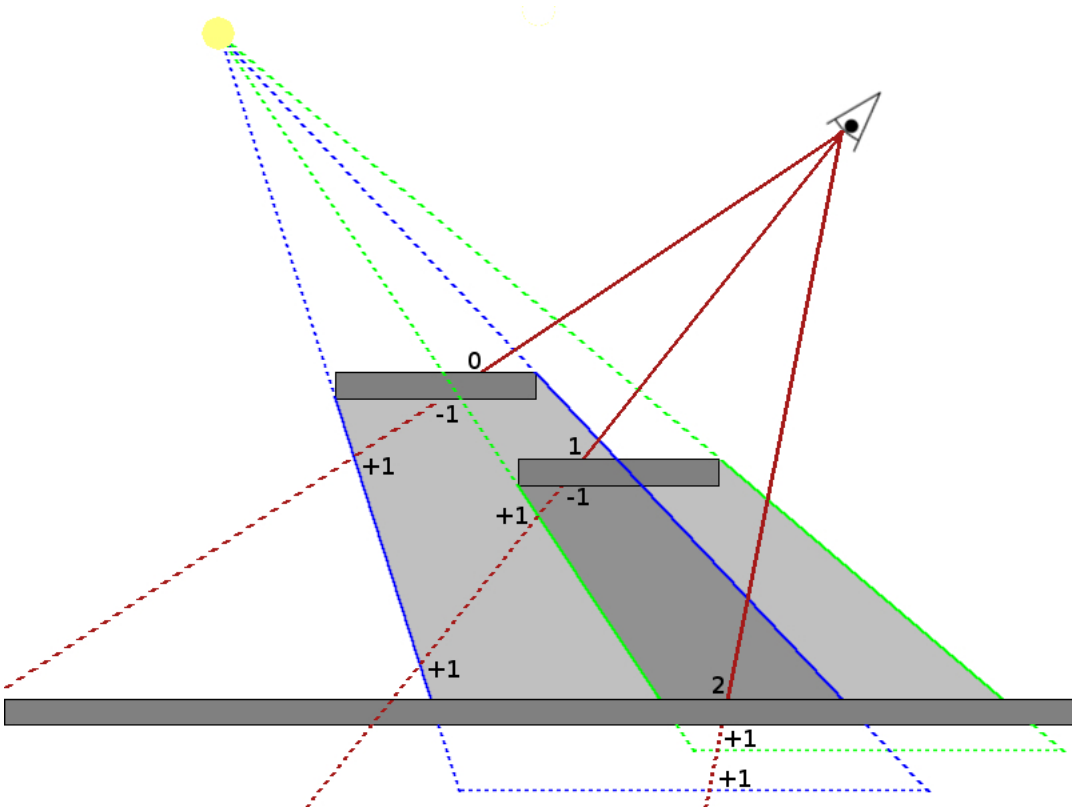
Figur A13: *Denne figur viser hvordan scenen ser ud uden skygger.*



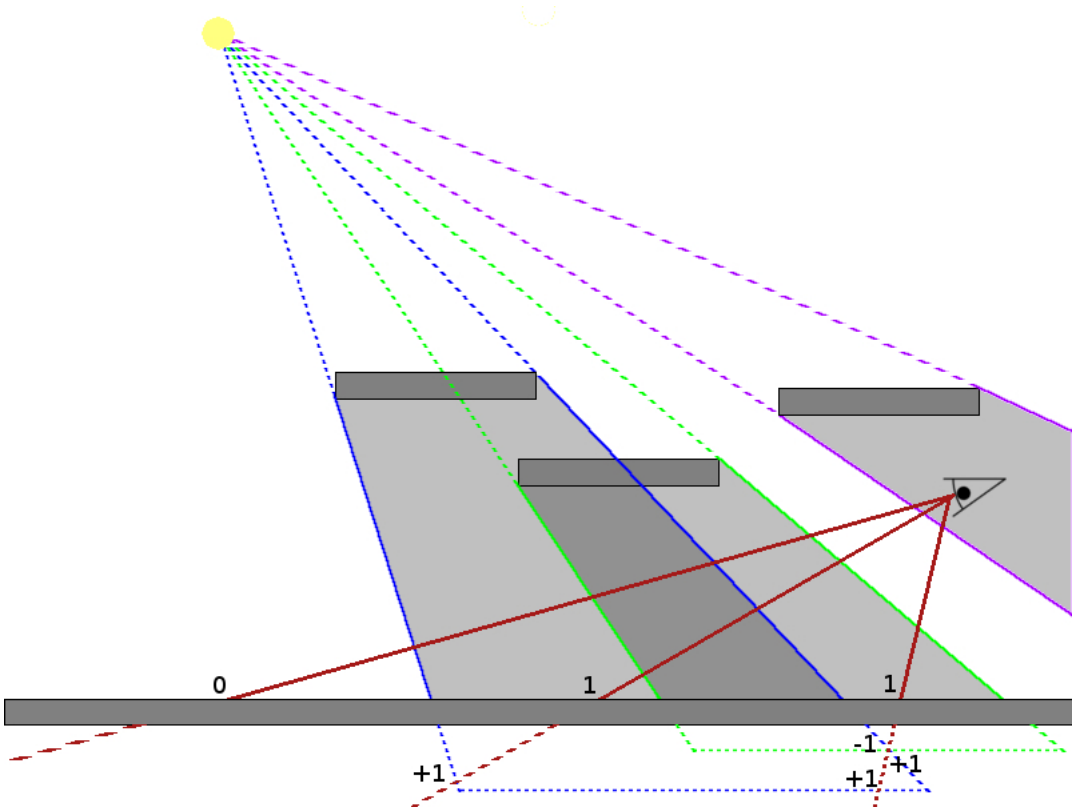
Figur A14: *Denne figur viser første gennemkørsel af Z-fail, hvor der bruges front-face culling. Bemærk at volumenerne skal være aflukket, så derfor vil der være 2 gange +1 fra linjen helt ude til højre. 1 gang for den grønne volumen og 1 gang for bunden af den blå volumen (som linjen vil skærer uden for billedet).*



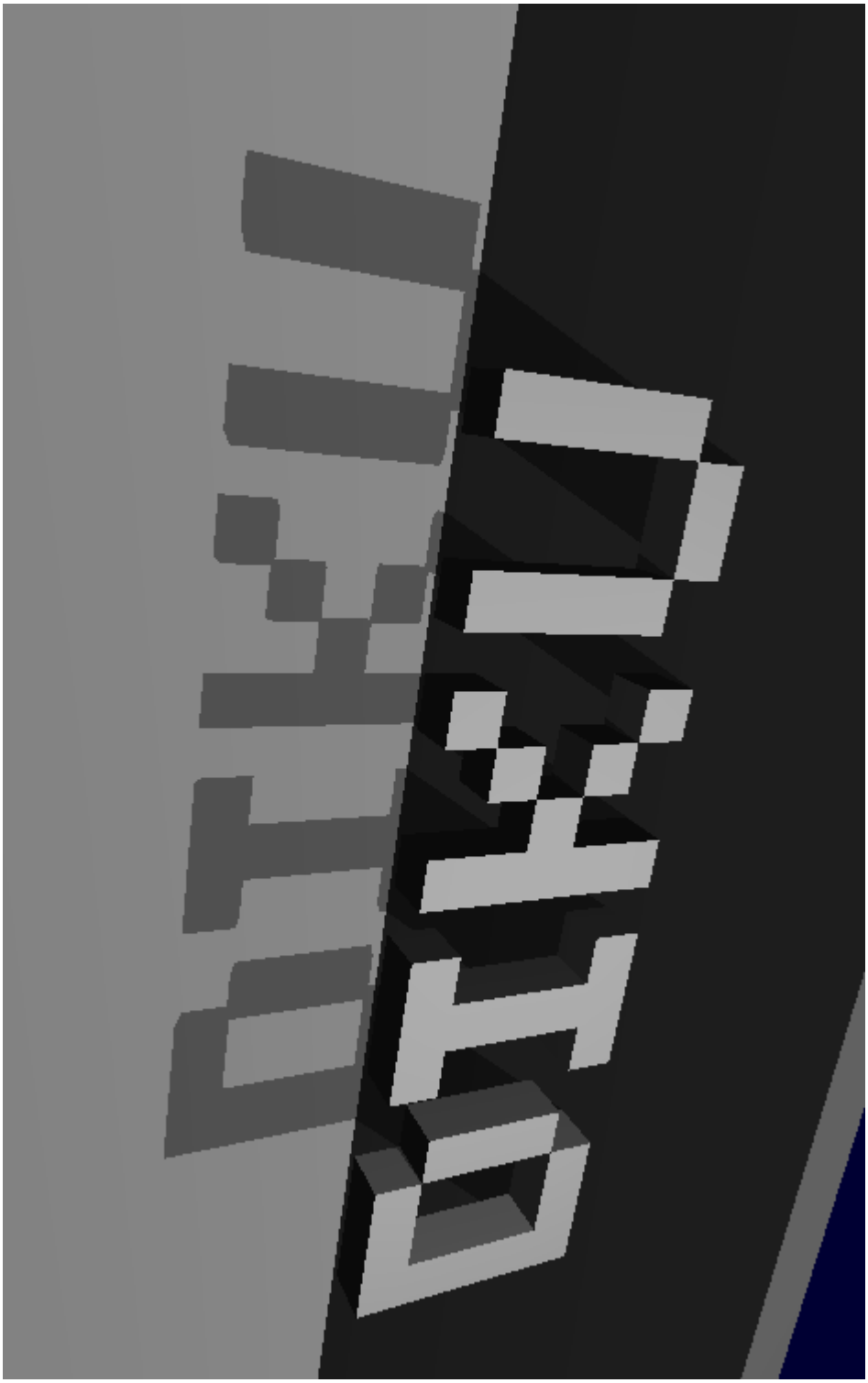
Figur A15: *Denne figur viser anden gennemkørsel af Z-fail, hvor der bruges back-face culling. Bemærk igen at volumenerne skal være afluk-ket, så toppen af volumenerne vil dekrementere værdien.*



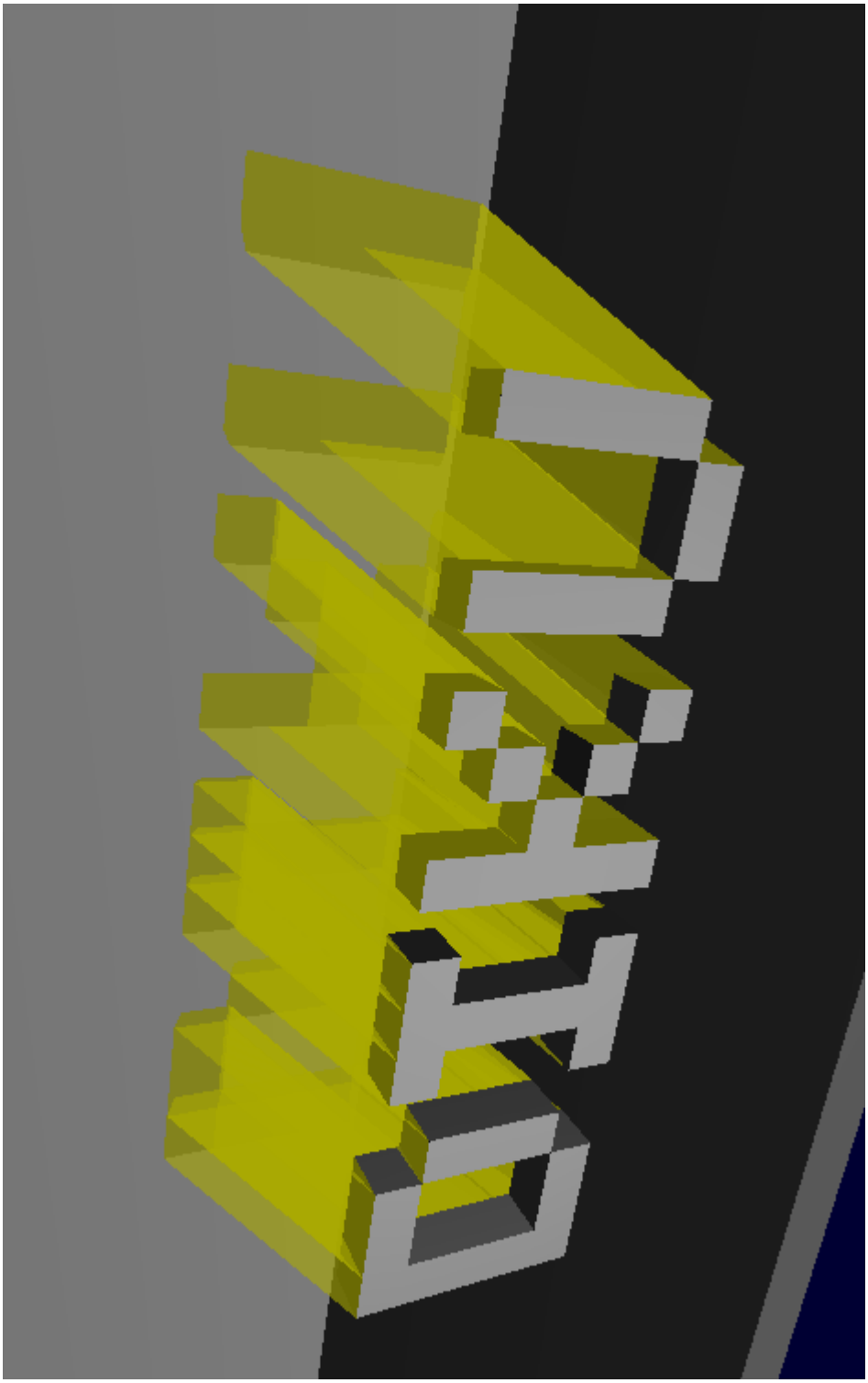
Figur A16: *Denne figur viser det færdige resultat af Z-fail algoritmen.*



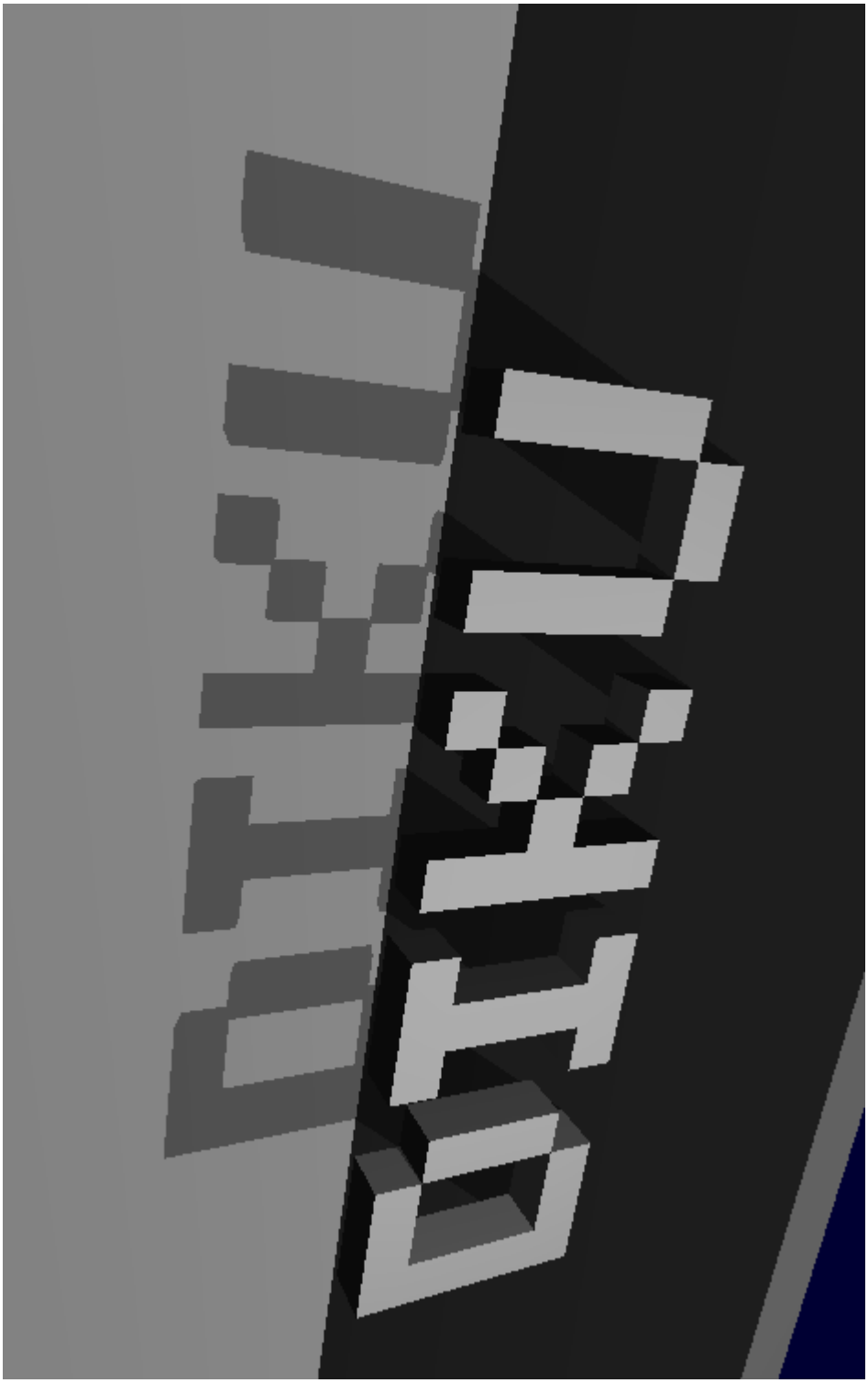
Figur A17: *Her kan man se hvordan Z-fail, helt automatisk, klarer situationen hvor kameraet er indeni et skyggevolumen.*



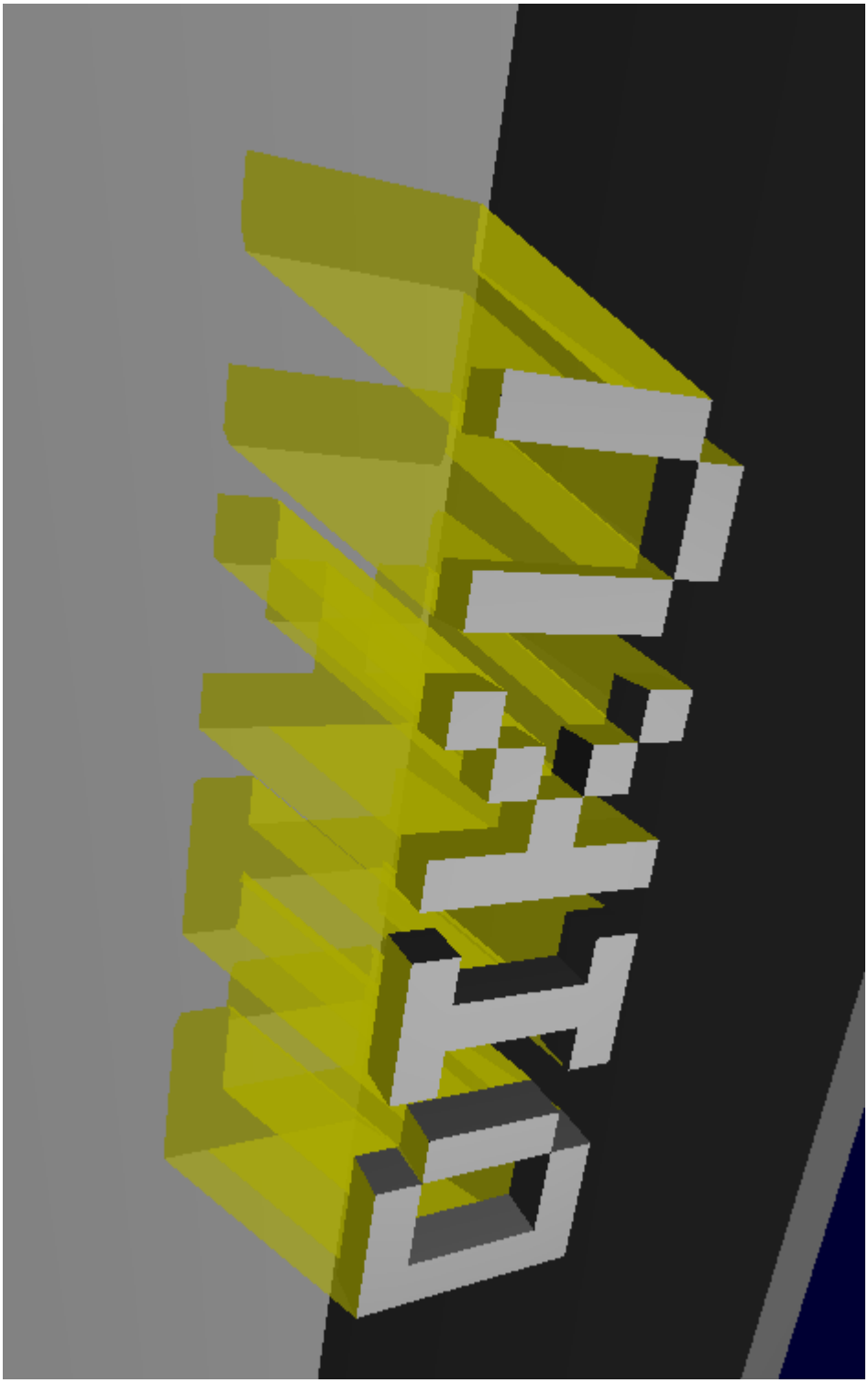
Figur A18: *I denne scene bliver der ikke brugt silhuetter til at tegne skyggerne. Bemærk det store antal af volumener der bliver brugt for at lave skyggerne.*



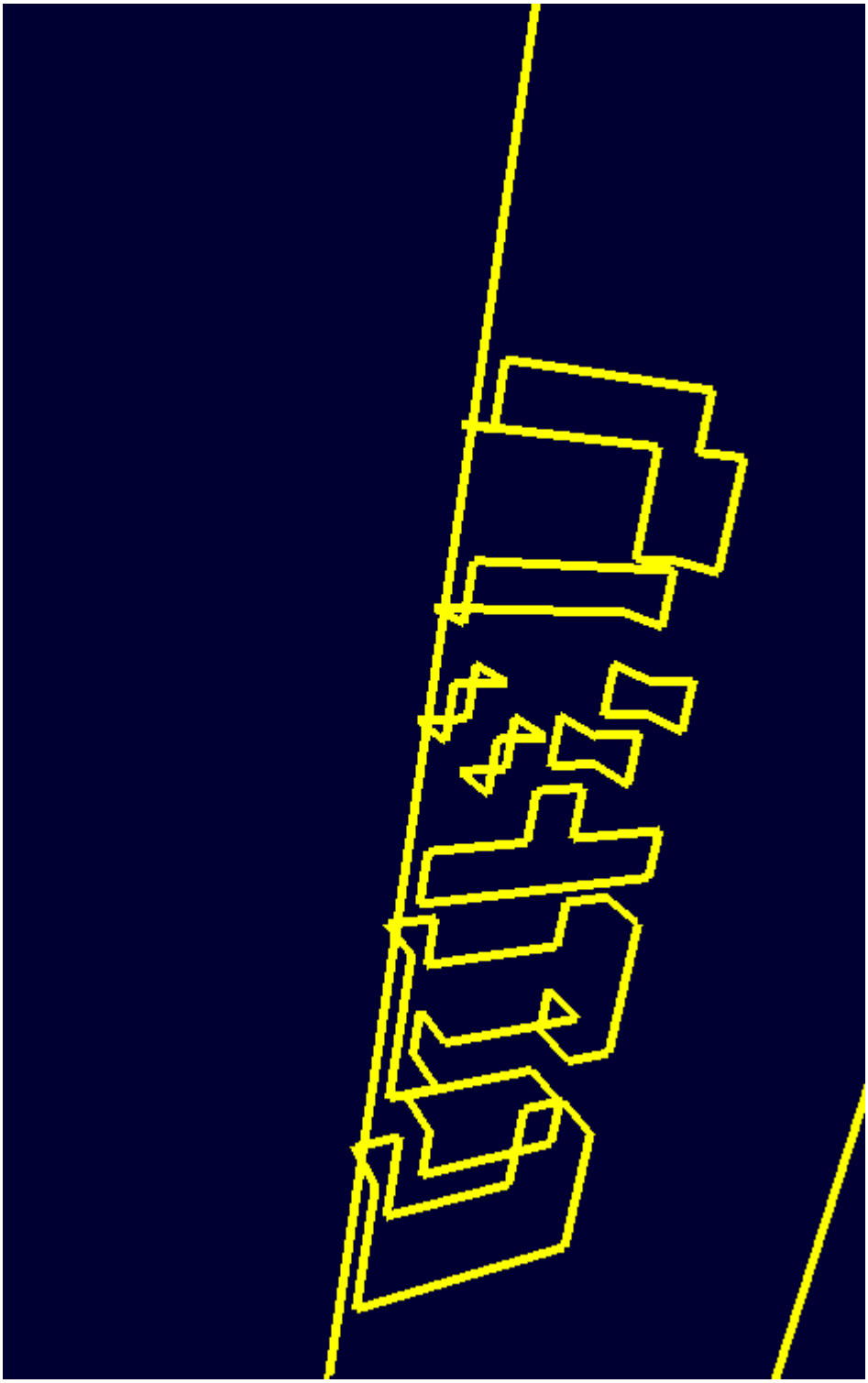
Figur A19: *I denne scene bliver der ikke brugt silhuetter til at tegne skyggerne. Bemærk det store antal af volumener der bliver brugt for at lave skyggerne.*



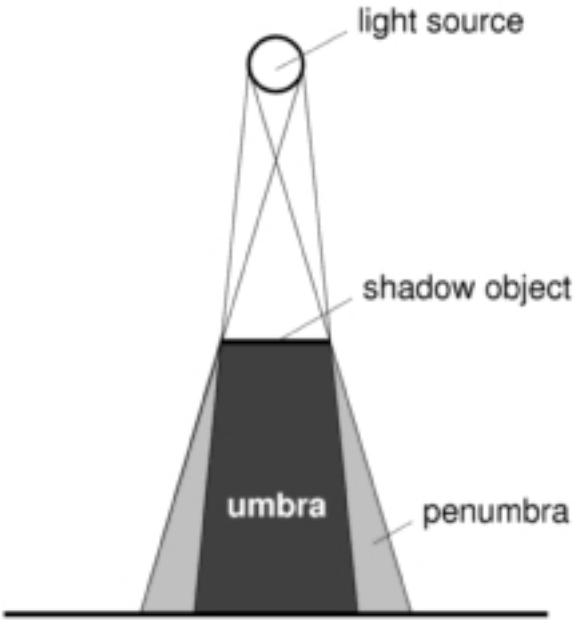
Figur A20: *Dette er samme scene som før, men her bliver der brugt silhuetter. Som man kan se, er selve skyggerne stort set ens. Ser man derimod på antallet af volumen der bliver brugt, kan man tydelig se en ændring. I det først billede bliver der i alt tegnet 24 volumener, i mens der kun bliver tegnet 11 på andet billede. Bemærk især denne forskel ved 'D' og 'I'. Det sidste billede viser de silhuetter der bliver genereret ud fra scenen.*



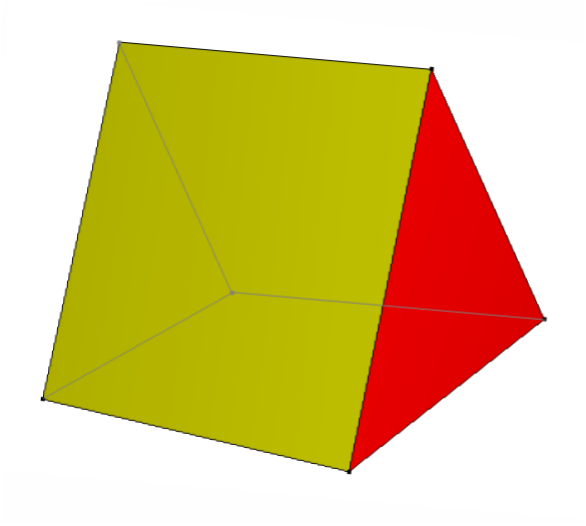
Figur A21: *Dette er samme scene som før, men her bliver der brugt silhuetter. Som man kan se, er selve skyggerne stort set ens. Ser man derimod på antallet af volumen der bliver brugt, kan man tydelig se en ændring. I det først billede bliver der i alt tegnet 24 volumener, i mens der kun bliver tegnet 11 på andet billede. Bemærk især denne forskel ved 'D' og 'I'. Det sidste billede viser de silhuetter der bliver genereret ud fra scenen.*



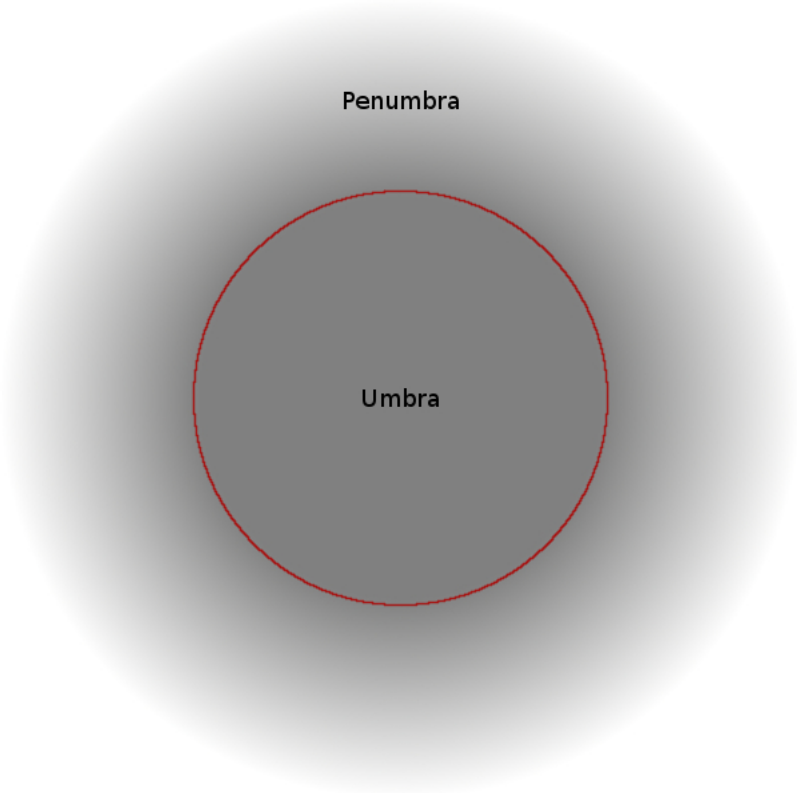
Figur A22: *Dette er samme scene som før, men her bliver der brugt silhuetter. Som man kan se, er selve skyggerne stort set ens. Ser man derimod på antallet af volumen der bliver brugt, kan man tydelig se en ændring. I det først billede bliver der i alt tegnet 24 volumener, i mens der kun bliver tegnet 11 på andet billede. Bemærk især denne forskel ved 'D' og 'I'. Det sidste billede viser de silhuetter der bliver genereret ud fra scenen.*



Figur A23: *Denne figur viser forskellen mellem umbra og penumbra. Umbra er den del der er helt i skygge, mens penumbra er den del der kun er delvis i skygge.*

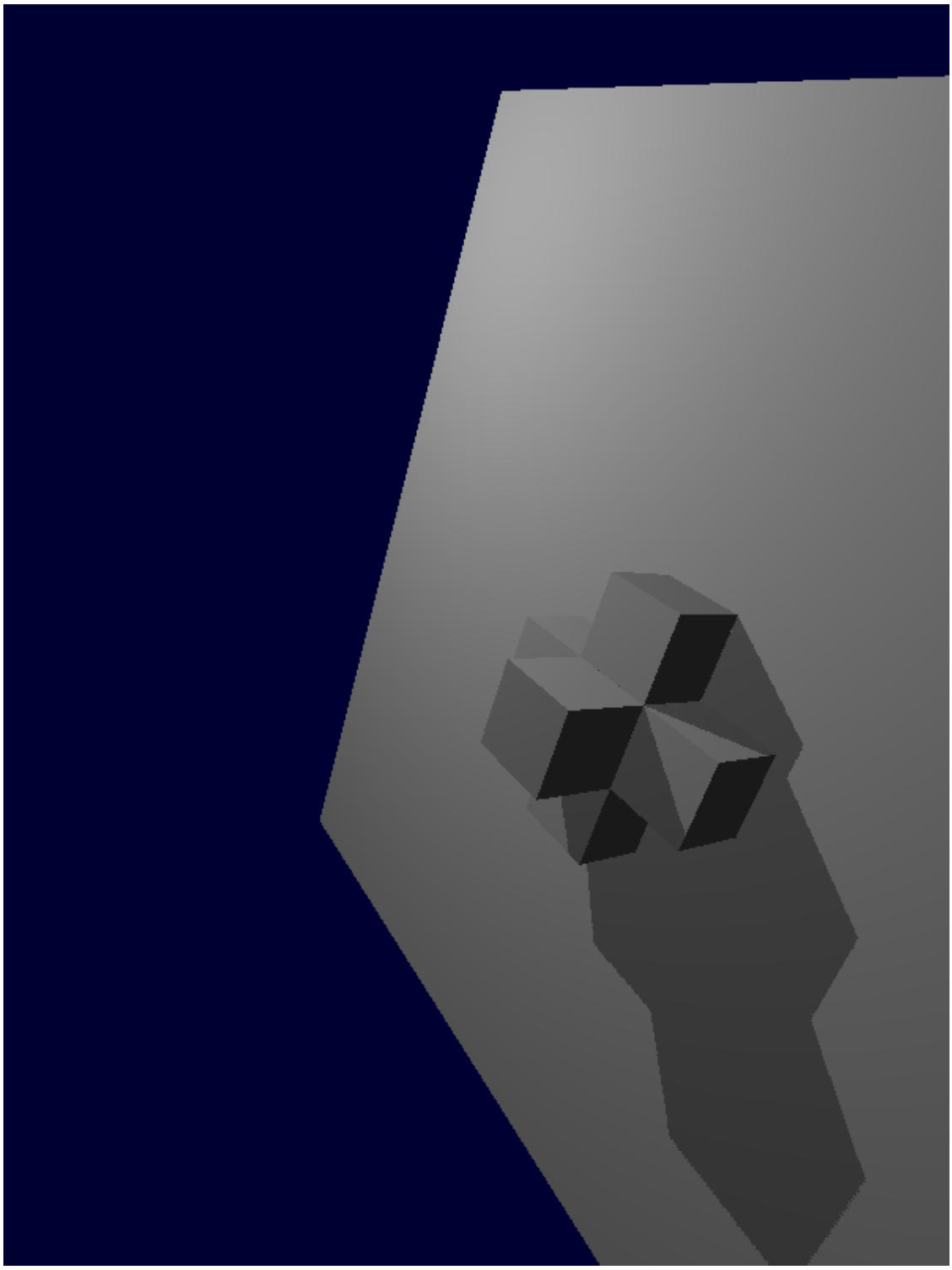


Figur A24: *Dette viser hvad en kile er. Den består af 2 trekanter samt 3 trapezer.*

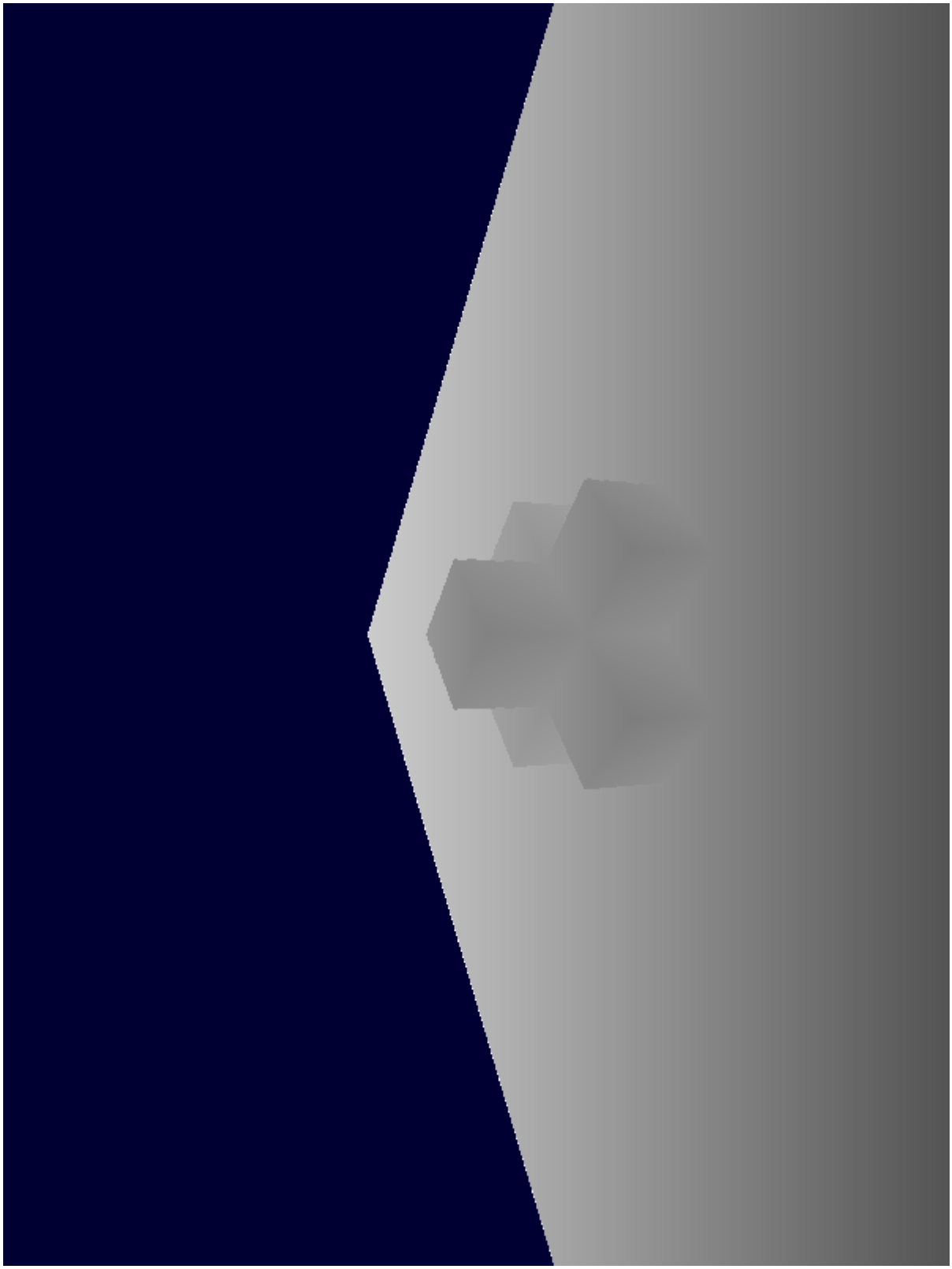


Figur A25: *Denne figur viser et simpelt eksempel på hvordan penumbra wedges fungere. Her kan man se at Umbraen ligger helt i skygge. Penumbraen falder i intensitet jo længere væk den kommer fra selve skygge, hvilket giver den flyende overgang.*

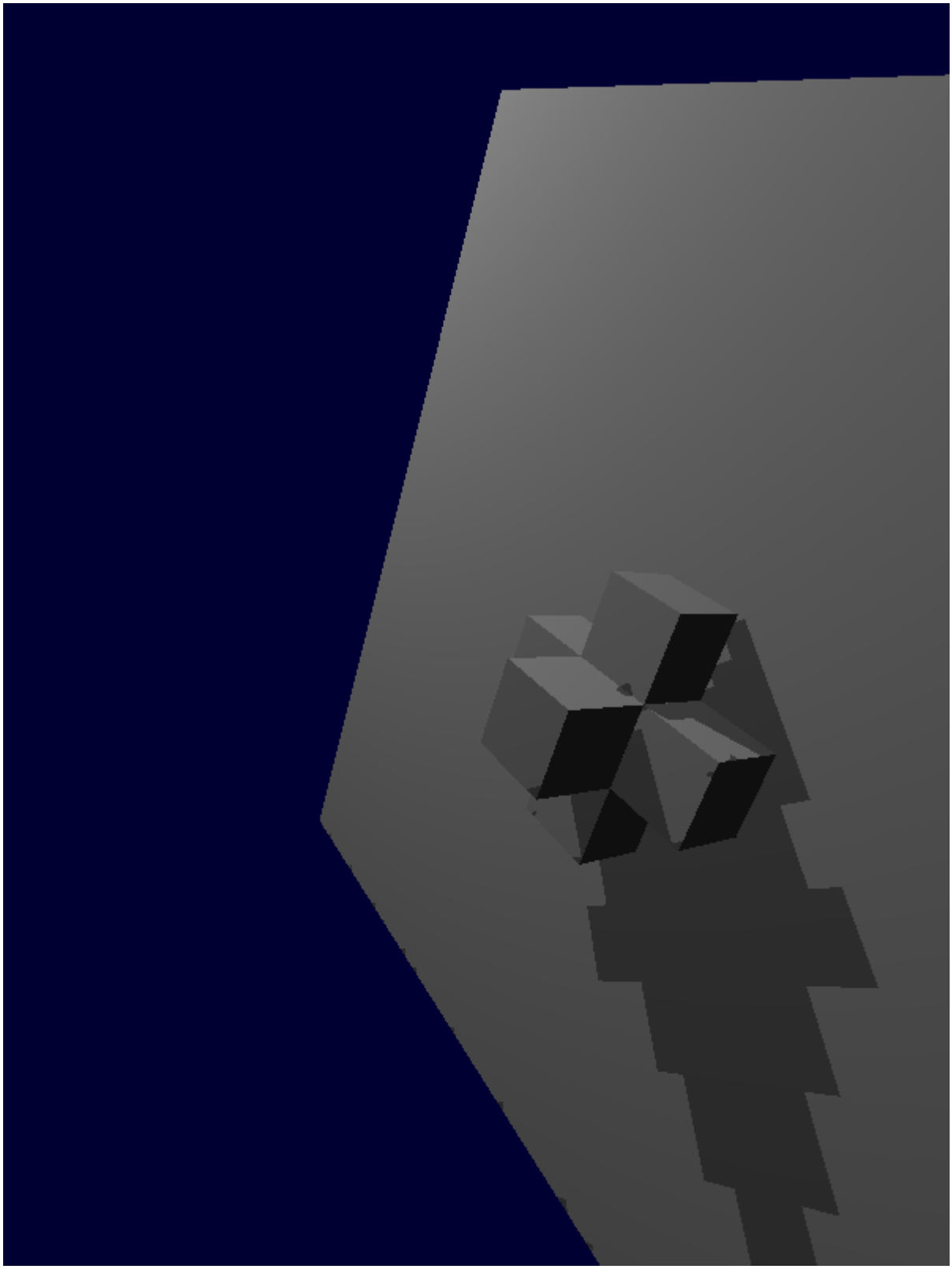
A.2 Billeder til kapitel 3



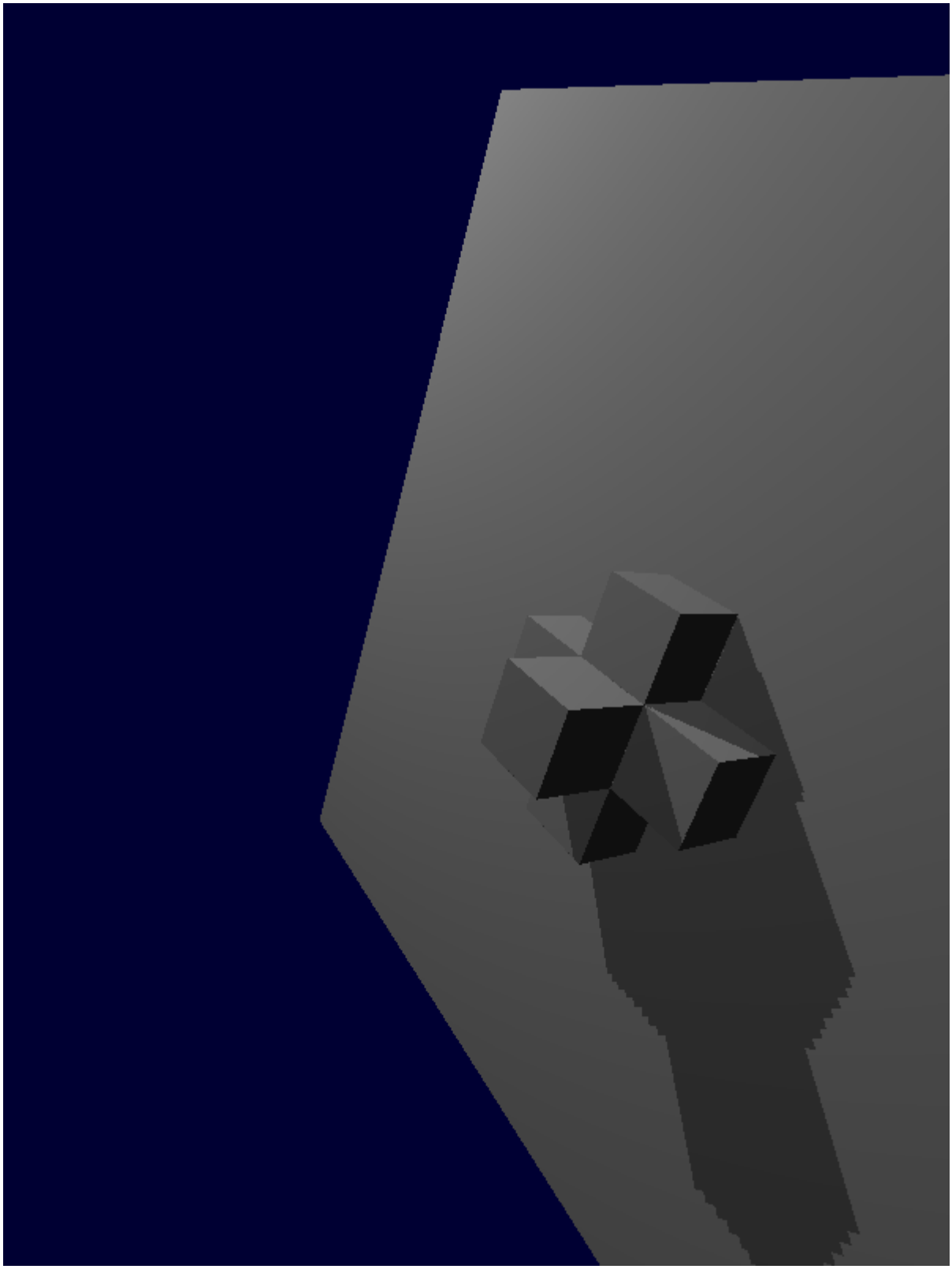
Figur A26: *Denne figur viser den reelle scene med skygger.*



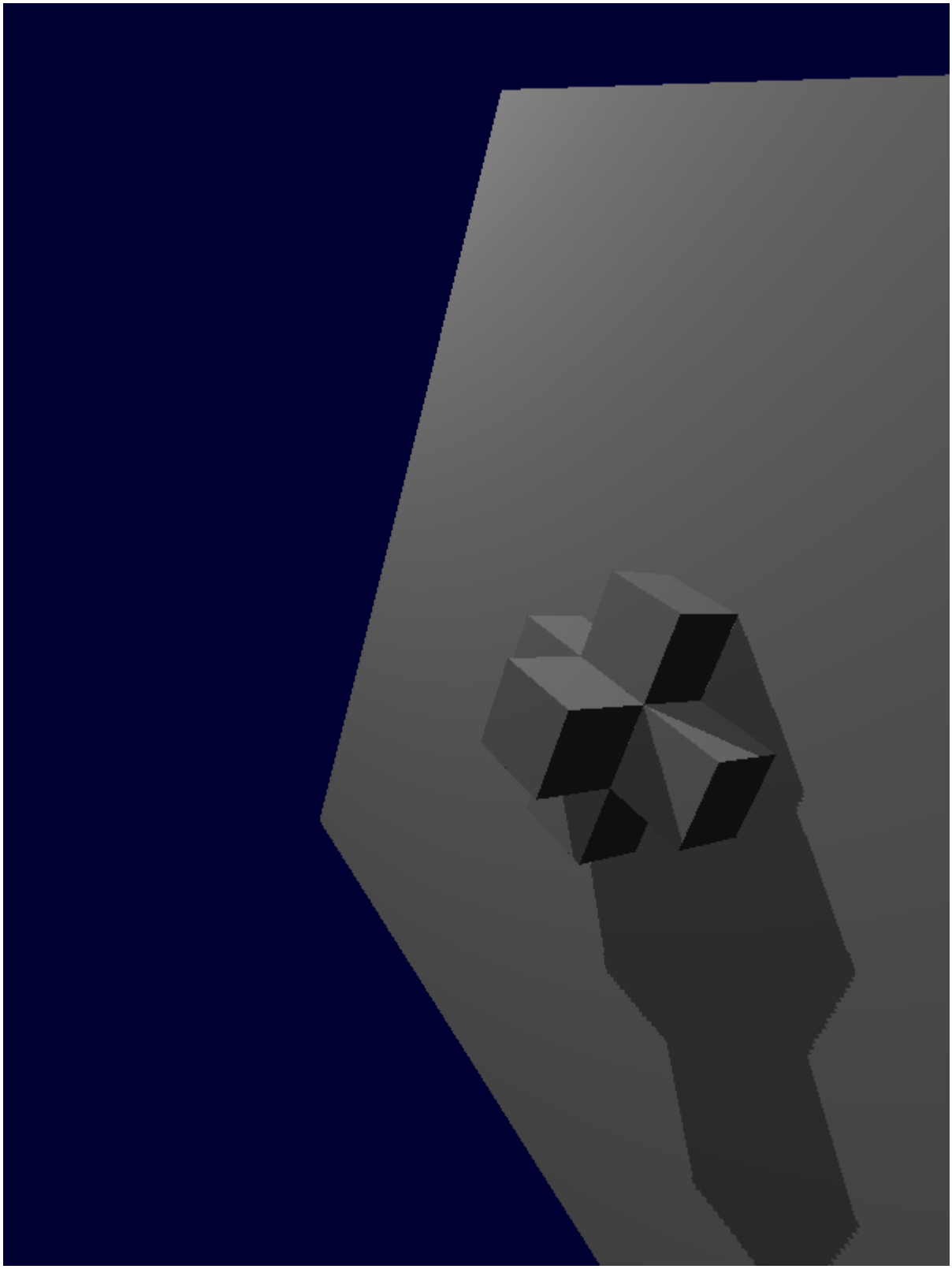
Figur A27: *Denne figur viser dybde mappet, udregnet fra lyset, til forrige figur.*



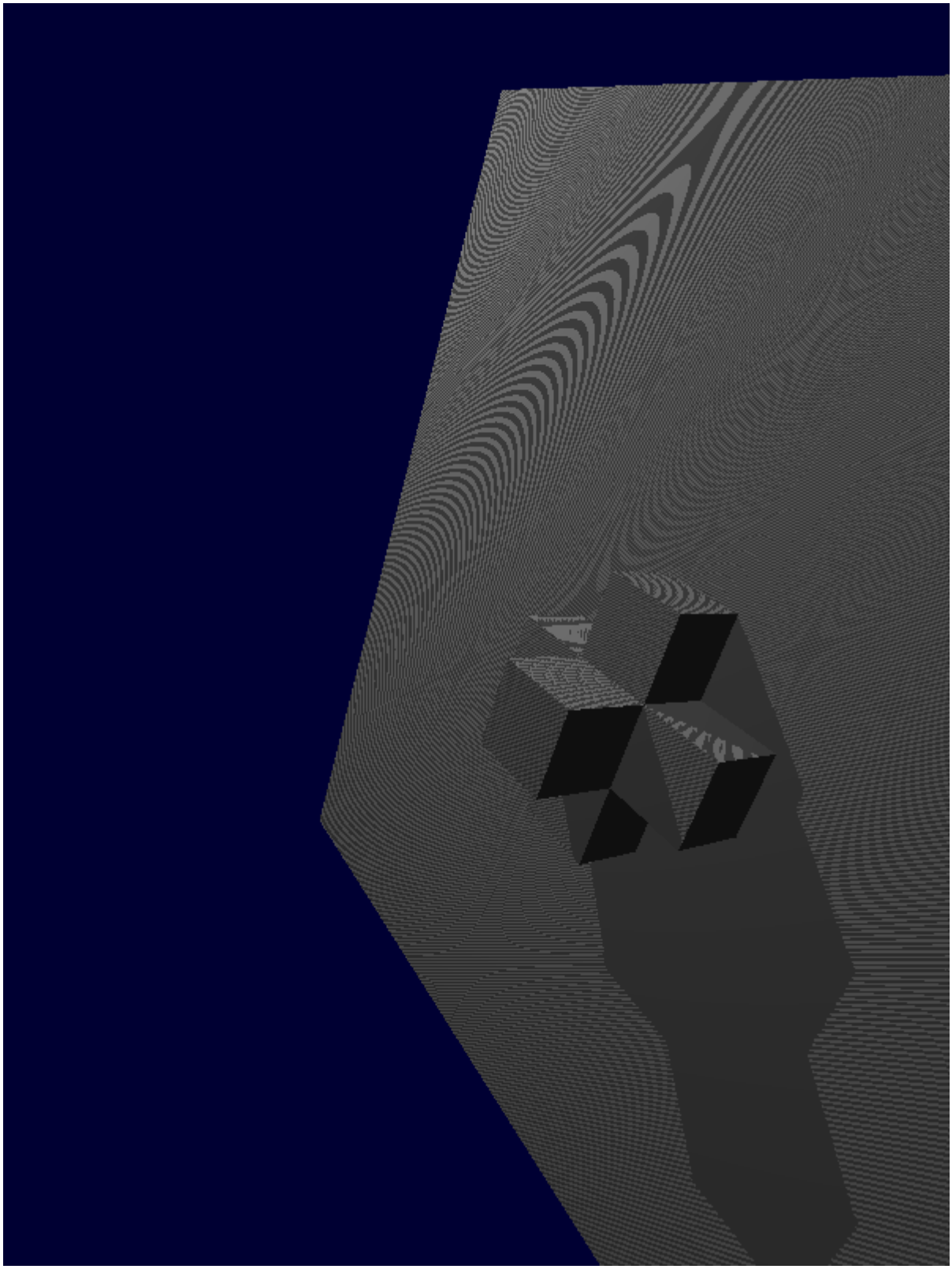
Figur A28: *Shadowmap med en opløsningen* 80×60



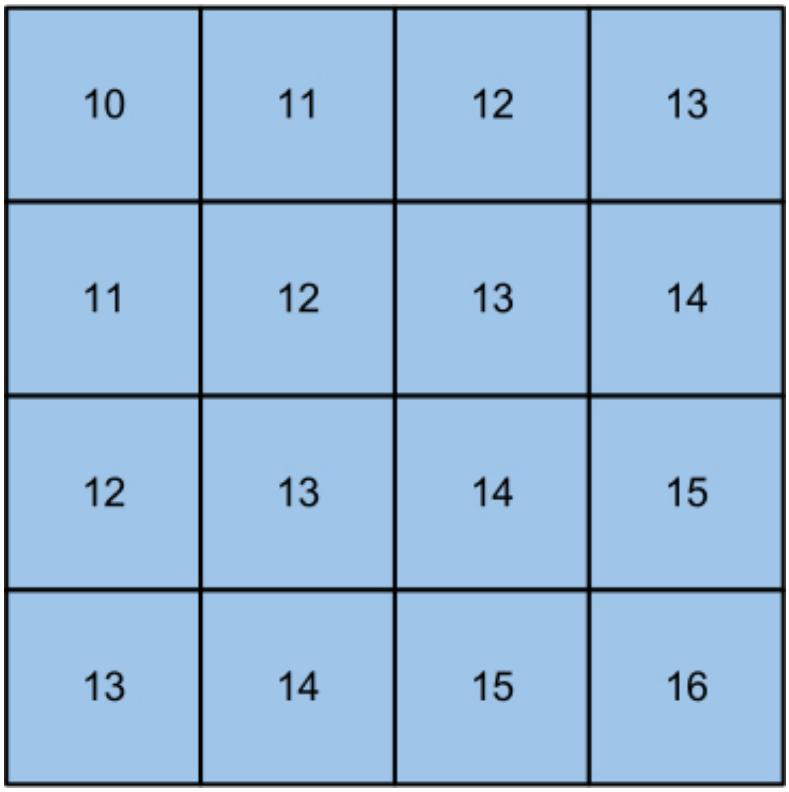
Figur A29: *Shadowmap med en opløsningen* 800×600



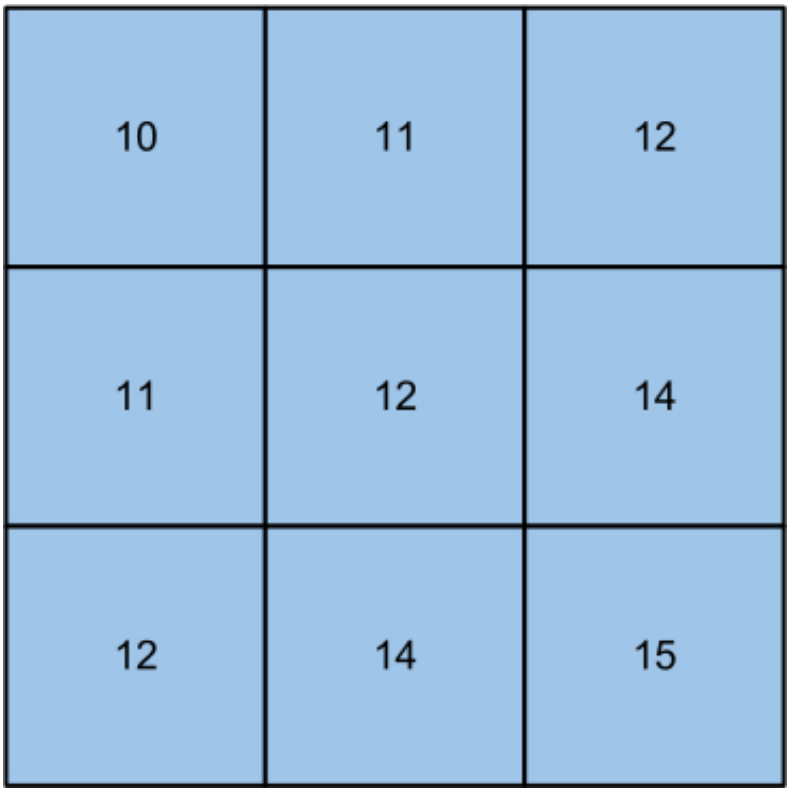
Figur A30: *Shadowmap med en opløsningen* 1600×1200



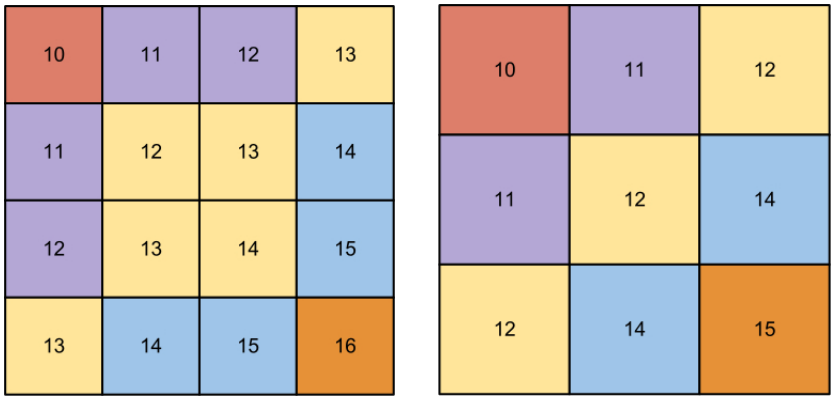
Figur A31: *Denne figur viser et eksempel på Moirés pattern fra vores program.*



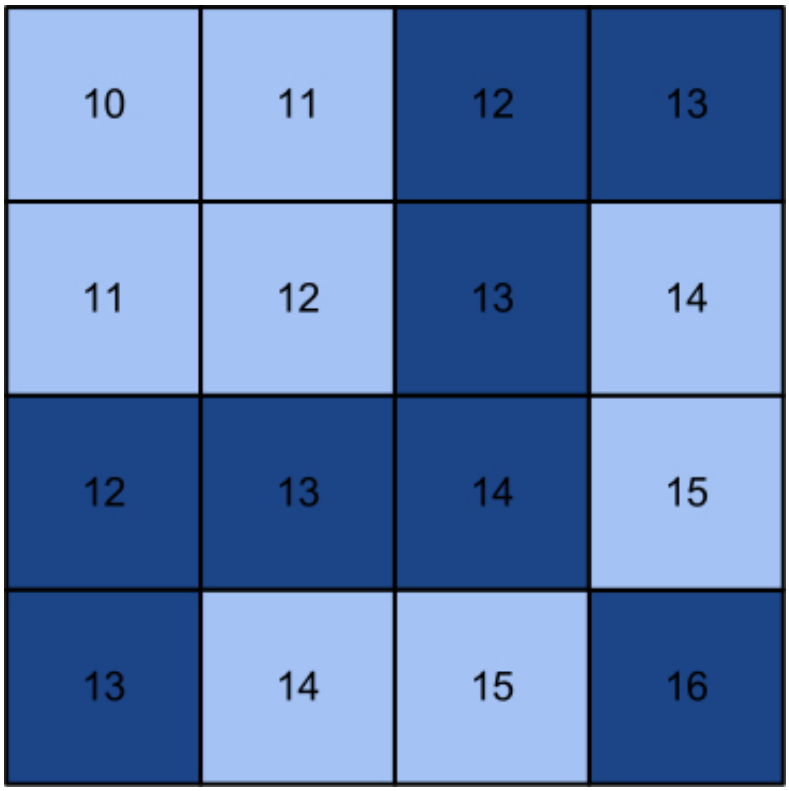
Figur A32: *Dette er dybdeværdier (i forhold til lyset) af nogle punkter set fra kameraet, bemærk at ingen af punkterne ligger i skygge.*



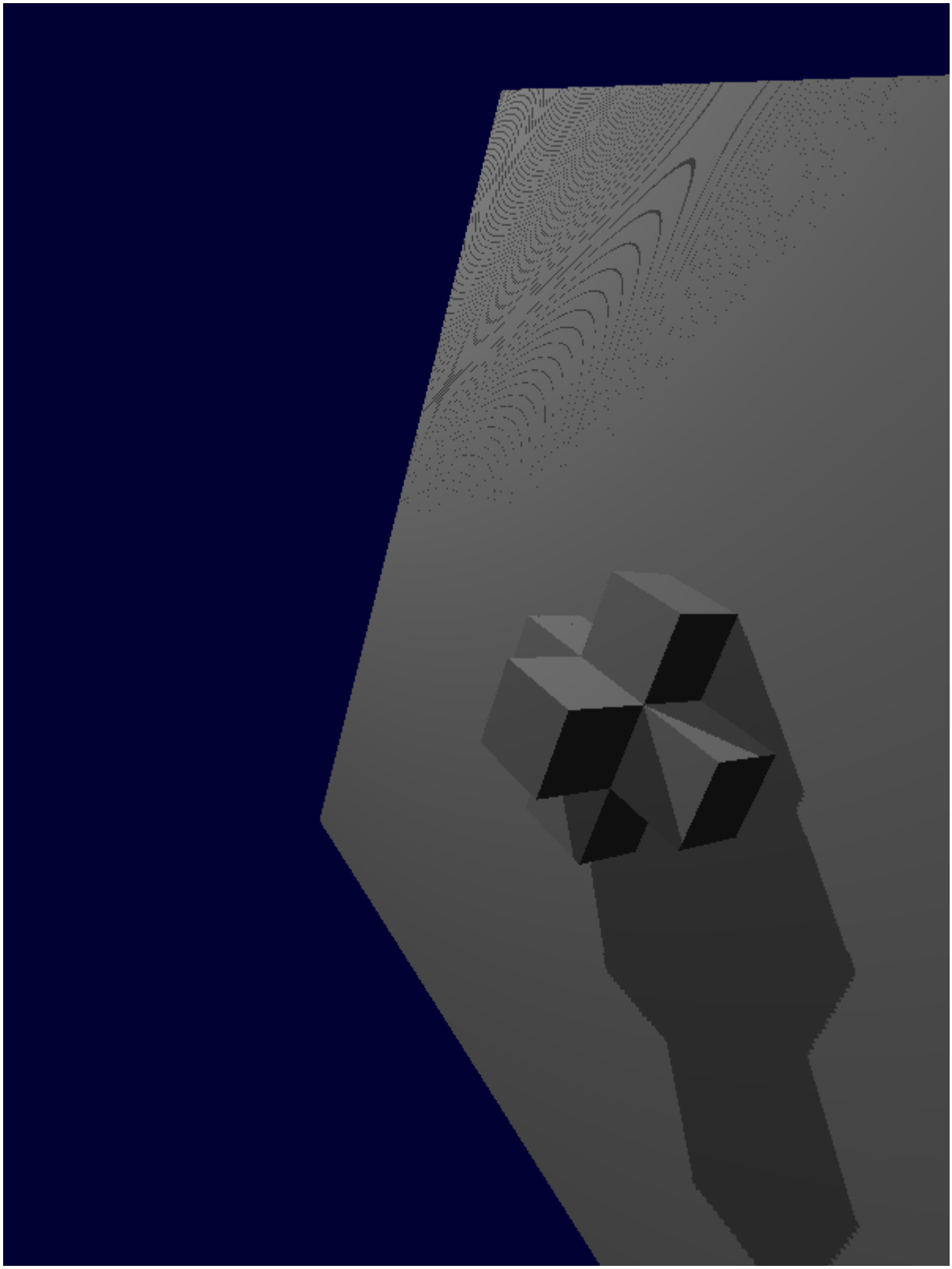
Figur A33: *Dette er dybdeværdier af nogle punkter set fra lyset, det vil sige vores shadowmap.*



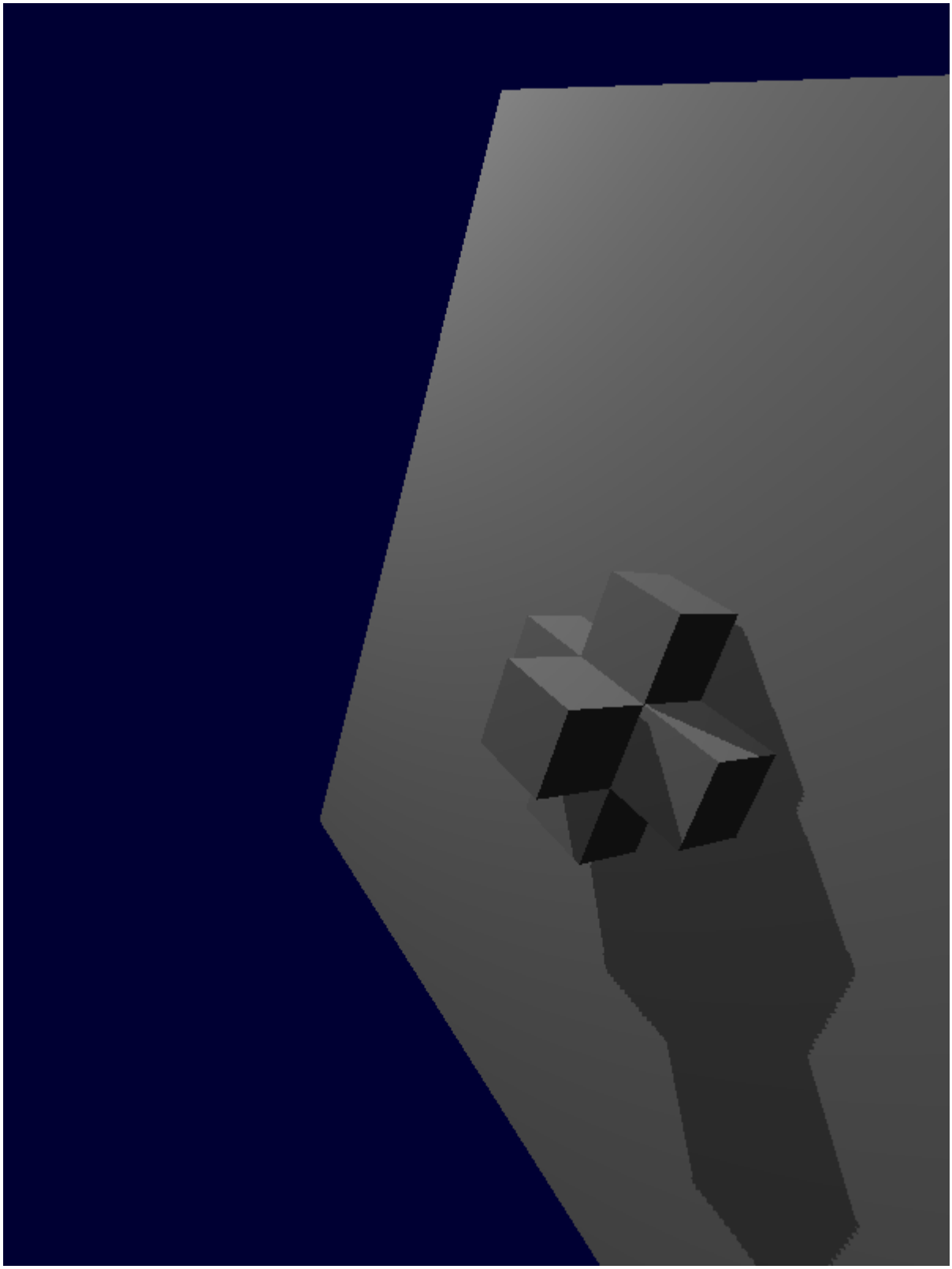
Figur A34: *Her kan man se hvordan diverse punkter fra kameraet, mapper til værdierne i shadowmappet. De røde pixels på venstre figur mapper til de røde figure på højre osv. Hvis værdien på en given pixel fra kameraet (venstre figur) er højere end værdien i shadowmappet (højre figur, så ligger punktet i skygge.)*



Figur A35: *Her kan man så se nogle af pixelsne ligger i skygge selv om de ikke burde (Husk at ingen af punkter lå i skygge i første figur). Denne figur er et eksempel på moire's pattern når man er helt tæt på.*

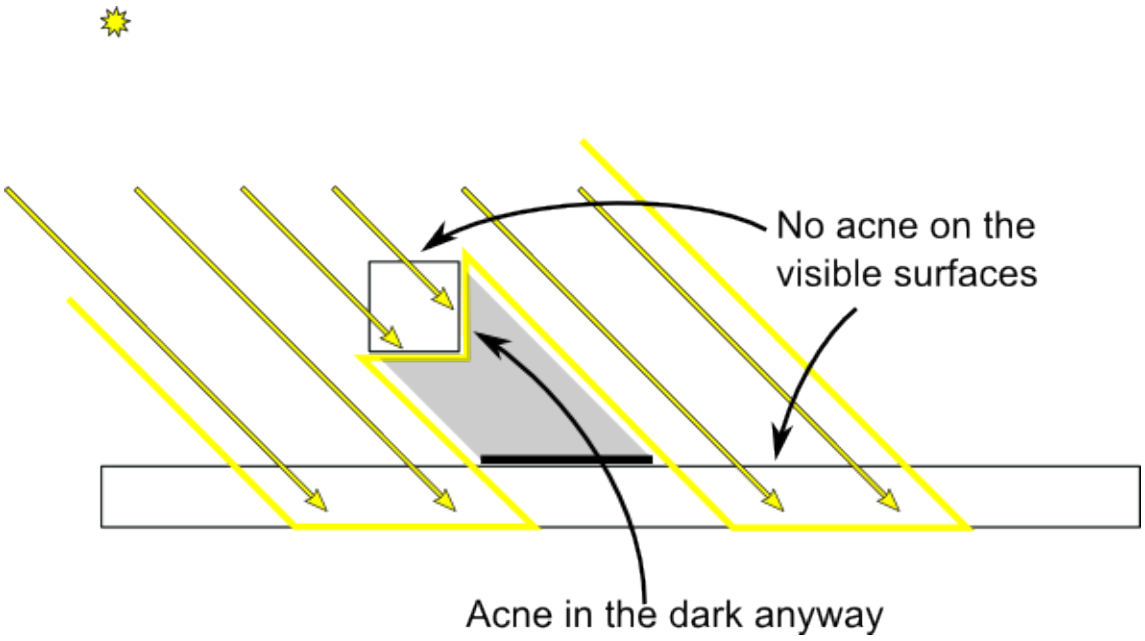


Figur A36: *I dette billed bliver der brugt en bias konstant på 0.00001. Som man kan se giver det et væsentligt bedre billed, men det er stadig ikke nok til at fjerne problemet helt.*

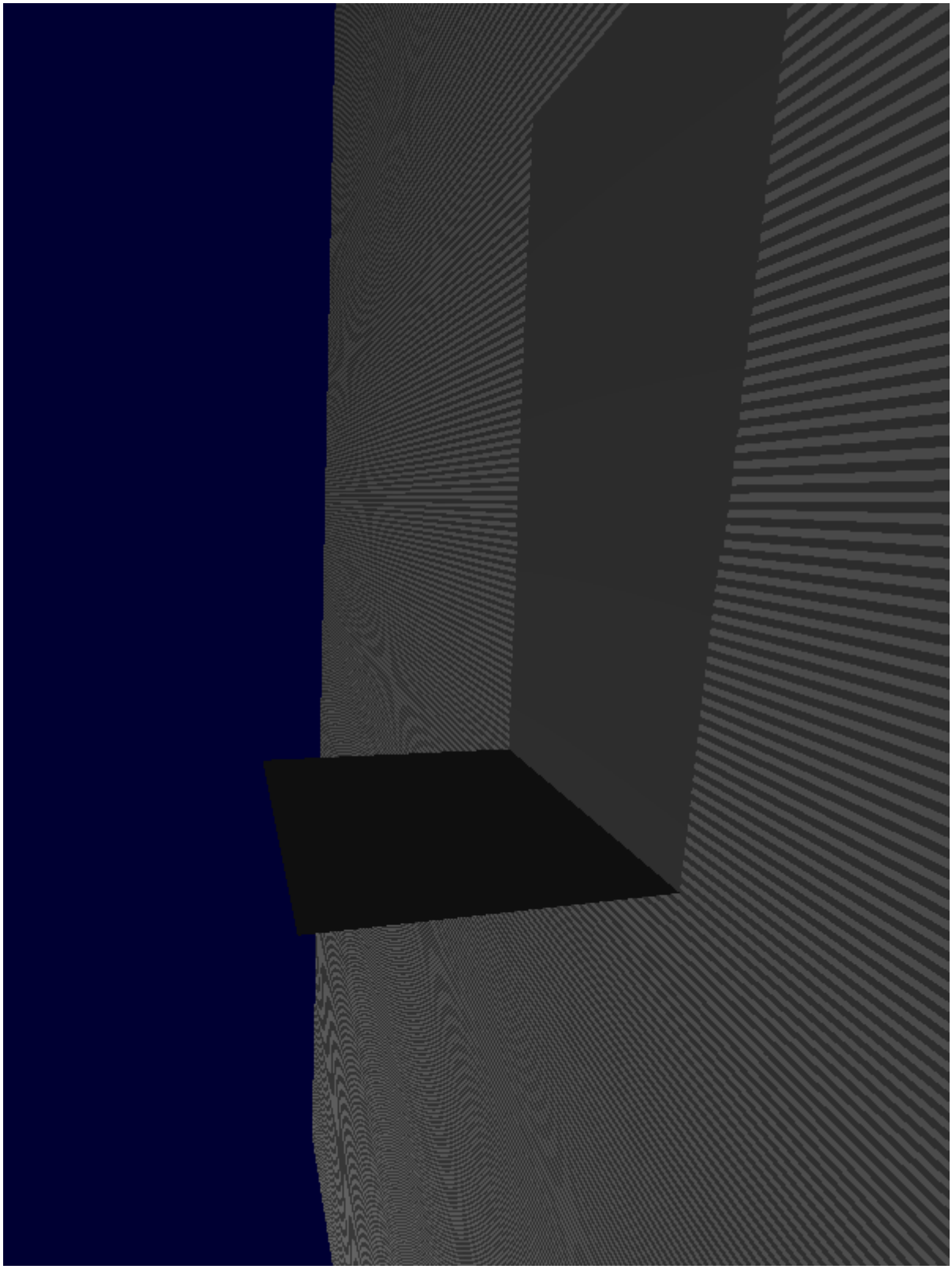


Figur A37: *Her bliver der brugt en bias konstant på 0.0001. Bemærk at alle de uønskede mønstre nu er væk.*

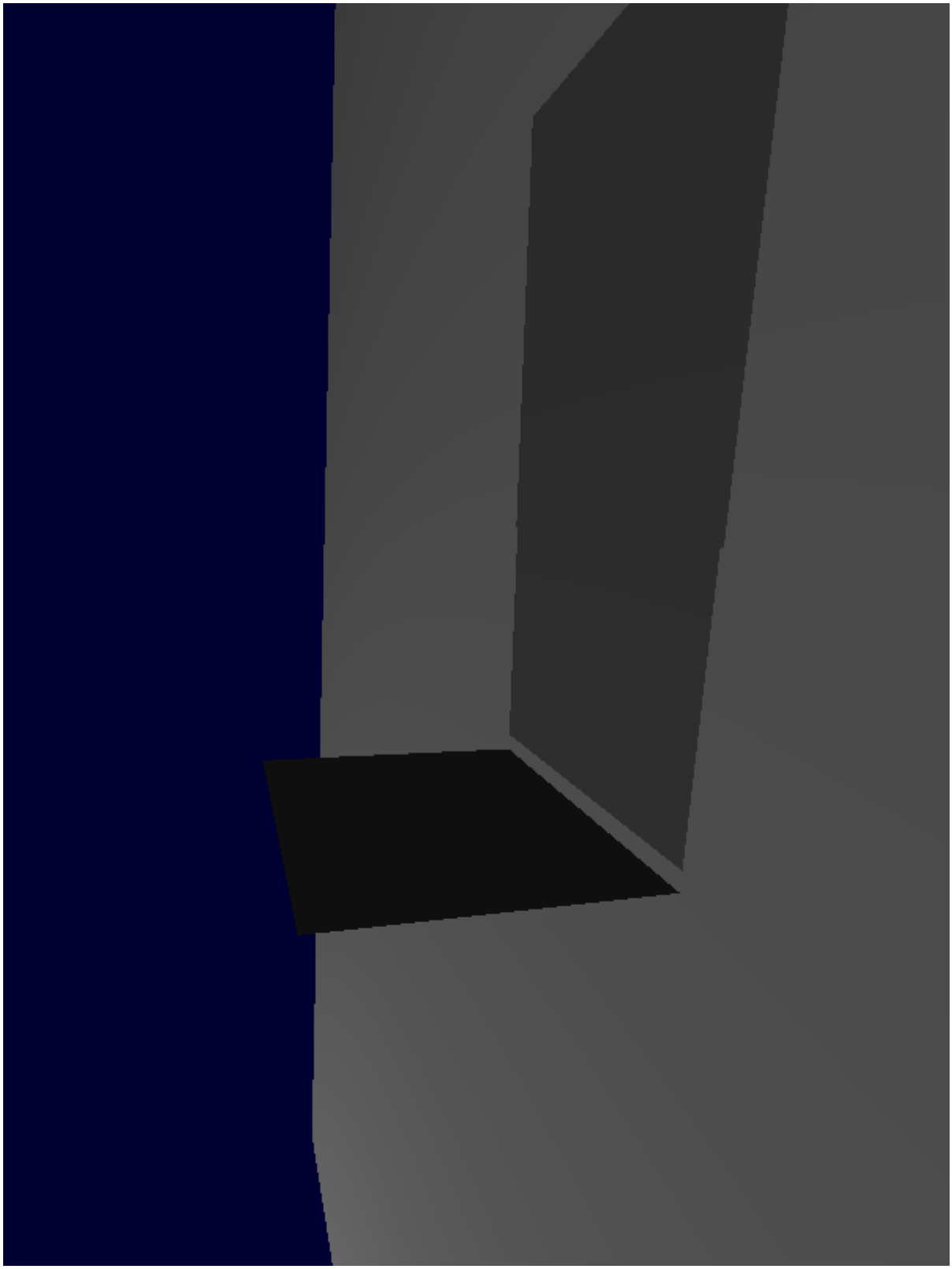
A.3 Billeder til kapitel 6



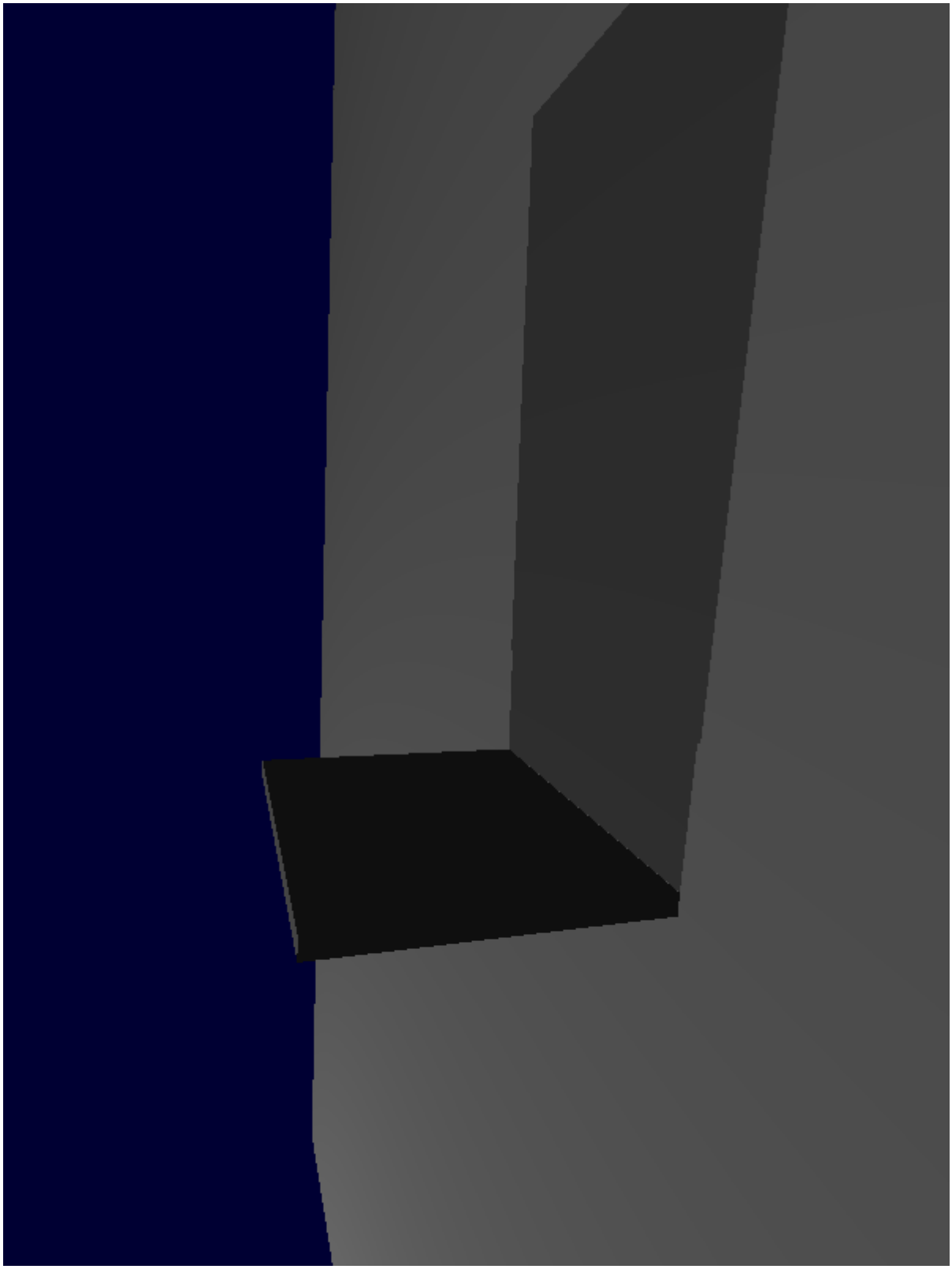
Figur A38: *Billed er taget fra OpenGL-tutorial [OT13].*
Her kan man se en demonstration af shadowmapping med culling-metoden, som man kan se flyttes moire's pattern nu over på bagsiden er af figuren. det vil sige den del hvor figuren skygger for sig selv.



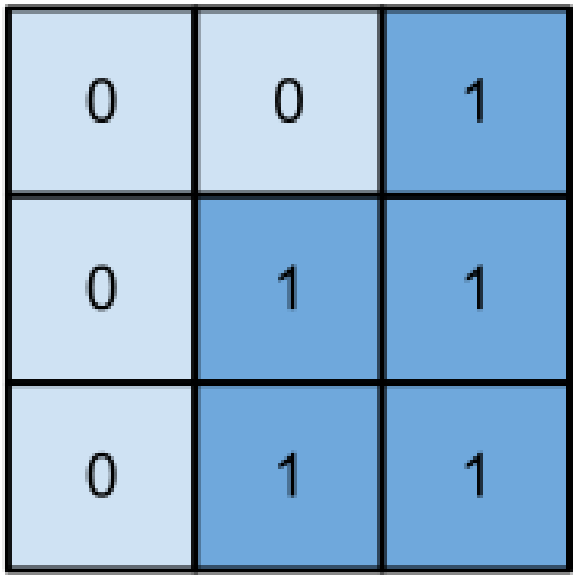
Figur A39: *En test scene uden bias. Her er der ingen peter panning og et meget tydeligt moire's pattern.*



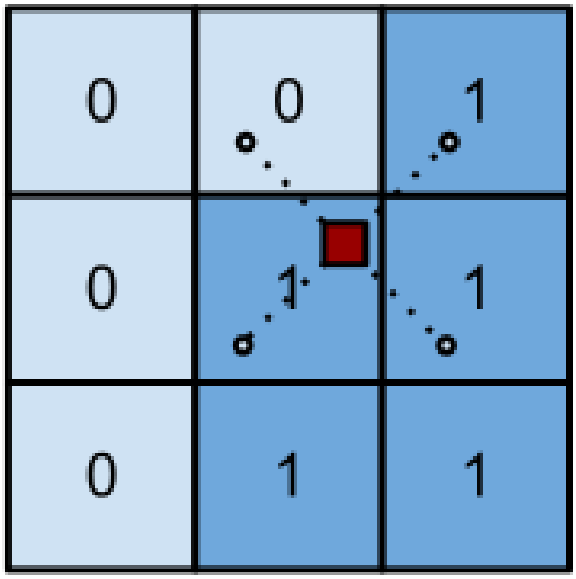
Figur A40: *Sammen Scene som før, men nu uden moire's pattern. Her er det tydeligt at se hvilken effekt tilføjelsen af en bias har. Fænomenet hvor skyggen først starter et stykke efter figuren kaldes som sagt peter panning.*



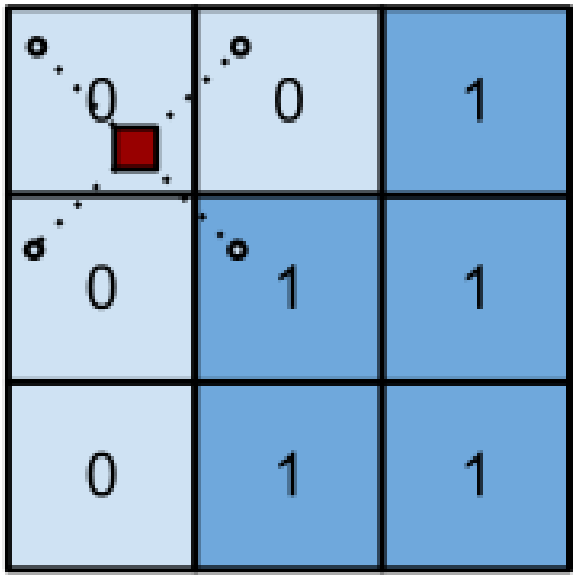
Figur A41: *Her kan man se at en ikke-flad version af geometri fra før har løst problemet.*



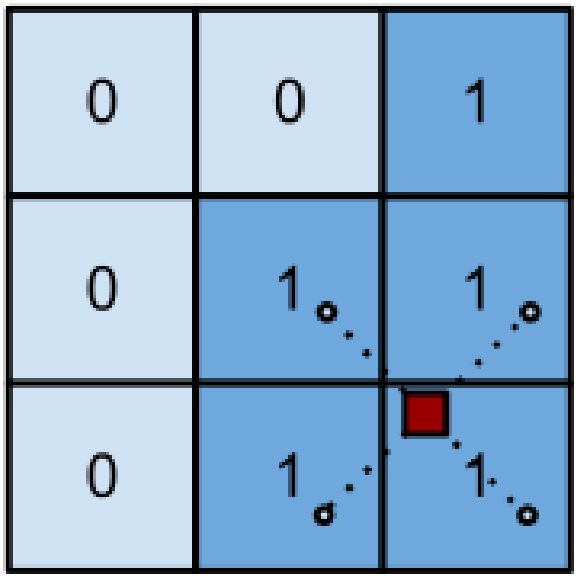
Figur A42: *Dette er et eksempel på et udsnit af et shadowmap. Der hvor værdierne er 1 er der hvor der skal være skygge i scenen set fra kameraet.*



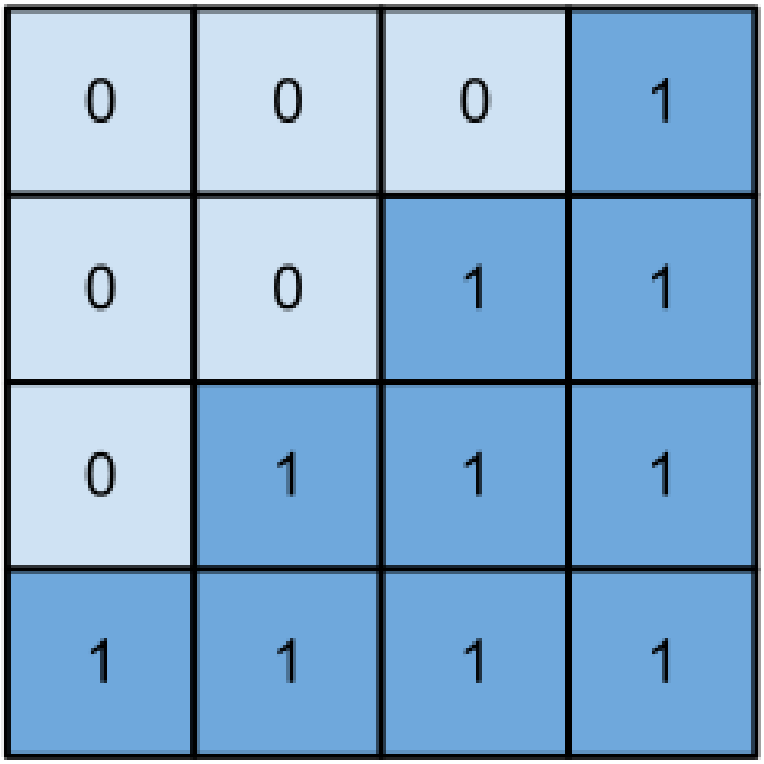
Figur A43: *I dette tilfælde vil det opslåede punkt ligge* $\frac{3}{4}$ *skygge. det vil sige skyggen kun er* $\frac{3}{4}$ *så kraftig som den hvis punktet havde ligget fuldkommen i skygge.*



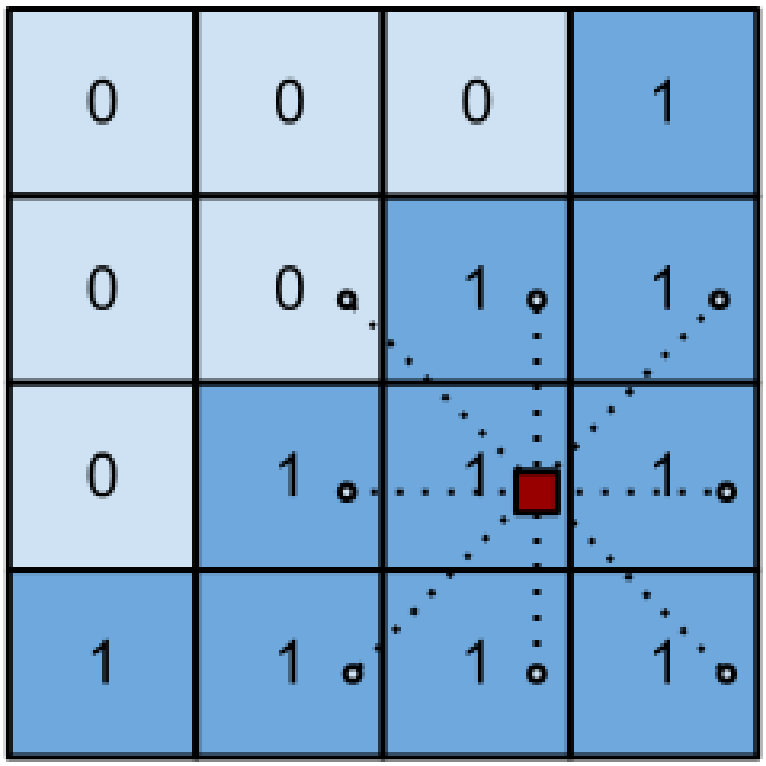
Figur A44: *Her er skyggens kun* $\frac{1}{4}$ *så kraftig*



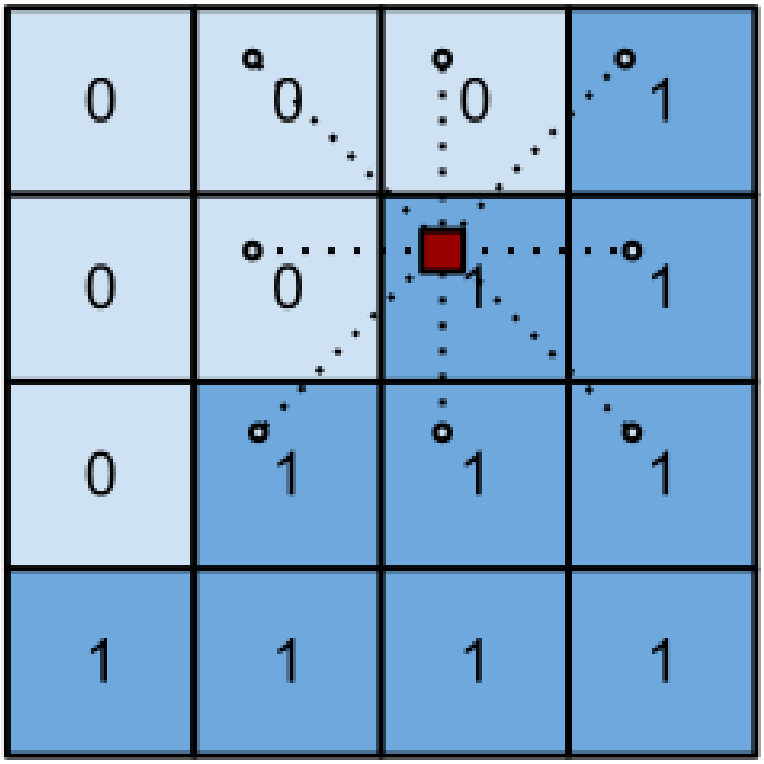
Figur A45: *Her ligger punktet fuldkommen i skygge.*



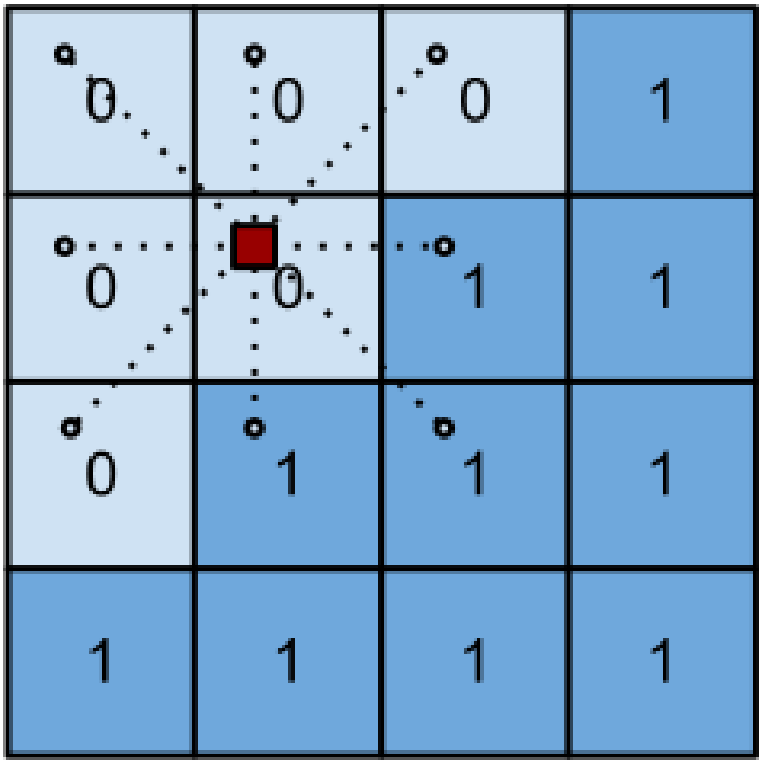
Figur A46: *Lige som før ser vi på et eksempel på et udsnit af et shadowmap. Der hvor værdierne er 1 er der hvor der skal være skygge i scenen set fra kameraet.*



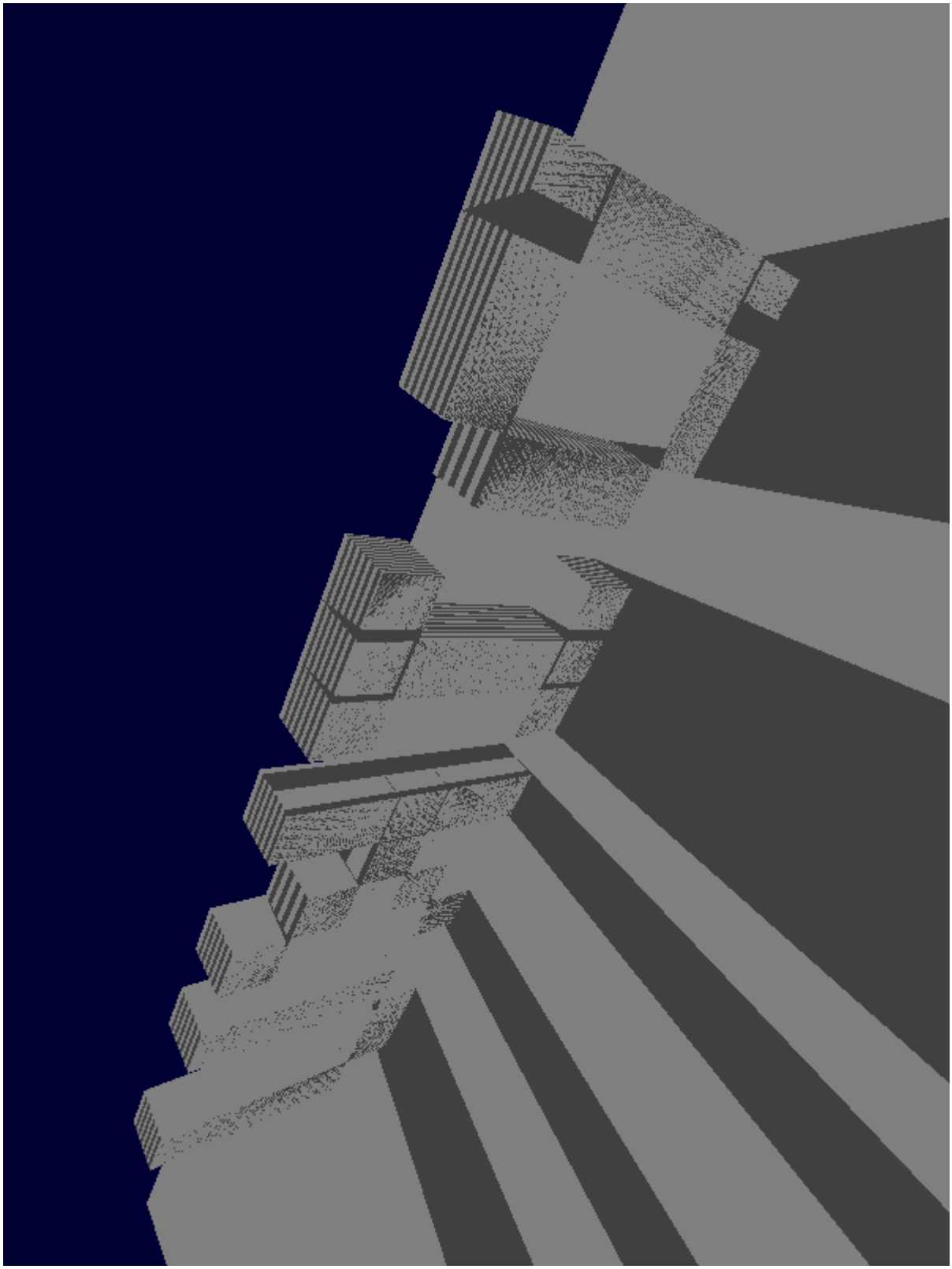
Figur A47: *Punktet ligger i* $\frac{8}{9}$ *skygge.*



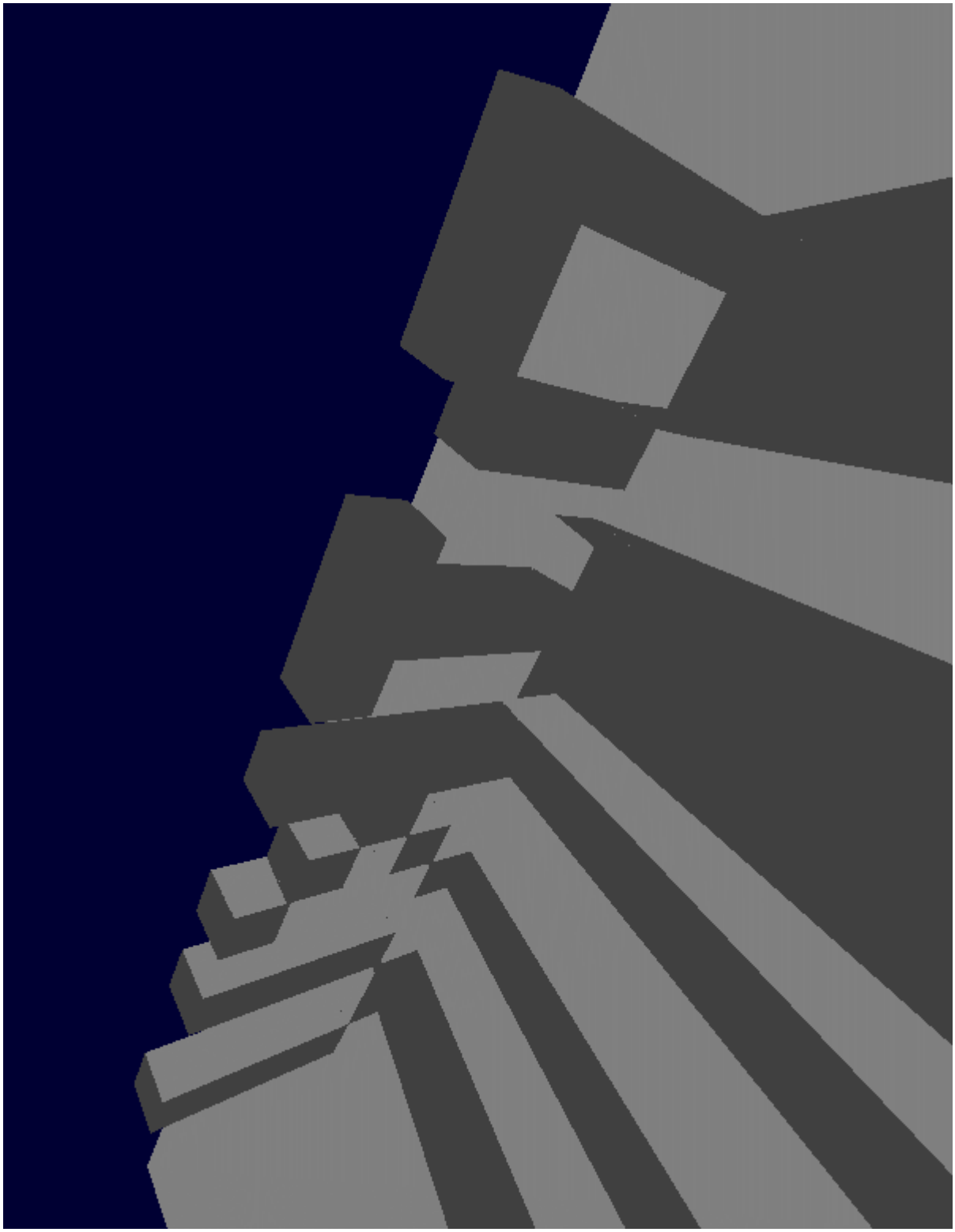
Figur A48: *Her er skyggens kun* $\frac{6}{9}$ *så kraftig som fuldkommen skygge.*



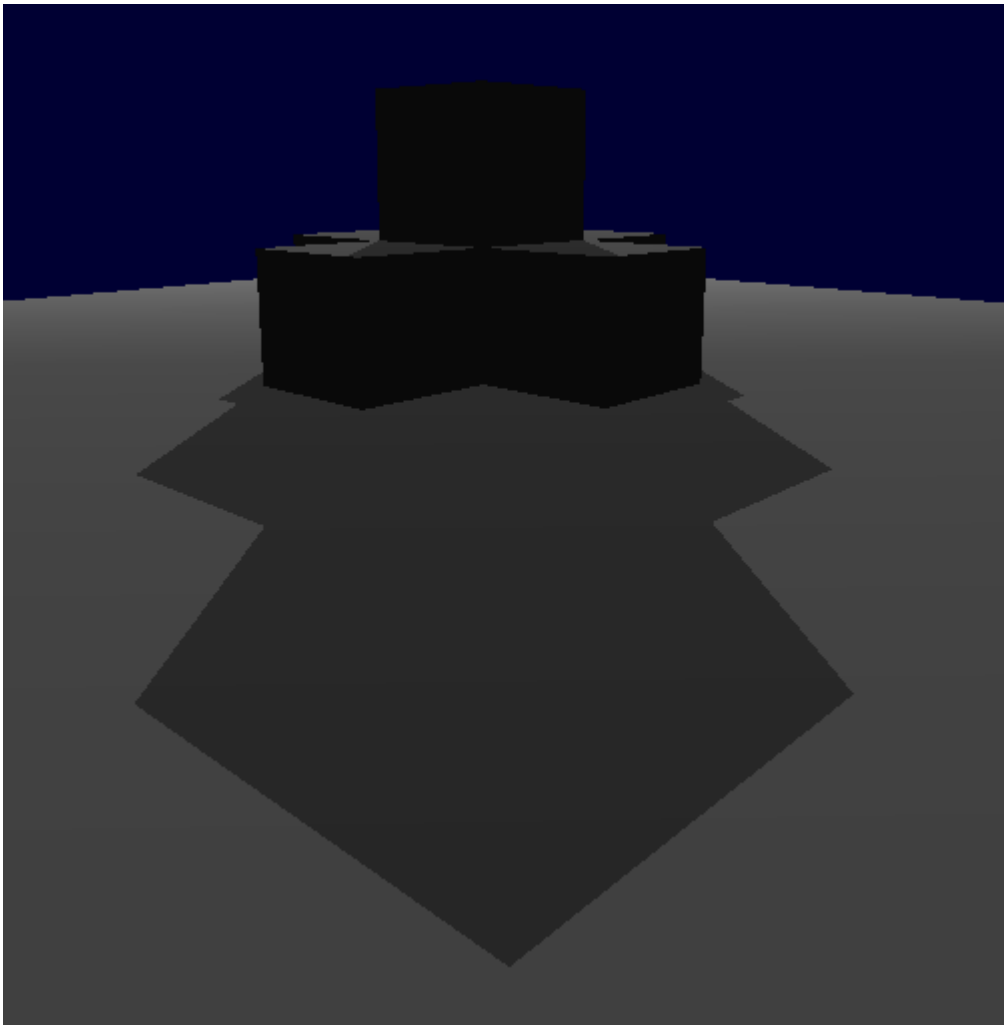
Figur A49: *Her er skyggens kun* $\frac{3}{9}$ *så kraftig som fuldkommen skygge.*



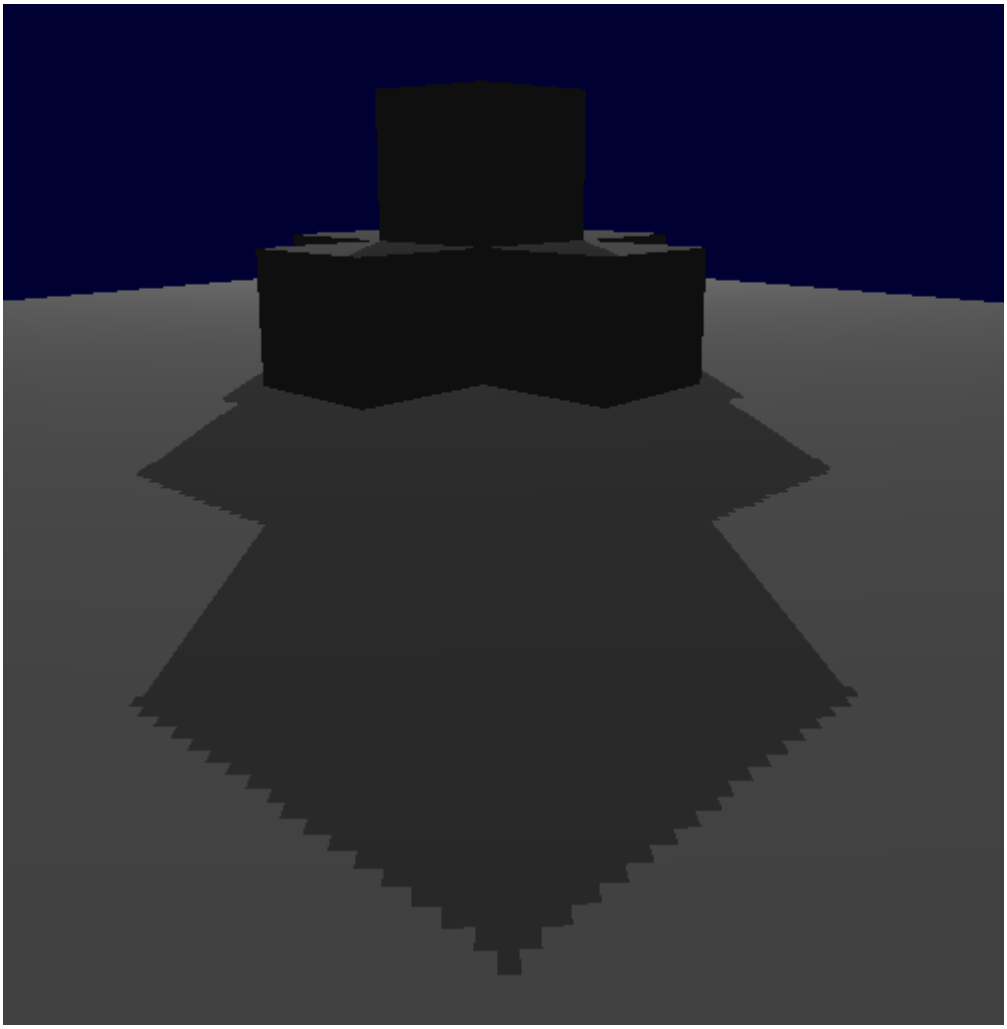
Figur A50: *Her kan man se at de er moire's pattern der hvor der burde være skygge på figurene selv. Selve de skygger som figurene kaster er der ingen problemer med.*



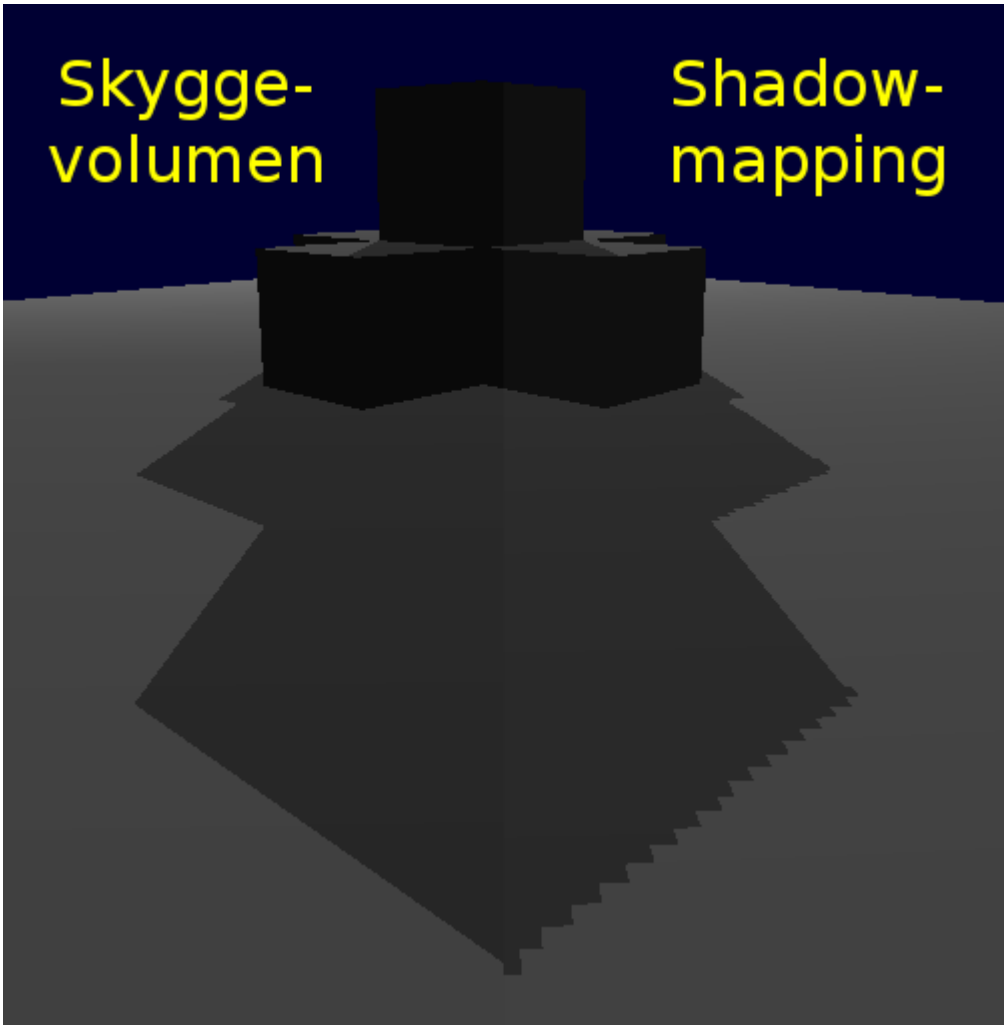
Figur A51: *Her kan man se at moire's pattern er forsvundet efter vi har tilføjet tjeket på normalerne.*



Figur A52: *Her kan man se at moire's pattern er forsvundet efter vi har tilføjet tjeket på normalerne.*



Figur A53: *Her kan man se at moire's pattern er forsvundet efter vi har tilføjet tjeket på normalerne.*



Figur A54: *Her kan man se at moire's pattern er forsvundet efter vi har tilføjet tjeket på normalerne.*

B Litteraturliste

[AMA02] Tomas Akenine-Möller and Ulf Assarsson. Approximate soft shadows on arbitrary surfaces using penumbra wedges. In *Proceedings of the 13th Eurographics workshop on Rendering*, EGRW '02, pages 297–306, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[BP07] Michael Bunnell and Fabio Pellacini.
GPU Gems - Chapter 11. Shadow Map Antialiasing. http://http.developer.nvidia.com/GPUGems/gpugems_ch11.html, September 2007.
Sidst besøgt: 01/05-2013.

[Cro77] Franklin C. Crow. Shadow algorithms for computer graphics. In *Proceedings of the 4th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '77, pages 242–248, New York, NY, USA, 1977. ACM.

[Kil96] Mark Kilgard. *OpenGL programming for the X Window System*. Addison-Wesley Developers Press, Reading, Mass, 1996.

[Kil03] Cass Everitt & Mark J. Kilgard.
Robust Stenciled Shadow Volumes. http://www.slideshare.net/Mark_Kilgard/robust-stenciled-shadow-volumes, Marts 2003.
Sidst besøgt: 21/05-2013.

[McC00] Michael D. McCool. Shadow volume reconstruction from depth maps. *ACM Transactions on Graphics (TOG)*, 19(1):1–26, 2000.

[Mic] Microsoft.
What Is a Stencil Buffer? http://msdn.microsoft.com/en-us/library/bb976074.aspx.
Sidst besøgt: 02/05-2013.

[Ope] OpenGL. Opengl shading language (glsl) reference pages. http://www.opengl.org/sdk/docs/manglsl/.

[Ope11] OpenGL. Opengl 2.1 reference pages. http://www.opengl.org/sdk/docs/man2/, 2011.

[OT13] Tutorial 16 : Shadow mapping — opengl-tutorial.org.
http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-16-shadow-mapping, April 2013.
Sidst besøgt: 25/04-2013.

[Por] Banu Octavian (Choko) & Brett Porter.
NeHe Productions: Shadows. http://nehe.gamedev.net/tutorial/shadows/16010/.
Sidst besøgt: 29/05-2013.

[Ros10] Randi Rost. *OpenGL shading language*. Addison Wesley, Upper Saddle River, NJ, 2010.

[San08] Fabien Sanglard.
ShadowMapping with GLUT and GLSL. http://www.fabiensanglard.net/shadowmapping/index.php, 2008.
Sidst besøgt: 31/05-2013.

[Shr04] Dave Shreiner. *OpenGL reference manual : the official reference document to OpenGL, version 1.4*. Addison-Wesley, Boston, 2004.

[Shr10] Dave Shreiner. *OpenGL programming guide : the official guide to learning OpenGL, versions 3.0 and 3.1.* Addison-Wesley, Upper Saddle River, NJ, 2010.

[Swi] Swiftless.
22. OpenGL Camera – Swiftless Tutorials - Game Programming and Computer Graphics Tutorials. http://www.swiftless.com/tutorials/opengl/camera.html.
Sidst besøgt: 12/04-2013.

[Wil78] Lance Williams. Casting curved shadows on curved surfaces. In *Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '78, pages 270–274, New York, NY, USA, 1978. ACM.

[YK02] Hun Yen Kwoon.
The Theory of Stencil Shadow Volumes - Graphics Programming and Theory.
http://www.gamedev.net/page/resources/_/technical/graphics-programming-and-theory/the-theory-of-stencil-shadow-volumes-r1873,
December 2002.
Sidst besøgt: 17/04-2013.