Subdivision Surfaces

subdivision silhuet-søgning

Eksamensprojekt ved institut for

Informatik og Matematisk Modellering

af Mikkel Gjøl

rev. 1.01

31. Januar 2005

Abstract

In this project a novel method has been developed for correction of silhouettes in low-polygon visualization of subdivision surfaces.

Visualization using ordinary subdivision generates nice, smooth surfaces, but is unfit for realtime use due to the resulting amount of geometry. Instead the visualization can be based on a coarse approximation to the subdivision surface, constructed using a small number of subdivision iterations. It is possible to improve the visual approximation by using normalmapping. This makes the inner part of the surface look smooth while the silhouette still reveal the coarse nature of the approximating surface. By combining normalmapping with the proposed method, the silhouette is corrected resulting in great likeness to the subdivision limit-surface.

The presented method finds contour points on the limit surface for a Loop subdivision surface. It then creates an approximation to the silhouette of the surface based on these points. Points on the contour is found rapidly by using table-based subdivision of the control-mesh edges. Thus, the model can be rendered in it's coarse form while maintaining a smooth silhouette.

Silhouette-correction works on dynamic objects and is fast enough for realtime visualization.

Resumé

Der er i dette projekt udviklet en metode til korrektion af silhuetter, ved lavpolygon visualisering af subdivision surfaces.

Sædvanlig visualisering ved subdivision-underinddeling giver pæne bløde flader, men er uegnet til anvendelse i realtid grundet den store mængde geometri der resulterer. I stedet kan anvendes en grov approksimation til subdivision-fladen, givet ved kontrolmodellen efter få subdivision-iterationer. En forbedring af visualiseringen kan opnås ved brug af normalmapping, der får det indre af fladen til at fremstå blød, mens silhuetterne afslører den lave detaljegrad. Ved kombination med den præsenterede metode korrigeres silhuetten, hvorved visualisering der har stor lighed med grænsefladen, opnås.

Den udviklede metode finder konturpunkter på grænsefladen til en Loop subdivision surface, og danner herudfra en approksimation til fladens silhuet. Konturpunkterne findes ved hurtigt at foretage en søgning på alle kanter i den oprindelige model, ved hjælp af tabelbaseret subdivision. Herved kan modellen renderes i sin lavdetaljerede form, mens konturerne stadig fremstår bløde.

Silhuetkorrektionen kan foretages på dynamiske objekter, og giver hastigheder anvendelig til visualisering i realtid.

Forord

Denne rapport dokumenterer gennemførelsen af et eksamensprojekt ved Danmarks Tekniske Universitet, og er sidste krav for opnåelse af civilingeniørgraden. Eksamensprojektet er udført hos billedgruppen, ved institut for Informatik og Matematisk Modellering på DTU frem til 31. januar 2005, under vejledning af lektor Niels Jørgen Christensen.

Jeg vil indlede med at takke en række mennesker, der har hjulpet og støttet mig under udarbejdelsen af dette projekt.

- Niels-Jørgen Christensen, for gode råd og vejledning, samt sin indsats for computergrafik i Danmark – uden denne ville mange unge begavelser gå tabt.
- Andreas Bærentzen, for interesseret, indsigtsfuld og intelligent vejledning projektet igennem.
- Bent Dalgaard Larsen, for livlige diskussioner og indblik i hemmeligheder.
- Jakob Welner, Mikkel Jans og Henrik Bjerregaard Clausen for at låne mig deres fantastiske 3D-modeller!
- Henrik Søndergaard, for veludført rolle som lidelsesfælde, samt stor forståelse og opmuntrende bemærkninger.
- En stor tak også til de der har hjulpet mig med korrekturlæsning af rapporten. Andreas Bærentzen, Bent Dalgaard Larsen , Kim Gjøl, Mark Gjøl, Jan Marguc, Henrik Aasted Sørensen, og Morten Skaaning.
- Tak også til min familie og mine venner for støtte og opmuntring, samt lejlighedsvist at foregive, at de stadig kan huske hvordan jeg ser ud.

Hernæst vil jeg undskylde for den anglificerede brug af sproget dansk. På et område som dette er det vanskeligt at undgå anvendelsen af engelske gloser. For dog ikke at forvirre begreberne, har jeg valgt at anvende ord og udtryk direkte lånt fra litteraturen. Disse vil således være på engelsk. Det er søgt løbende at forklare ord og begreber efterhånden som de bruges.

Det antages at læseren har baggrund i computergrafik, samt besiddelser god forståelse for algoritmik, grafteori og lineær algebra.

Mikkel Gjøl, s971661, 31 januar 2005.

Indholdsfortegnelse

1 Indledning

1.1 Baggrund

En af de primære anvendelser for computere, har altid været til beskrivelse af den verden vi lever i. De første computere blev designet for at foretage beregninger af projektilbaner, og ikke mange områder er siden gået fri for at blive eftergjort i det virtuelle rum. Et af de områder hvor computere hurtigt vandt indpas, var i design og fremstilling af plader til skibe og biler. På disse områder var præcision af afgørende betydning, og computere repræsenterede en måde at fremstille ensartede gode resultater. Anvendelsen af flader der kunne beskrives matematisk og efterfølgende evalueres maskinelt, gav mulighed for konsekvent højere præcision i produktionen. De former man ønskede at beskrive, bar ofte præg af at være "bløde" af udseende. Til at begynde med handlede det primært om at skabe biler med et særligt aerodynamisk præg. Særligt to producenter blev kendt for deres bidrag til forskningen på området; Renault-, og Citroén- fabrikkerne udviklede i slutningen af 1960erne, sideløbende, det der i dag er navngivet "Beziér"-flader. Brugen af computer-assisteret-design og -fremstilling (CAD/CAM[1]) gjorde ikke blot formgivningen af fladerne lettere, men de konstruerede flader besad også hensigtsmæssige matematiske egenskaber, der gjorde de resulterende fysiske flader mere robuste.

Beziér-flader har dog også en række mindre heldige egenskaber, som senere har været genstand for forbedringer. I første omgang af DeBoor *[DEBOOR1, DEBOOR2]*, der med baggrund i Beziér-flader introducerede de nu meget anvendte B-splines, der, udover at kunne evalueres hurtigt, også er lettere at kontrollere. Der er dog en række restriktioner på hvilke typer flader der kan modelleres med B-splines – specifikt er de begrænset til at beskrive modeller via firkant-lapper. I 1978 generaliserede henholdsvis Catmull og Clark, og Doo og Sabin, B-spline teorien til at omfatte arbitrær geometri. Disse nye generaliserede B-splines fik den samlede betegnelse "Subdivision surfaces", efter den iterative underinddeling der udgjorde deres konstruktion. *[SABIN1, CATMULL]*

På trods af de nye muligheder der var i subdivision surfaces, var de reelle anvendelser i de følgende år ret sparsomme. Den manglende udbredelse kan primært tilskrives, at gode og robuste værktøjer til brug af B-splines allerede eksisterede, samt at selve evalueringen af subdivision fladerne var ganske ressourcekrævende. Yderligere er matematikken knyttet til subdivision surfaces meget kompliceret, hvilket i høj grad begrænsede anvendelsesområderne. Et eksempel er, at det ikke har været muligt at evaluere en subdivision flade i et arbitrært punkt, indtil for ganske nylig *[STAM1, STAM2, ZORIN2]*.

Efterhånden som computere er blevet mere kraftfulde og der er opnået større indsigt i matematikken omkring subdivision flader, er emnet dog taget op igen. Fladerne har især fundet anvendelse indenfor beskrivelse af organiske former, men vinder efterhånden også indpas i industrielt design. Ikke mindst filmindustrien har taget fladerne til sig, hvor især animationsfirmaet Pixar har markeret sig med deres animationspakke "Marionette", der er baseret på Catmull-Clark flader[2]. I dag er netop Catmull-Clark subdivision flader blevet industri-standard, og findes i en eller anden form i alle større kommercielle animationssystemer.

1 CAD, Computer-Aided-Design – CAM, Computer-Aided-Manufacturing.
2 Ed Catmull er medstifter af Pixar, der blev grundlagt i 1986.

Illustration 1.1: Til venstre et typisk eksempel på de bløde linier der ses i meget nyere design. Til højre et billede af figuren "Gollum" fra Lord of The Rings. Modellen blev i høj grad skabt af Bay Raitt, der har bidraget til at drive udviklingen af subdivision surface-modellering fremad.

1.2 Anvendelser i sammenhæng med realtidsvisualisering

En grundlæggende egenskab ved subdivision surfaces er den iterative underinddeling af fladerne, der generelt resulterer i eksponentielt mange trekanter i forhold til basis-modellen. Dette er ikke et væsentligt problem når slutresultatet er en rendering[3] hvor tid og ressourcer kan dedikeres til processen. I realtidsanvendelser er der dog generelt meget strenge grænser for, hvilken geometrisk kompleksitet der er mulig.

Anvendelserne af subdivision surfaces i realtidssammenhænge har da også været sparsomme. Hardwareproducenten Nvidia har lavet en enkelt proof-of-concept demo, Ogre, der anvender adaptiv underinddeling af geometri. Tillige præsenteres i *[BRICKHILL]* en metode udviklet til anvendelse på Sony's konsol, Playstation 2. Denne platform har meget stor regnekraft, men forholdsvist begrænsede lagerressourcer. Det er således meningsfyldt at danne geometrien for hvert billede lige inden den tegnes, for herefter at smide den væk. En lignende tendens er at se på PC-

3 Faktisk anvender Pixar's meget udbredte program "Renderman" som udgangspunkt en metode, der først underinddeler alle flader, og herefter scanline konverterer dem. Dette sker for at kunne lave præcis og hurtig sampling af fladerne, uden at skulle anvende Raytracing.

platformen, hvor geometri lagres statisk på grafikkortet. I en såkaldt vertex-shader, kan den statiske geometri herefter ændres lige inden den tegnes, hvorefter ændringerne smides væk. Den afgørende forskel i forhold til ps2 er, at ny geometri ikke kan skabes, hvilket reelt gør det umuligt at evaluere subdivision surfaces ved rendering.

På det seneste er en metode, med udgangspunkt i et paper fra 1997 *[COHEN01]*, blevet meget udbredt ved visualisering af detaljeret geometri. Under navnet ”normalmapping” kan metoden kort beskrives som en sampling af en række parametre på en detaljeret flade – primært farve og fladens normal, som senere kan anvendes til at forbedre udseendet af en lavdetaljeret udgave af samme flade. Metoden giver mulighed for en fuldstændig adskillelse af den højdetaljerede og den lavdetaljerede model, hvilket giver meget frie tøjler for hvordan detaljerne i den skabes. Eksempelvis er bumpmapping, multitexturing og lignende mere avancerede teknikker fuldt legale. Tilsvarende er detaljegraden og den geometriske struktur af den detaljerede model uden betydning, hvilket gør det fuldt ud muligt at anvende subdivision surfaces til at skabe disse modeller.

Illustration 1.2: Billeder fra ID Softwares Doom3, der gør god brug af normalmapping. Silhuetten lader dog meget tilbage at ønske.

Et af de steder hvor metoden har fået mest omtale, er i ID Softwares computerspil Doom3, der gør intensiv brug af normalmapping. På Illustration 1.2 ses et par billeder fra spillet. Det er tydeligt, at normalmapping giver en god illusion af detaljerigdom, selv på meget lavdetaljerede objekter. Det væsentligste problem med modellerne, er den meget kantede silhuet, der bryder illusionen og åbenlyst afslører den simple geometri[4].

I forprojektet til dette eksamensprojekt, blev det vist, at en tilsvarende metode kunne anvendes til at tilnærme udseendet af en polygon til dens subdivision grænseflade. Analogt til ovenstående eksempel, blev illusionen om "bløde" flader brudt nær den kantede silhuet. I tidligere arbejde *[PULLI, HAVEMAN1, HAVEMAN2]* er vist metoder til at forbedre disse silhuetter via underinddeling. Metoderne har hovedsageligt været koncentreret om rekursiv underinddeling af fladerne samt håndtering af redundans i beregningerne, hvorfor muligheden for pæne silhuetter primært er betragtet som en sidegevinst.

1.3 Formål

Hovedformålet med dette projekt er, at undersøge hvorledes kontur- og silhuet-kanter kan dannes for en subdivision flade, uden eksplicit at foretage en fuld, rekursiv underinddeling af fladen. Projektet er rettet imod anvendelse til visualisering af subdivision flader i realtid. Visualiseringen skal være at forveksle med modellens grænseflade, med særligt fokus på ikke at give skarpe kanter grundet polygonalisering.

Udgangspunktet for projektet er en antagelse om, at det må være muligt at finde punkter der ligger præcis på et objekts silhuet, alene ud fra kendskabet til kontrolfladen. Idet vi kan beregne både positioner og normaler for grænsefladen, kan en søgning foretages efter punkter på fladen, hvis normal er ortogonal på øjevektoren. Når disse silhuetpunkter er fundet kan en visualisering af subdivison fladerne foretages via simpel triangulering af fladen omkring disse punkter. Kombinationen med den normalmapping-metode der blev udviklet i forprojektet gør det potentielt muligt at foretage denne triangulering uden hensyntagen til problemer med shading-artefakter, der ellers kan opstå ved brug af f.eks. per-vertex belysning. En fuld søgning efter silhuetpunkter henover fladen er en ganske tung operation, der kræver beregning af meget information om subdivision grænsefladen. Den hurtigste metode til beregning af subdivision-flader er på nuværende tidspunkt via tabelbaseret beregning, som beskrevet i *[BRICKHILL, SCHRÖDER3]*. Projektet vil således tage udgangspunkt i disse metoder, i søgningen efter silhuetpunkter.

For at vise anvendeligheden af den fremsatte hypotese, vil der blive programmeret en prototype der skal give indsigt i den praktiske implementering af metoden. Denne prototype vil også ligge til grund for den senere evaluering af metoden, i henhold til såvel hastighed som visuel kvalitet.

Idet kombinationen med forprojektet kan ses som en samlet løsning til visualisering af subdivision surfaces, er det væsentligt at den her udviklede metode kan fungere i sammenhæng med den da udviklede "subdivision surface normalmapping".

Der vil i dette projekt blive gjort en indsats for at frembringe resultater der er praktisk anvendelige.

4 Det bør nævnes, at den lave detaljegrad ikke er direkte relateret til normalmapping, men hænger sammen med den i spillet anvendte skyggealgoritme.

Først og fremmest sigtes der efter en metode der kan anvendes på et bredt spektrum af subdivision surface-typer – inkl. sædvanlige udvidelser så som spidse punkter og skarpe kanter. Derudover skal metoden være anvendelig i realtid, på "forbruger"-niveau hardware. Projektet er således underlagt de hardware-mæssige begrænsninger der gør sig gældende på nuværende PCer, afsnit 5.5. Med udgangspunkt i Richard Huddy's mindeværdige ord "Have you tried just drawing the damn thing?" - er det yderligere et mål at konstruere en metode der er hurtigere end blot at tegne modellen subdivided så mange gange at den ikke er til at skelne fra grænsefladen.

Hastighedsoptimeringer er næsten altid en vægtning af effektivitet kontra nøjagtighed. Det bestræbes i denne rapport at angive de steder hvor denne type vægtninger kan foretages, og hvilke konsekvenser disse valg har for resultatet. Det væsentligste mål med projektet vil dog være at lave en god visualisering af subdivision fladen, hvorfor de fleste af disse valg træffes til fordel for nøjagtighed frem for hastighed.

1.4 Afgrænsning

Det er ikke målet med dette projekt at præsentere en fuldstændig løsning på problemet omkring subdivision surface visualisering. Der vil blive fokuseret på problemet omkring silhuetter da dette har været mest afgørende for, at de visualiserede flader ikke har fremstået "bløde". Andre kendte problemer vil således ikke blive behandlet i dette projekt – mest relevant i henhold til forprojektet ville være artefakterne relateret til afstand imellem grænseflade og den lav-detaljerede flade.

Udgangspunktet for metoden vil være en kontrolpolygon, som antages at være lavet med henblik på subdivision. Det er således ikke målet, som det er med eksempelvis PN-triangles, afsnit 6.5.2, at konstruere en metode der forbedrer udseendet på vilkårlige modeller.

Det er yderligere ikke målet, at præsentere en optimal implementering af den viste metode. Ved begyndelsen af et projekt som dette, kan det være vanskeligt at overskue alle de muligheder der bør undersøges. For derfor ikke at løbe ind i problemer, er den sikre men umiddelbart mere besværlige vej valgt. Således er det anvendte framework, herunder modelstrukturer og lignende, mere generelt end nødvendigt for den endelige løsning. Enklere og hurtigere implementeringer af den præsenterede metode kan derfor opbygges ud fra den her fremstillede grundlæggende metode.

Selvom det er et væsentligt mål at frembringe en metode der kan anvendes på flere subdivision metoder, vil der i dette projekt alene blive fokuseret på Loop subdivision surfaces. Metoden må ikke udelukke anvendelse af andre subdivision-metoder, men der vil ikke blive givet detaljer i henhold til implementeringen af metoden i forhold til disse. Den bedste løsning vil som regel være knyttet til det praktiske område. Subdivision-modelleringsprogrammer kræver eksempelvis hyppige, topologiske ændringer, mens landskabsvisualisering modsat udviser stor ensartethed i sine beregninger. Den bedste løsning vil således oftest være en kombination af flere metoder – projektet her præsenterer blot en ny metode til paletten af værktøjer.

1.5 Strukturering af rapporten

Det er i denne rapport søgt at give en fyldestgørende præsentation af det arbejde der er udført under

udarbejdelsen af eksamensprojektet. En del af dette arbejde har haft form af et litteraturstudie, hvor anvendeligheden af eksisterende metoder er undersøgt i henhold til subdivision surfaces.

I afsnit 2 beskrives den grundlæggende teori omkring spline kurver og flader. Herefter gennemgås generaliseringen til subdivision surfaces, og i afsnit 3 beskrives relevante udvidelser af metoden, med specifikt fokus på visualisering i realtid. I dette afsnit gennemgås også teorien for tabelbaseret subdivision, der efterfølgende ligger til grund for silhuetsøgningen.

I afsnit 4 beskrives litteraturen omkring detaljering af lavdetaljerede flader i realtid. De præsenterede metoder sættes i relation til silhuetter og brug i sammehæng med subdivision surfaces.

Hypotesen præsenteres i afsnit 5, efter en række overvejelser omkring silhuetter på subdivision surfaces, og muligheder for silhuet-korrektion via kantsøgning.

En del af projektet er at udvikle en prototype på denne hypotese. Denne er beskrevet i detaljer i afsnit 6 omkring implementering. I afsnit 7 og 8 præsenteres og diskuteres resultater frembragt med denne prototype, og en analyse af metoden foretages.

Slutteligt foretages en konklusion, og en opsummering af udvidelser af metoden samt forslag til fremtidigt arbejde præsenteres.

Dette projekt har været opdelt i to – et forprojekt og et hovedprojekt. Forprojektet omhandlede, under den lidet dansk-klingende titel ”Subdivision surface normalmapping”, en metode til forbedring af lysberegninger på subdivision flader. Projektet viste en fremgangsmåde til, uden at tilføje geometri til en model, at bringe udseendet nærmere subdivision-grænsefladen. En del af det behandlede materiale i den foreliggende rapport overlapper naturligt det fra forprojektet, men gengives her for fuldstændighedens skyld.

Udgangspunktet for den grundlæggende teori er således uændret, men der er tilføjet to afsnit om B-splines på matrixform, afsnit 2.2.3 og 2.6, der skal lette overgangen til subdivision teori. Yderligere er der til afsnittet omkring subdivision surfaces, tilføjet et afsnit om analyse af konvergensforhold for subdivision metoder, og beregning af grænseværdier. En væsentlig del af dette projekt beskæftiger sig med hurtig beregning af subdivision surfaces ved brug af tabelopslag. Emnet blev også beskrevet i forprojektet, men er udvidet markant til en fuldstændig beskrivelse af teknikken.

2 Teori for bløde kurver og flader

I dette afsnit vil den grundlæggende teori bag subdivision surfaces blive gennemgået. Sammenhængen med B-spline teori og den iterative definition af fladerne vil blive forklaret. Yderligere vil subdivision matricer, der er en anden måde at angive subdivision, kort blive beskrevet, og baggrunden for beregning af grænse-positioner og -normaler vil blive ridset op. Teorien omkring analyse af konvergens for subdivision flader for ekstraordinære punkter vil ikke blive berørt, og der henvises til udførlige analyser andetsteds *[SCHWEITZER, ZORIN1]*.

Subdivision surface teori er i bund og grund en generalisering af B-splines. Essensen i dette er, at regulære dele af subdivision flader er identiske med B-splines. En forståelse af B-spline teori er derfor en forudsætning for indsigt i subdivision surfaces. Efter en kort gennemgang af først bezier- og B-spline kurver og dernæst flader, vil styrker og svagheder ved disse blive diskuteret. Subdivision flader vil herefter blive gennemgået i en form, der umiddelbart kan implementeres.

Udgangspunktet for design af de kurver der beskrives i de følgende afsnit, er en stykkevist lineær kontrolpolygon, der styrer formen af de dannede kurver. En fordel ved denne måde at designe kurver er, at den er intuitiv, idet de resulterende kurver tydeligt følger kontrolpolygonen. Tilføjelse af detaljer sker ligeledes let, ved at tilføje ekstra punkter i det område detaljerne ønskes. En måde at danne en kurve ud fra kontrolpolygonen er, at vægte kontrolpunkterne ud fra en matematisk funktion. Den simpleste funktion er blot en lineær vægtning af nabopunkter, hvilket giver kontrolpolygonen selv.

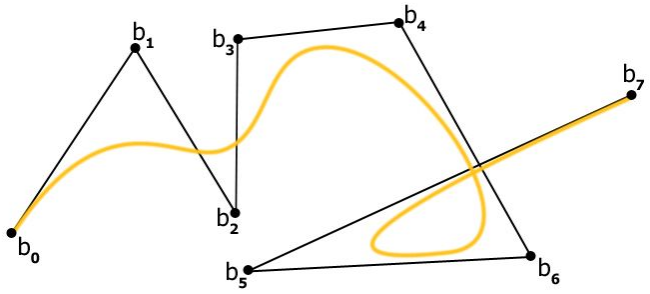


Illustration 2.1:
Eksempel på blød kurve konstrueret ud fra en kontrolpolygon.

De to følgende afsnit omhandler Beziér-kurver og B-splines, der er eksempler på en sådan vægtning af kontrolpolygonen. Som en del af gennemgangen, vises også en anden måde at konstruere bløde kurver, nemlig ved iterativ underinddeling. Denne tankegang er grundlæggende i den senere generalisering til subdivision flader.

2.1 Bezier Kurver

Bezier-kurverne blev udviklet uafhængigt af Paul de Casteljau og Pierre Bézier, til brug inden for design af biler. Kurverne fik navn efter Bezier, da han modsat Casteljau, publicerede en række artikler om emnet. Casteljau har til gengæld lagt navn til den mest anvendte metode til beregning af kurverne.

Der er mange måde at beregne Bezier-kurver på - en af dem, er ved at repræsentere dem ved Bernstein-polynomierne:

$$B_i^n(t) = \binom{n}{i} t^i (1-t)^{n-i}, \quad i = 0 \ldots n$$

$$-hvor$$

$$\binom{n}{i} = \frac{n!}{(n-i)!i!}, \quad i = 0, \ldots, n, \quad 0! = 1$$

formel 2.1

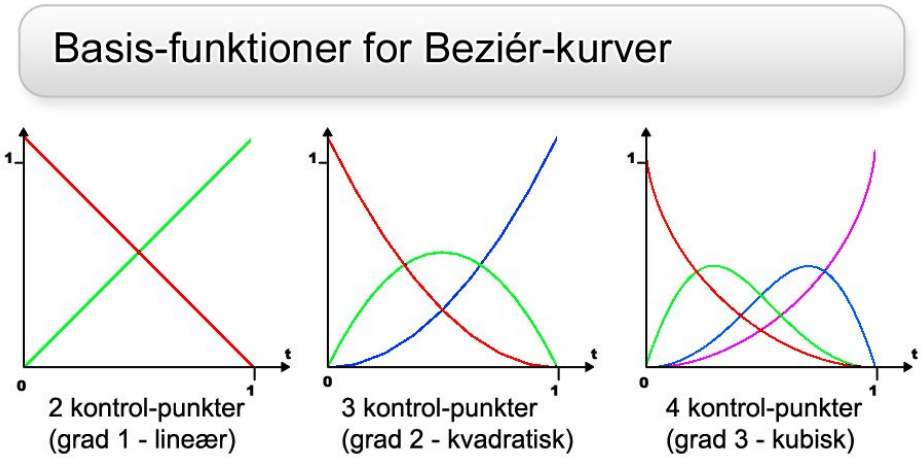


Illustration 2.2: Bernstein polynomierne.

Det ses, at Bernstein polynomierne summerer til 1 overalt, og ligger i intervallet [0,1] – associeres hvert polynomie med en del af kontrolpolygonen, dannes affine kombinationer af kontrol-punkterne. Det er herved muligt at opstille ligningen for en kurve, **r**(t), der alle steder gives ud fra en vægtning, af kontrol-punkterne, $\boldsymbol{b_0} \ldots \boldsymbol{b_i} \ldots \boldsymbol{b_n}$.

$$\boldsymbol{r}(t) = \sum_{i=0}^{n} B_i^n(t)\,\boldsymbol{b_i}, \quad t \in [0;1]$$

formel 2.2

Idet Beziér-kurver baserer sig på polynomier, betegnes de polynomielle kurver. Det observeres at alle kontrolpunkter påvirker alle dele af kurven. Dette er en ufordelagtig egenskab, da kurver med mange kontrolpunkter fører til en række problemer. Dels vil manipulation medføre mange beregninger[5] idet hele kurven skal genberegnes, dels er det uintuitivt, at den ene ende af kurven ændrer sig, hvis man hiver i kontrolpunkterne i den anden ende. Det bemærkes, at kurven interpolerer kontrolpolygonen i endepunkterne.

Beziér-kurver er lette at differentiere[6], og har maksimal differentiabilitet, idet de er polynomielle. En kurve med *n* kontrolpunkter, ligeledes af grad *n*, er således C^{n-1} . En kurve siges at være C^n

5 Evaluering af et Bezier-kurvestykke med *n* knudepunkter, tager $O(n^2)$.
6 Faktisk er de afledte givet automatisk ved anvendelse af De Casteljau's algoritme

kontinuert, hvis den n'te differentierede af kurven overalt er kontinuert og forskellig fra nul. En ret linie, $y = ax + b$ er således C^1, da den anden afledede er 0. Man kan tale om "lokal kontinuitet" f.eks. i samlingspunkter mellem to kurver.

De Casteljau algoritmen til evalueringen af Bezier-splines bygger på forward differencing, der er en generel numerisk metode til evaluering af polynomier. En geometrisk konstruktion af beziér-kurver kan foretages meget enkelt. I formel 2.2 ses det, at en beziér-kurve er parametriceret over intervallet $t \in [0;1]$. For at finde punktet på kurven svarende til en given værdi af t gøres følgende: Hver kant i kontrolpolygonen antages ligeledes at være parametriceret over intervallet [0;1]. Der dannes nu en ny kontrolpolygon, bestående af et punkt fra hver kant i den tidligere kontrolpolygon. Punktet fra hver kant, er givet ved en lineær interpolation mellem dens to endepunkter, vægtet med værdien *t*. Denne fremgangsmåde gentages, indtil den nye kontrolpolygon kun består af et enkelt punkt. Dette punkt ligger på beziér-kurven, se illustration 2.3.

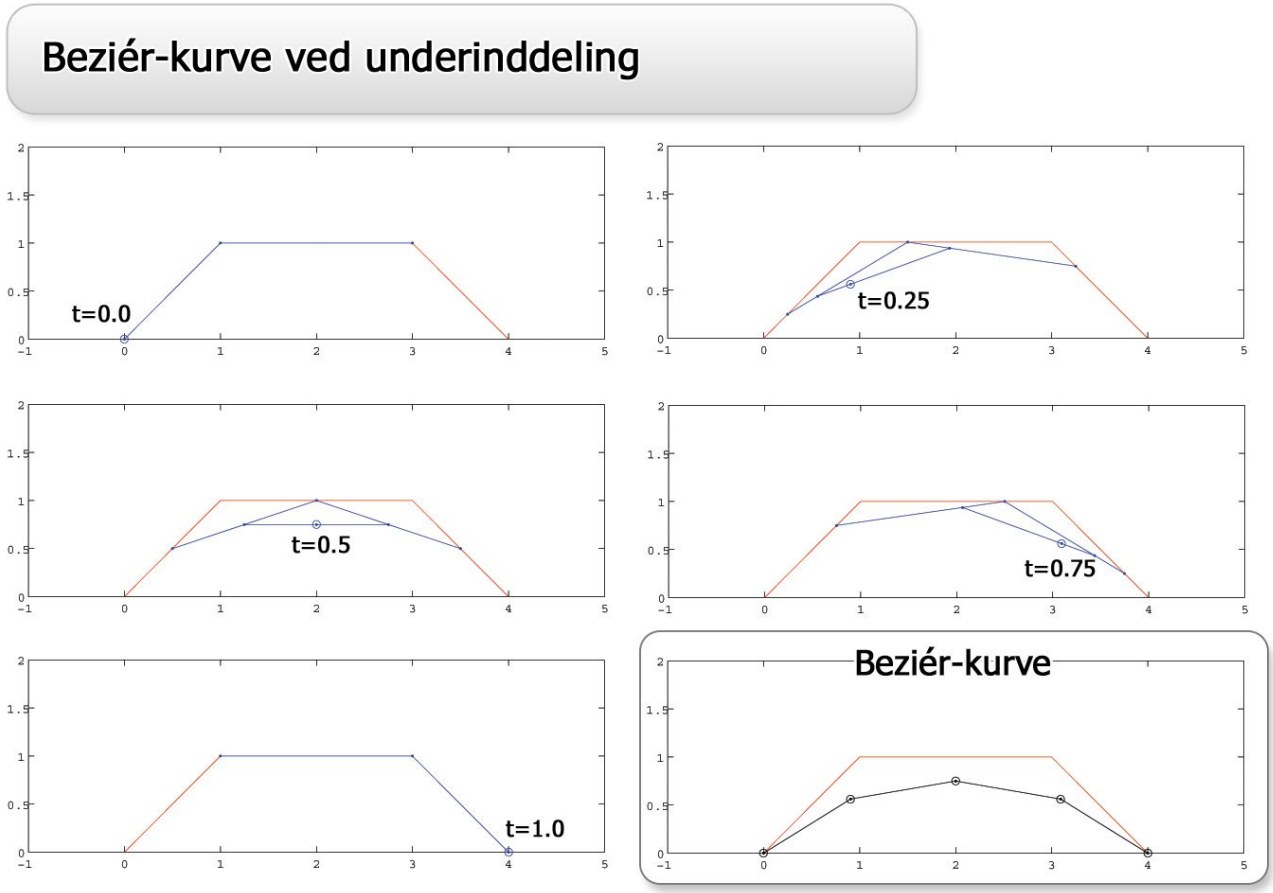


Illustration 2.3
Beziér-kurve evalueret ved Casteljau's algoritme - punkter markeret med et 'O' er de endelige punkter på kurven.

2.2 B-spline Kurver

Ordet 'spline' kommer fra skibsbyggeri, hvor man tidligere anvendte trælister[7] (splines) når man tegnede buede kurver. B'et står for basis, og hentyder til, at B-splines har en stykkevist polynomiel basis. B-spline kurver kan betragtes som en række Beziér-kurver, der er stykket sammen med samme kontinuitet som hver af Beziér-kurverne. B-spline kurven siges at have samme grad som de Beziér stykker den består af. En række af de gode egenskaber ved Beziér-kurver, arves direkte af B-splines, mens de værste ulemper som f.eks. global indflydelse undgås.

En B-spline af grad n er givet ved en kontrol-polygon med N punkter, samt en ikke-aftagende knudefølge af længde $N+n$ (+2 dummy-knuder afhængig af implementeringen). Som før kan vi angive en basis-funktion, og definere kurven ud fra denne. Basisfunktionen for det i'te segment af en B-spline kurve af grad n, er givet ved:

$$\boldsymbol{B}_{i,n}(t) \;=\; \otimes_0^n B_{i,0}(t),\;\; \otimes \textit{ angiver foldning},\quad \boldsymbol{B}_{i,0}(t)=\begin{cases}1, & t_i \leqslant t < t_{i+1}\\ 0, & otherwise\end{cases}$$

$$\Rightarrow\; \boldsymbol{B}_{i,n}(t) \;=\; \frac{t-t_i}{t_{i+1}-t_i}\boldsymbol{B}_{i,n-1}(t) + \frac{t_{i+2}-t}{t_{i+2}-t_{i+1}}\boldsymbol{B}_{i+1,n-1}(t)$$

formel 2.3

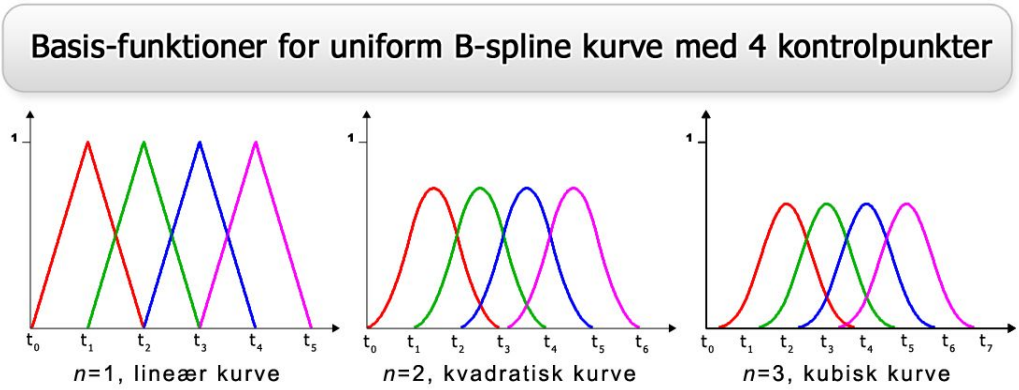


Illustration 2.4
Bemærk at overlappet imellem funktionerne vokser med graden.

B-spline-kurven kan således dannes som

$$\boldsymbol{r}(t)=\sum_{i=0}^{N}\boldsymbol{B}_{i,n}(t)\boldsymbol{b}_i$$

formel 2.4

Fordelen ved at anvende denne basis-funktion, fremfor bernstein-polynomier, er, at kurven har lokal støtte. Det betyder, at et givet punkt på kurven påvirkes af maksimalt $n+1$ kontrol-punkter, hvorfor det er muligt at foretage lokale ændringer til kurven. Yderligere giver denne kurve-repræsentation mulighed for, at "vægte" de enkelte kontrol-punkter, givet ved at variere en følge af tal, kaldet "knuder". De enkelte kontrol-punkter vægtes således, ved at ændre på knudefølgen. Hvis tallene i knudefølgen er uniformt stigende (0, 1, 2, 3...), kaldes kurven uniform. Stiger følgen ikke-uniformt,

7 De anvendte træ-lister var C^2 kontinuerte.

og er kontrolpunkterne således vægtet forskelligt, kaldes kurven tilsvarende ikke-uniform (”non-uniform”). *k* ens, på hinanden følgende knuder, siges at have multiplicitet *k*. En kurve af grad *n*, der indeholder knuder med multiplicitet *k*, er C^{n-k} kontinuert. Det følger heraf, at har en kurve knuder med multiplicitet højere end graden, vil der generelt opstå en diskontinuitet i selve kurven.

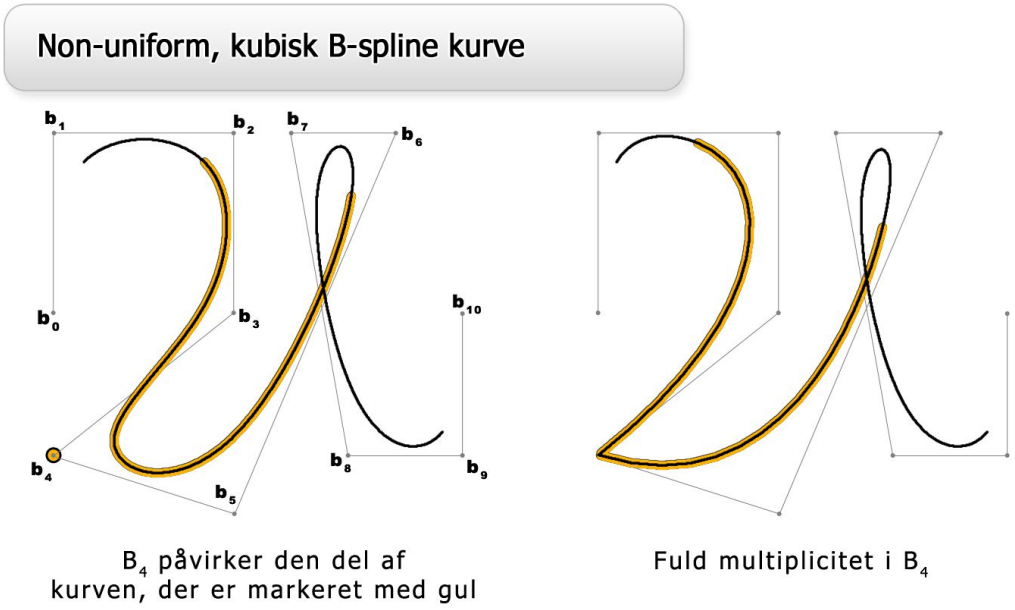


Illustration 2.5
B-splines har lokal støtte, og giver mulighed for at ”vægte” kontrolpunkterne ved at ændre knudefølgen. For rationelle kurver er det yderligere muligt at ændre de enkelte kontrolpunkters ”vægte”.

2.2.1 Rationelle B-splines

Selv for ikke-uniforme B-splines, er der væsentlige begrænsninger på mængden af de kurver der kan beskrives. Simple former som en cirkel, eller keglesnit generelt, kan ikke umiddelbart repræsenteres eksakt. En måde at gøre dette, er at specificere kontrolpolygonen i homogene koordinater *[RTR2]* – hvorved fås en rationel (”rational”) B-spline. Hvert kontrolpunkt gives således en ekstra koordinat, ω, der for værdien 1 giver samme kurver som ikke-rationelle B-splines. Rationelle kurver evalueres ved følgende forhold imellem polynomier,

$$\boldsymbol{r}(t) = \frac{\sum_{i=0}^{N} \omega_i \boldsymbol{B}_{i,n}(t)\boldsymbol{b}_i}{\sum_{i=0}^{N} \omega_i \boldsymbol{B}_{i,n}(t)}$$

formel 2.5

<**hindsight**: enten er ω i b_i 1, eller også evalueres kurven her i ikke-homogene koordinater>

Denne måde at angive kurver, hænger tæt sammen med projektiv geometri. Teorien vil ikke blive gennemgået her, og der henvises i stedet til litteraturen, eksempelvis *[GRAVESEN]*. Udover muligheden for at vægte de enkelte kontrolpunkter, har kurverne også den egenskab, at de er

invariante overfor perspektivisk projektion. Det betyder, at den perspektiviske projektionen af en B-spline kurve, ligeledes er en (rationel) B-spline – faktisk den samme B-spline, som var kurven selv projiceret. Dette er en væsentlig kvalitet i praksis, da det hermed er tilstrækkeligt at transformere kontrolpolygonen til skærmkoordinater, frem for alle evaluerede kurvepunkter.

En B-spline kurve med konstant afstand mellem knuder, mulighed for at styre knudemultiplicitet, samt mulighed for at vægte hvert enkelt punkt, kaldes en NURBS kurve – Non-Uniform Rational B-Spline.

2.2.2 Iterativ approksimation til B-spline kurver

Det er muligt at indsætte en ekstra knude i knudefølgen, uden at ændre formen af kurven. Det betyder dog, at kontrol-polygonen ændres. Indsættelse af en knude til fuld multiplicitet resulterer i, at det tilsvarende kontrolpunkt kommer til at ligge på selve B-spline kurven. Indsættes på denne vis knuder til fuld multiplicitet overalt i knudefølgen, haves B-spline kurven. Således givet en kontrolpolygon $\boldsymbol{b}^k$, hvori der indsættes knuder uniformt i midten af alle knudeintervaller, vil den nye kontrol-polygon, $\boldsymbol{b}^{k+1}$ tilnærme B-spline kurven. Fortsættes denne uniforme underinddeling rekursivt, vil kontrolpolygonen $\boldsymbol{b}^{\infty}$ i grænsetilfældet være B-spline kurven. Når denne kurve frembringes iterativt, betegnes den "grænsekurven", idet kontrolpolygonen blot konvergerer mod B-spline kurven.

Idet denne iterative algoritme danner den næste iteration ud fra en lineær vægtning af den foregående kontrolpolygon, kan den opstilles på matrixform *[STOLLNITZ]*. Kontrolpolygonen efter subdivision af en given iteration, er således givet ved[8]

$$
\begin{aligned}
\boldsymbol{b}^{k+1} &= \boldsymbol{S}_{\boldsymbol{b}^k}\boldsymbol{b}^k \\
\boldsymbol{b}^{k+2} &= \boldsymbol{S}_{\boldsymbol{b}^{k+1}}\boldsymbol{b}^{k+1} = \boldsymbol{S}_{\boldsymbol{b}^{k+1}}(\boldsymbol{S}_{\boldsymbol{b}^k}\boldsymbol{b}^k) = \boldsymbol{S}^2\boldsymbol{b}^k \\
&\vdots \\
\boldsymbol{b}^{k+i} &= \boldsymbol{S}^i\boldsymbol{b}^k = \boldsymbol{S}^{k+i}\boldsymbol{b}^0 \\
&\vdots \\
\boldsymbol{b}^{\infty} &= \boldsymbol{S}^{\infty}\boldsymbol{b}^0
\end{aligned}
$$

formel 2.6

- hvor $\boldsymbol{S}^i$ betegner subdivision matricen opløftet til potens i. Generelt har $\boldsymbol{S}$ samme antal søjler, som punkter i kontrolpolygonen, og vokser således for hver iteration, idet kontrolpolygonen vokser ved knudeindsættelse). Det bemærkes at $\boldsymbol{S}$ er uafhængig af de specifikke positioner for punkterne i $\boldsymbol{b}^k$, men afhængig af antallet af punkter. Det ses af formel 2.6, at hvis en potens af subdivision-matricen kendes, kan vi med en enkelt matrix-multiplikation finde kontrolpolygonen til en given iteration.

8 Omskrivningen til en S opløftet til en potens er lidt sær, idet $\boldsymbol{S}$ afhænger af $\boldsymbol{b}$. Meningen er dog, at det er muligt at springe subdivision-iterationer over, ved at subdivisionmatricerne sammen. Notationen vist her, følger litteraturen.

Uniform knudeindsættelse i kubisk B-Spline

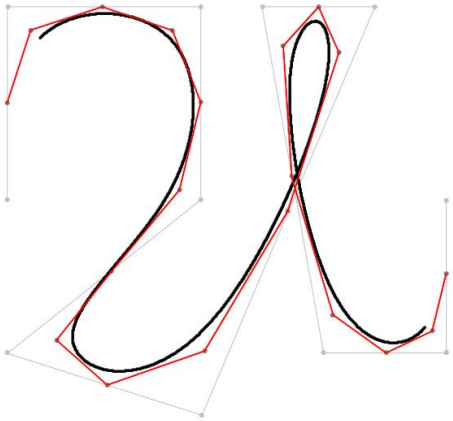


Illustration 2.6
Underinddeles alle knudeintervaller uniformt, fås en ny kontrolpolygon der tilnærmer den endelige B-spline.

2.2.3 B-splines angivet ved subdivision

Frembringelse af B-spline kurven via uniform knudeindsættelse, er identisk med den subdivision-metode der blev præsenteres i *[CHAIKIN]*, hvilket dog først blev indset efterfølgende af *[RIESENFELD]*. I stedet for at anskue kurverne som resultatet af vægtning af kontrolpunkterne med en basisfunktion, tog Chaikin direkte udgangspunkt i geometrien for kontrolpolygonen. Chaikin's metode frembringer kurver identiske med uniforme kvadratiske B-splines ved, så at sige, at klippe hjørner af kontrolpolygonen. Givet en kontrolpolygon $\boldsymbol{b}^{\boldsymbol{k}} = b_0 \ldots b_n$, dannes en ny kontrolpolygon $\boldsymbol{b}^{\boldsymbol{k+1}} = q_0, r_{0,} q_{1,} r_{1,} \ldots, q_{n-1}, r_{n-1}$. For hvert segment b_i , b_{i+1} i kontrolpolygonen, findes to punkter q_i og r_i ved

$$q_i = \frac{3}{4} b_i + \frac{1}{4} b_{i+1}$$
$$r_i = \frac{1}{4} b_i + \frac{3}{4} b_{i+1}$$

formel 2.7

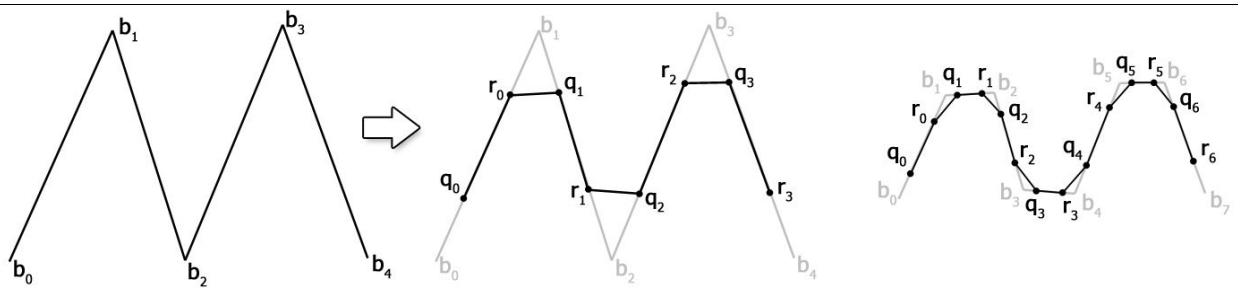


Illustration 2.7:
Chaikins algoritme til frembringelse af kubiske B-spline kurver ved iterativ subdivision.

Dette kan angives som en matrice, hvorved fås den *lokale* subdivision matrix,

$$\boldsymbol{S} = \frac{1}{4}\begin{bmatrix} 3 & 1 \\ 1 & 3 \end{bmatrix}$$

formel 2.8

Denne matrice anvendes uden ændringer i alle iterationer, og er uafhængig af antallet af punkter i kontrolpolygonen. Metoder for hvilke den lokale subdivision-matrice altid er den samme, kaldes ”stationære”. Det bemærkes, at summen af hver række er 1, hvorfor nye punkter dannes ved affine kombinationer af punkterne i kontrolpolygonen. Fordelen ved denne måde at angive kurver er hovedsageligt at den er enkel og let forståelig. Den kræver ikke evaluering af komplekse basisfunktioner, men baserer sig på simpel vægtning af punkter. Til gengæld er metoden vanskeligere er analysere, og giver ikke umiddelbart en parametrisering af den resulterende kurve, hvilket igen gør det vanskeligere at evaluere kurven i et vilkårligt punkt. En udvidelse af metoden findes, der giver mulighed for at danne rationelle kurver *[NASRI]*.

En anden måde at betragte subdivision-processen, er som en to-trins raket, hvor geometrien først underinddeles, og derefter udglattes. En god og intuitiv tilgang til subdivision kan ses i *[WARREN2]*. Her bygges en subdivision metode op, ved at betragte gennemsnit mellem nabo-vertices. En anden fordel ved generel subdivision-form i henhold til kurver er, at det let udvides til at omfatte generelle netværk. Herved kan arbitrære netværk underinddeles – og cykliske grafer konstrueres umiddelbart. *[WARREN1 s. 208]*

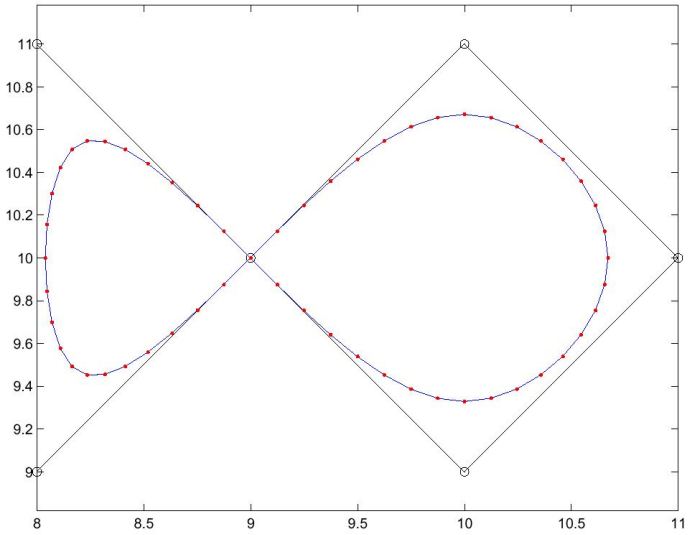


Illustration 2.8
Eksempel på subdivision af generelt netværk

2.3 Spline Flader

B-spline flader er en generalisering af kurverne. Man kan betragte en B-spline flade som en kurve af kurver – altså en kurve, hvor kontrol-punkterne er kurver. Dette realiseres ved, at hver række af kontrol-punkter har samme dimension som antallet af kontrol-punkter i søjle-kurverne. Der er tradition for, at parametrene i de to retninger benævnes *(u, v)*. Givet et kontrolnet med N rækker og M søjler, og kontrol-punkter $[b_0 \ldots b_i \ldots b_N] \times [b_0 \ldots b_j \ldots b_M]$, er fladen givet ved

$$\boldsymbol{r}(u,v) = \sum_{i=0}^{N} \sum_{j=0}^{M} \boldsymbol{b}_{i,j} \boldsymbol{B}_{i,n}(u) \boldsymbol{B}_{j,m}(v)$$

formel 2.9

Der er ingen restriktioner på graderne, *n* og *m,* af fladen i de to retninger, og de kan således godt være forskellige. B-spline-flader arver alle egenskaberne fra kurverne. Følgeligt har fladerne maksimal kontinuitet i hver retning, C^{n-1}, C^{m-1} , og det er muligt at påvirke formen ved at ændre på knudefølgen *[GRAVESEN]*.

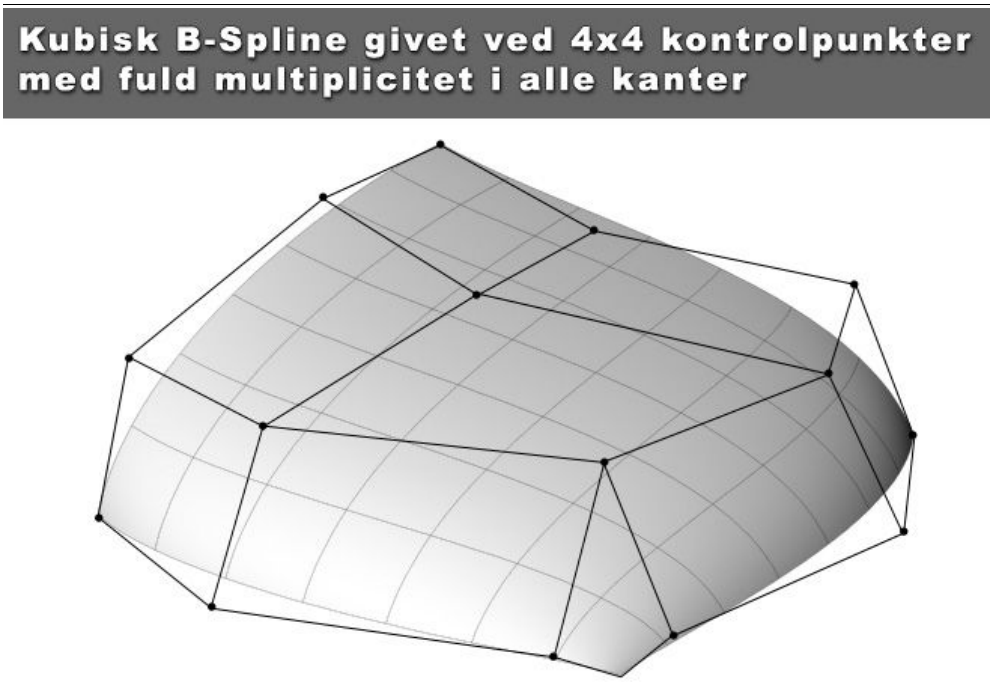


Illustration 2.9
Kubisk B-spline flade givet ved 4x4 kontrolpunkter, med fuld multiplicitet i alle kanter.

2.4 Hvad er problemet med B-spline flader?

På trods af, at B-splines er meget fleksible, er der en række afgørende problemer forbundet med brugen af dem. Vanskelighederne bunder i den grundlæggende begrænsning, at B-splines kun kan danne flader, der kan parametriseres til et sammenhængende, to-dimensionelt kort. Muligheden for at give de enkelte punkter vægt, gør det muligt at beskrive matematiske primitiver som f.eks. kugler eller keglesnit, ved hjælp af rationelle B-splines. Dette viser en af styrkerne ved B-splines.

Betragtes eksemplet med kuglen, vil parameteriseringen til et 2D-kort give singulariteter ved polerne. Yderligere er fladen ikke defineret i samlingen der kan ses på illustration 2.10 - punkter i denne samling kan dog findes ved at lave yderligere en parametricering af fladen. Det kan bevises, at alle flader, uanset matematisk topologi, kan omdannes til et to-dimensionelt kort ved at foretage en række snit i fladen. Ofte vil resultatet dog være en forvrænget parametricering der indeholder diskontinuiteter, som eksemplificeret på illustration 2.11, hvilket er uhensigtsmæssigt.

Man vil derfor ofte beskrive en flade som en mængde B-spline flader. Flader med kompleks matematisk topologi, skal således splittes op i flere underflader. For at sikre samme kontinuitet i samlingerne mellem flader, som på selve fladen, er det nødvendigt at tilføje en række randbetingelser. Dette er besværligt, og tilføjer høj kompleksitet, især hvis fladerne animeres. I praksis vil de lapper der udgør den samlede model, ofte have varierende detaljegrad. Tesselering af ulige nabo-patches kræver særlige hensyn, da der let opstår revner i samlingerne.

Som udgangspunkt kan der ikke tilføjes lokale detaljer til B-spline flader, da eneste mulighed er, at tilføje eller fjerne hele knuderækker eller søjler. Ser man på generelle polygon-modeller, ligger der ikke denne begrænsning.

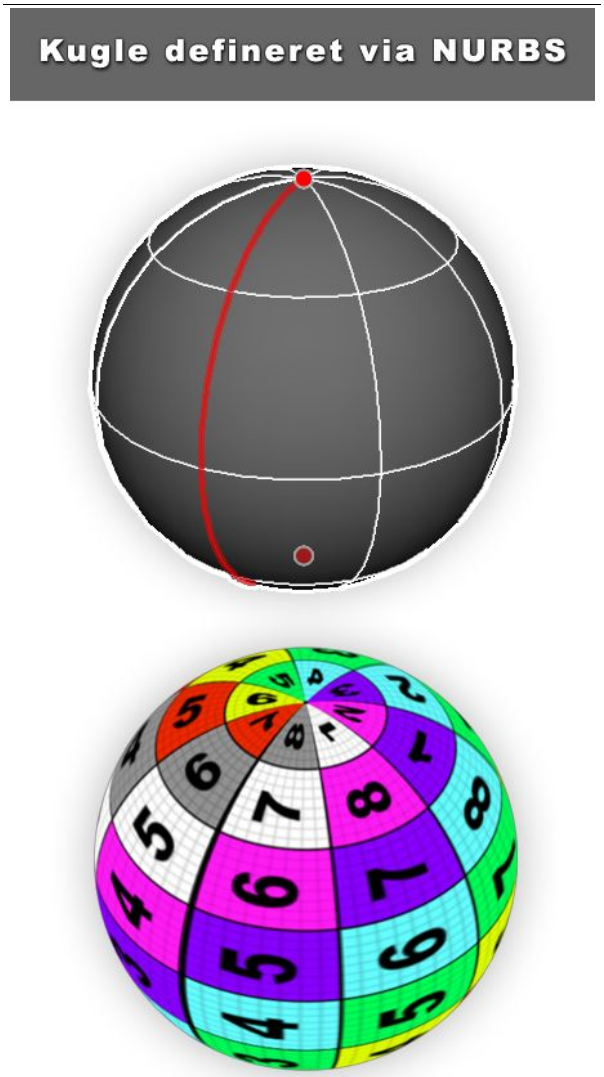


Illustration 2.10
Den røde linie markerer hvor der er en samling i fladen. De to punkter ved polerne er singulariteter i parametriceringen.

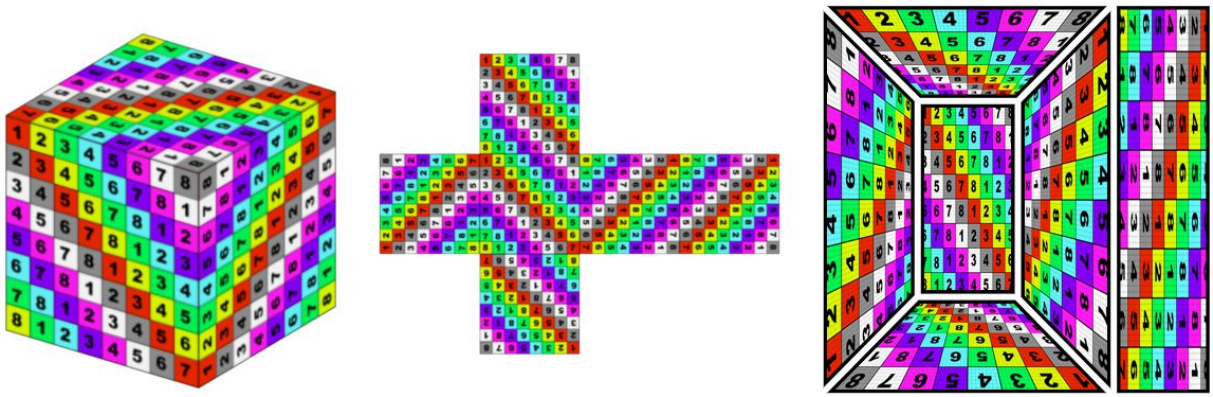


Illustration 2.11
Fra venstre mod højre: Den oprindelige kasse, kassen foldet ud til et 2d-kort

Der er arbejdet meget med løsninger på disse problemer, og mange smarte værktøjer og metoder er udviklet. I sidste ende, er det dog lappeløsninger på et grundlæggende problem ved B-splines.

Både B-spline kurver og flader finder udbredt anvendelse særligt inden for industrielt design hvor man har taget fleksibiliteten ved B-splines til sig, og udviklet arbejdsgange der er tæt knyttet hertil. Især kurverne er blevet industristandard, men de lider heller ikke under samme problemer som flader baseret på B-splines.

Revner efter tesselering grundet forskelligt detaljeniveau

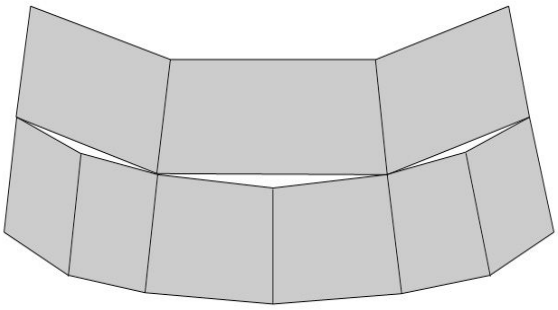


Illustration 2.12

Tilføjelse af detaljer til flader

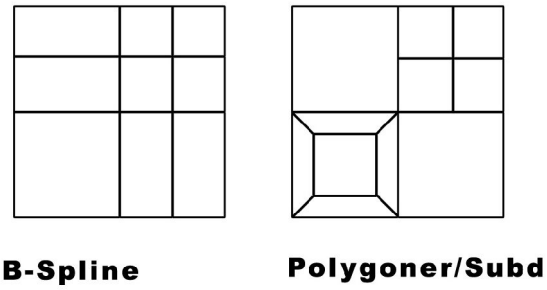


Illustration 2.13

2.5 Subdivision-flader

Subdivision surfaces er på linie med kurverne givet ved en kontrolpolygon – et kontrolmesh, og et sæt regler for, hvordan denne skal underinddeles. Forskellen til underinddeling af kurvenetværk er, at netværket nu betragtes som bestående af flader. Således tilføjes nye kanter der deler fladen, hvor

der i kurvetilfældet blot blev underinddelt eksisterende kanter[9]. Der findes mange subdivision metoder, der hver har deres styrker og svagheder. Den metode der har vundet størst indpas er udviklet af Catmull og Clark i 1978, og tager udgangspunkt i netværk bestående af firkanter. Det er disse flader der, med modifikationer, blev anvendt af Pixar i "Geri's Game" der I 1997 indbragte dem en Oscar.

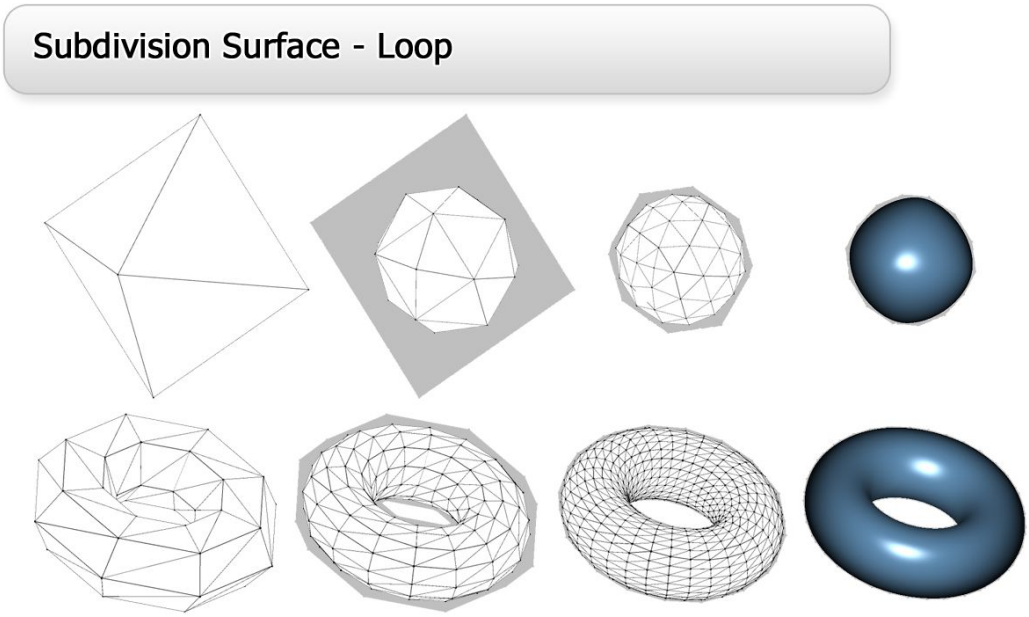


Illustration 2.14
Loop-underinddeling fra den grove model yderst til venstre, til grænsefladen yderst til højre. Det ses, at modellen skrumper for hver iteration, idet den anvendte metode er approksimerende. Det bemærkes, at en torus er den eneste lukkede, fuldstændigt regulære flade.

2.5.1 Loop subdivision

Charles Loop var den første der generaliserede subdivision til triangulære flader *[LOOP1]*. Metoden tager udgangspunkt i trekantede Beziér-patches *[BOEHM]*, hvilket betyder, at regulære områder af fladen evalueres til bløde flader identiske med disse. Udgangspunktet i trekantede net betyder, at metoden kan anvendes på alle typer geometri – da alle netværk af flader kan brydes ned i trekanter. Det betyder dog ikke, at metoden er universel. Når eksempelvis en firkant trianguleres, tilføjes en sammenhæng ('constraint') der ikke var der i forvejen. Det er derfor stadig mere korrekt, at gå ud fra den oprindelige geometri, i stedet for at triangulere fladen. Der er udviklet subdivision-metoder der frembringer B-spline flader for både trekanter og firkanter. Disse vil dog ikke blive behandlet her, men der henvises til *[STAM3, WARREN3]*.

9 En tilsvarende generalisering til underinddeling af volumenmodeller kan foretages, og præsenteres blandt andet i *[MAPS, PASCUCCI]*

Loop subdivision sker i to trin: Først underinddeles alle kanter i modellen[10] som vist i illustration 2.16. Positionen af de nye kantpunkter, $\boldsymbol{b}_{ei}^{k+1}$, gives ved en vægtning i forhold til flade-naboerne i $\boldsymbol{b}^k$, se illustration 2.17. Hernæst udglattes alle punkter i $\boldsymbol{b}^k$, altså de gamle punkter, til $\boldsymbol{b}^{k+1}$. Dette sker ved, at vægte dem i forhold til de oprindelige naboer i $\boldsymbol{b}^k$ som vist til højre på illustration 2.17. Denne underinddeling af flader giver, i k'te iteration, 4^k gange så mange trekanter som der er i basis-polygonen.

Underindeling af polygoner

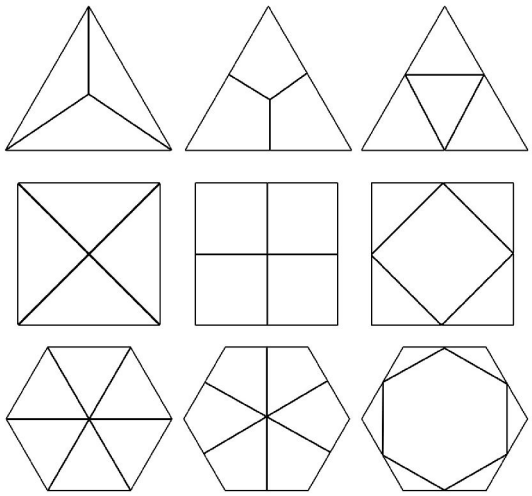


Illustration 2.15
Venstre: Ingen kant-inddeling, resulterer hurtigt i meget aflange polygoner. Midt: Anvendes i f.eks. Catmull/Clark. Højre: Loop og Modified Butterfly.

Givet det lukkede kontrolmesh $\boldsymbol{b}^k$, foretages en Loop subdivision-iteration der fører over i $\boldsymbol{b}^{k+1}$, ved

$$\boldsymbol{b}_i^{k+1} = (1 - n\beta(n))\boldsymbol{b}^k + \beta(n)\sum_{i=0}^{n-1} adj_i(\boldsymbol{b}^k)$$

$$\boldsymbol{b}_{ei}^{k+1} = \left(\frac{1}{8}\right)(3\boldsymbol{b}_0 + 3\boldsymbol{b}_1 + \boldsymbol{b}_2 + \boldsymbol{b}_3)$$

$$-\text{hvor } n \text{ er valensen, og } adj_i(\boldsymbol{b}^k) \text{ er } i\text{'te nabo til punktet } \boldsymbol{b}^k.$$

formel 2.10

Loop-subdivision

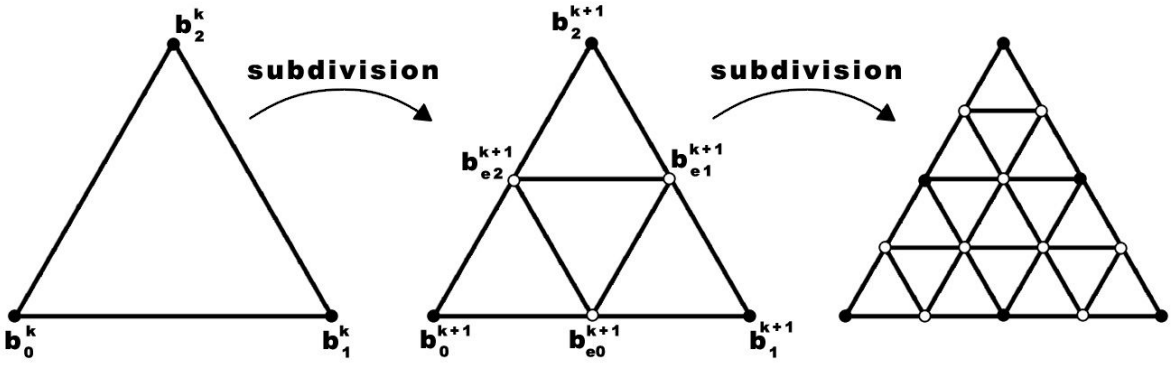


Illustration 2.16 - loop iteration. Hver trekant bliver til fire nye, ved at indsætte et punkt på hver kant.

10 På illustration 2.15 vises Loop-subdivision på ikke-trekanter. Denne type underinddeling betegnes "corner-cutting". Loop-reglerne giver "pæne" flader for denne type underinddeling, men der er ikke foretaget formelle beregninger af de resulterende kurvers kontinuitet.

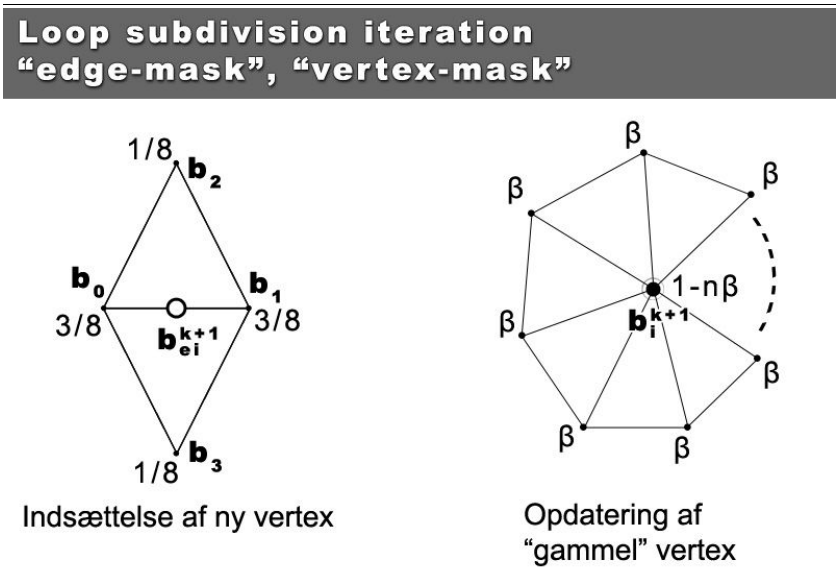


Illustration 2.17
Loop regler for at føre en polygon fra detaljegrad k til k+1. n er valensen af et givet punkt.

Værdien af konstanten *β(n)* kan vælges forskelligt. Loop angiver følgende vægte

$$\beta(n)=\frac{1}{n}\left(\frac{5}{8}-\frac{3+2\cos\left(2\pi/n\right)^2}{64}\right)$$

formel 2.11

Anvendes denne vægtning, dannes flader der har C^2 kontinuitet i punkter væk fra ekstraordinære punkter, og C^1 alle andre steder. Et ekstraordinært punkt, er et punkt der har en anden valens end det ”normale”. I et firkantet net har et regulært punkt således valens 4, mens det i et triangulært net, som anvendt ved loop-subdivision, er 6. Andre værdier for *β(n)* kan anvendes, hvilket giver flader med andre karakteristika. I *[WARREN1]* angives vægte der danner flader med samme egenskaber, men er hurtigere at evaluere idet de undgår trigonometriske funktioner, og i *[SCHRÖDER1]* gives vægte der giver pænere flader for punkter med lav valens.

Det bemærkes, at reglerne der anvendes ved en subdivision-iteration, er stationære – de afhænger altså ikke af f.eks. arealet af de omkringliggende flader, eller hvilken iteration der beregnes. Man kunne formodentlig få pænere flader ved at anvende dynamiske regler, men så bliver det meget svært at sige noget om grænsefladerne rent matematisk. Dette er i forvejen vanskeligt, og kun ved udførlig analyse er det muligt at sige noget om en flades opførsel nær ekstraordinære punkter. Det bør her bemærkes, at ønsker man at bevare den højest mulige kontinuitet, gælder samme krav til subdivision flader som til B-splines. Forskellen er, at man har mulighed for at gå på kompromis med kravet om maksimal kontinuitet over hele fladen.

En anden udbredt metode til subdivision blev som nævnt udviklet af Catmull og Clark i 1978. Catmull-clark flader har den fordel, at modellen alene består af firkanter efter første iteration, se midt illustration 2.15. Loop subdivision kan også anvendes på andre typer netværk end trekanter, men da bevares den oprindelige struktur i de yderligere iterationer. En kontrolpolygon der indeholder en sekskant, vil således også indeholde en sekskant efter en Loop iteration, mens en Catmull-Clark iteration vil omdanne den til seks firkanter, se nederst illustration 2.15. Til gengæld

indsætter Catmull-Clark flader ekstraordinære nye punkter i første iteration, mens Loops metode kun genererer regulære punkter. Generelt bemærkes det, at trekant- og firkant-underinddeling ikke ændrer på valensen af de oprindelige punkter.

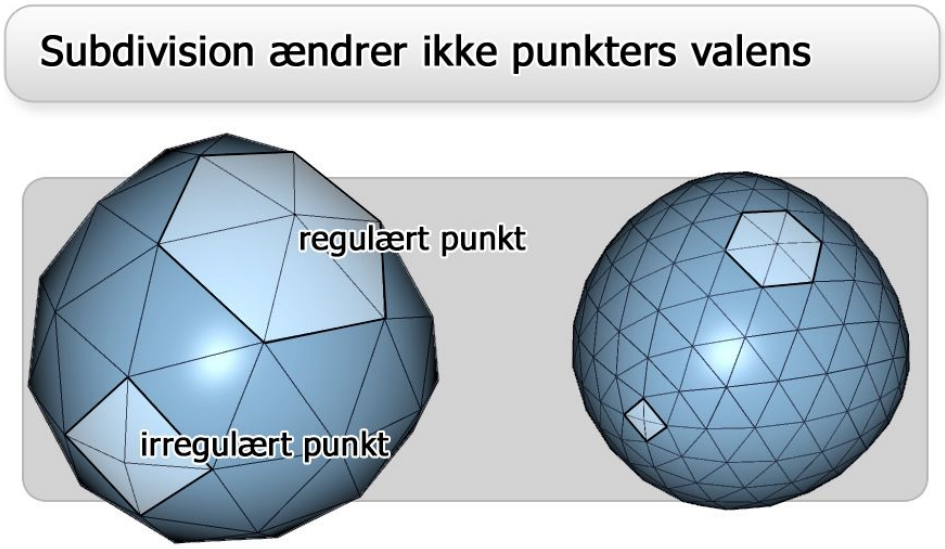


Illustration 2.18
Bemærk at området der påvirkes af et ekstraordinært skrumper for hver iteration

2.5.2 Støtten for en subdivision-iteration

Den del af en model der skal være til stede for at et givet område af modellen kan underinddeles, kaldes områdets "støtte" ("support"). For Loop-subdivision, er støtten for udglatning af en enkelt vertex, b_i vist til venstre på illustration 2.19. Under udglatning positioneres alle "gamle" punkter i forhold til deres egen og deres naboers positioner. Det er væsentligt at denne vægtning sker i forhold til naboerne inden trekanten underinddeles. Det er således nødvendigt at underinddele støtten delvist, for at finde de punkter der skal anvendes til udglatning, efter første iteration. Idet positionen af nye punkter på kanten kun afhænger af punkterne i deres fladenaboer, formel 2.10 s. 30, kan de ligeledes findes ud fra 1-ordens naboerne til den udglattede vertex. Det er således muligt at finde positionen af center-vertexen, b_i, for en vilkårlig subdivison-iteration, alene ud fra støtten til denne vertex.

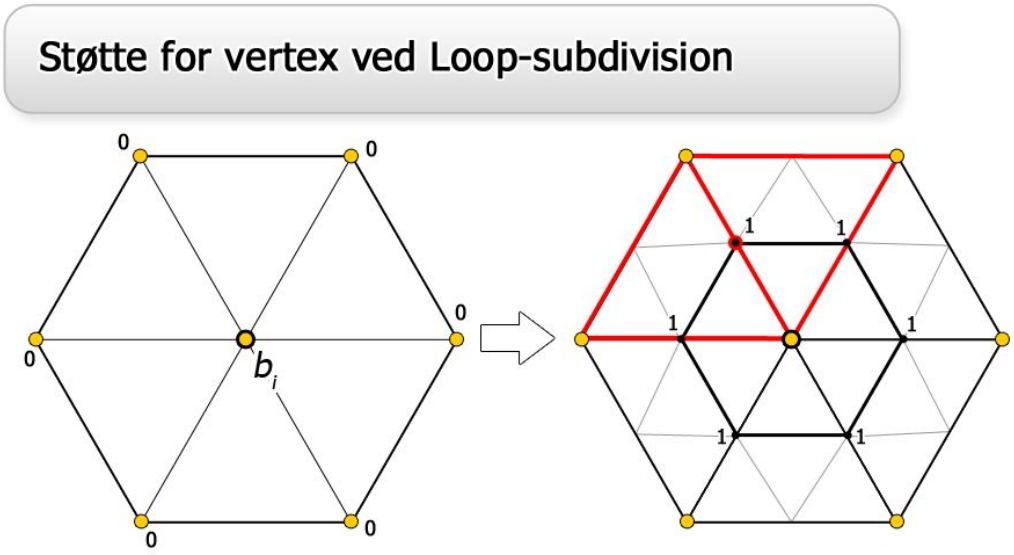


Illustration 2.19:
Støtte for udglatning af en vertex. Til højre ses punkterne der er nødvendige for at finde den midterste vertex's position i anden iteration. Det ses, at de indre kant-punkter for støtten kan findes ud fra centervertexens støtte. Med rødt er markeret de nødvendige punkter ved beregning af det nye kantpunkt.

Med støtten for en subdivision-metode, menes det naboområde af en regulær flade, der kræves for at beregne underinddelingen af fladen. Sædvanligvis ses på en enkelt trekant for Loop-subdivision (illustration 3.1 side 43) og en firkant for Catmull/Clark. Dette naboområde findes ved at underinddele en enkelt trekant, og observere hvilke punkter der anvendes ved såvel tilføjelse af nye punkter, som udglatning af gamle. For Loop og Catmull/Clark-subdivision består støtten således af alle trekanter der er forbundet via mindst en kant, til en vertex i det underinddelte område. Støtten udgør generelt en begrænset del af kontrolmodellen, og er afhængig af den subdivision-metode der anvendes.

2.5.3 Grænsenormaler og positioner

Loops metode er approksimerende, hvilket betyder at kontrolpolygonen iterativt tilnærmer sig grænsefladen, og at denne ikke generelt går igennem kontrolmodellen. Dette betyder også, at meshet ”skrumper” for hver iteration der køres. Under konstruktionen af den grove model, er det derfor nødvendigt at tage højde for, at modellen skrumper. Det værende sig hvad enten fladen skal fittes til en række måledata, eller blot modelleres manuelt til at ligne et objekt.

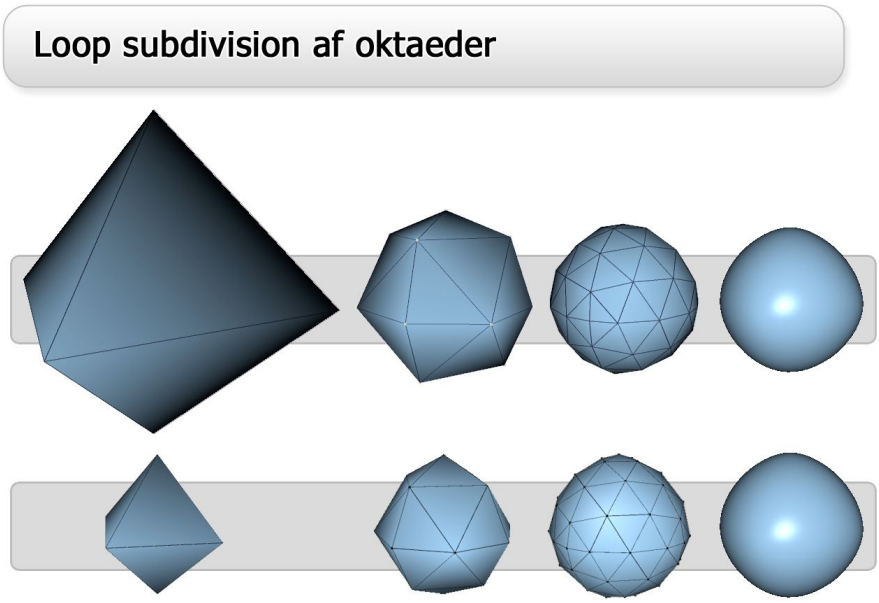


Illustration 2.20
Loop er en approksimerende metode. Nederst ses hver iteration, med alle punkter skubbet til grænsefladen.

Det er muligt at beregne den position en vertex ville have, efter et uendeligt antal subdivisions. Denne position kaldes for vertexens "grænseposition". Flyttes en vertex til sin grænseposition, kaldes dette at "skubbe" ("push") til grænsepositionen. For en polygonmodel er grænsepositionen i sagens natur den samme efter en subdivision-iteration, idet grænsefladen ikke ændrer sig. Det bemærkes, at denne beregning således kan foretages på modellen i en vilkårlig iteration – også på den grove model. Det er ikke meningsfyldt at fortsætte subdivisionen på en skubbet model, da kontrol-punkterne, og således grænsefladen, er ændret. Ønsker man at visualisere grænsepositionen, men gøre det muligt at foretage yderligere subdivision, er det således nødvendigt at bibeholde den oprindelige, ikke-skubbede, model.

En overordnet gennemgang af udledningen for beregning grænsepositionen vises i afsnit 2.6. En lettere anvendelig formulering af resultatet anføres her. Beregningen af grænsepunkter kan foretages fuldstændig som en almindelig udglatning, hvor blot vægten $\beta(n)$ udskiftes med en anden vægt, $\gamma(n)$. Vægten $\gamma(n)$ der skal anvendes for at finde grænsepositionen er givet ved

$$\gamma(n) = \frac{1}{n + \frac{3}{8\beta(n)}}$$

Formel 2.12

På tilsvarende vis kan den eksakte normal til grænsefladen beregnes i en vilkårlig vertex. Normalen til en flade anvendes til lysberegninger, og er derfor afgørende for fladens udseende. Anvendes ikke særlige regler for skarpe kanter, ("creases"), eller lignende, kan gennemsnitlige eller vinkelvægtede normaler anvendes uden visuel forskel. For et punkt $\boldsymbol{p}$ uden creases, på en uniformt underinddelt loop subdivision flade, er tangent-vektorerne givet ved:

$$\boldsymbol{t}_u = \sum_{i=0}^{n-1} \frac{\cos(2\pi i)}{n} adj_i(\boldsymbol{p}), \quad \boldsymbol{t}_v = \sum_{i=0}^{n-1} \frac{\sin(2\pi i)}{n} adj_i(\boldsymbol{p})$$

Formel 2.13

- normalen til fladen kan således findes som $\boldsymbol{t}_u \times \boldsymbol{t}_v$. Det skal kort nævnes, at indexet *i* til nabo *i* indgår i beregningen. Der er således, som i de almindelige regler for udglatning, en implicit antagelse om, at punkterne ligger jævnt fordelt omkring $\boldsymbol{p}$.

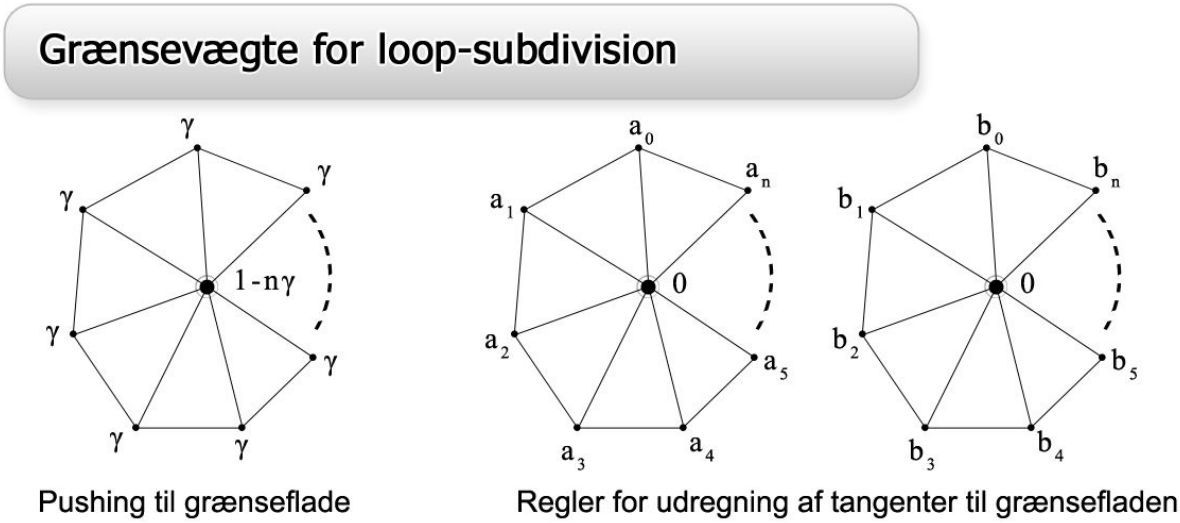


Illustration 2.21

Konstanterne ved tangenter til grænsefladen er givet ved: $a_i = \frac{cos2\pi i}{n}, b_i = \frac{\sin 2\pi i}{n}$

Notationen fra den øvrige litteratur er fulgt her, og tangenterne betegnes derfor t_u og t_v . Dette er dog mildt misvisende, idet det indikerer en entydig parameterisering af fladen, hvilket ikke er tilfældet. De to tangenter afhænger af valget af traverseringen af naboerne til det pågældende punkt. Det skal således bemærkes, at det er vigtigt at foretage traverseringen af nabopunkterne ens under beregningen af de to tangenter. Krydsproduktet af to tangenter vil altid give en normal til fladen, men der er risiko for at normalen inverteres, hvis den indbyrdes orientering af tangenterne vendes.

Grænsetangenterne vil være lige lange, men har generelt ikke enhedslængde. Ønskes en normal af enhedslængde, er det således mest effektivt at normalisere normalen frem for de to tangenter.

2.6 Subdivision matricen – beregning af grænsepunkter

I dette afsnit vil udledningen af vægte for beregning af grænsepositioner blive gennemgået overordnet. Udledningen baserer sig på egenværdi-analyse af subdivion-matricen, og er som sådan ganske kompliceret. En fuld analyse af dette problem ligger udenfor rammerne for dette projekt, og der henvises til udførlige beskrivelser i eksempelvis *[SCHWEITZER, STOLLNITZ, WARREN1]*. Der vil i det følgende ligeledes blive taget udgangspunkt i disse kilder.

Som vist i afsnit 2.5.2, kan en vilkårlig subdivision-iteration af et punkt og dets naboer i den iteration, beregnes ud fra støtten til punktet. Idet denne beregning er givet ved en lineær vægtning af

punkterne i støtten, kan den opstilles på matrixform. En væsentlig egenskab her er, at reglerne for vægtningen er stationære, og således ikke varierer fra iteration til iteration. For et punkt b_i^0 med valens *n*, kan punktet og dets nye naboer efter en subdivision-iteration, beregnes ved multiplikation med en matrix $\boldsymbol{S_n}$, se illustration 2.19 s.33. Denne matrix betegnes, ligesom for Chaikin's metode i afsnit 2.2.3, den lokale subdivisionmatrice, og har størrelsen $(n+1)\times(n+1)$. For en enkelt subdivision iteration har vi hermed følgende formel for underinddeling af en vertex og dens støtte,

$$\begin{pmatrix} b_i^1 \\ adj_0(b_i^1) \\ adj_1(b_i^1) \\ \vdots \\ adj_{n-1}(b_i^1) \end{pmatrix} = \boldsymbol{S_n} \begin{pmatrix} b_i^0 \\ adj_0(b_i^0) \\ adj_1(b_i^0) \\ \vdots \\ adj_{n-1}(b_i^0) \end{pmatrix}$$

formel 2.14

Benævnes en vektor[11] indeholdende støtten til punktet b_i^k ved en given subdivision-iteration *k* med $\boldsymbol{v_i^k}$, kan formel 2.14 skrives som

$$\begin{aligned} \boldsymbol{v^1} &= \boldsymbol{S_n v^0} \\ \boldsymbol{v^2} &= \boldsymbol{S_n v^1} = \boldsymbol{S_n}(\boldsymbol{S_n v^0}) = \boldsymbol{S_n^2 v^0} \\ &\vdots \\ \boldsymbol{v^{k+1}} &= \boldsymbol{S_n v^k} = \boldsymbol{S_n^k v^0} \end{aligned}$$

formel 2.15

Det bemærkes, at dette er analogt til situationen for B-splines på matrixform, givet i afsnit 2.2.

En måde at analysere konvergensen af subdivisionmatricen når den løftes til en stigende potens, er ved at foretage egenværdi-analyse af matricen. Herved kan findes egenværdierne $\lambda_0, \lambda_1, \ldots, \lambda_n$, ordnet i ikke-stigende rækkefølge, og de tilhørende højre egenvektorer $\boldsymbol{R_0}, \boldsymbol{R_1}, \ldots, \boldsymbol{R_n}$.

Vektoren af støttepunkter, $\boldsymbol{v_i}$, kan opløses efter den basis der udspændes af egenvektorerne. Herved findes følgende linearkombination af egenvektorerne for $\boldsymbol{S}$,

$$\begin{aligned} \boldsymbol{v^{k+1}} &= \boldsymbol{S_n^k v^0} \Rightarrow \\ \boldsymbol{v^{k+1}} &= \boldsymbol{S_n^k}(c_0 \boldsymbol{R_0} + c_1 \boldsymbol{R_1} + \ldots + c_n \boldsymbol{R_n}) \Leftrightarrow \\ \boldsymbol{v^{k+1}} &= c_0(\boldsymbol{S_n^k})\boldsymbol{R_0} + c_1(\boldsymbol{S_n^k})\boldsymbol{R_1} + \ldots + c_n(\boldsymbol{S_n^k})\boldsymbol{R_n} \end{aligned}$$

formel 2.16

Idet $\boldsymbol{S}\,\boldsymbol{R_i} = \lambda_i \boldsymbol{R_i}$ per definition for egenvektorer, fås

$$\boldsymbol{v^{k+1}} = c_0 \lambda_0^k \boldsymbol{R_0} + c_1 \lambda_1^k \boldsymbol{R_1} + \ldots + c_n \lambda_n^k \boldsymbol{R_n}$$

formel 2.17

11 Situationen vi betragter er for "1D-punkter". Idet samme vægt anvendes for interpolation af alle koordinater i positionen, betragter vi her positionen som en enkelt værdi.

Et krav for, at en subdivision-metode konvergerer er, at den tilhørende lokale subdivision-matrix har en enkelt største egenværdi, λ_0 , og at denne er 1 *[SCHWEITZER s.27]*. Der gælder således følgende[12],

$$\lambda_0 = 1 > \lambda_1 \geqslant \lambda_2 > \lambda_3 \ldots > \lambda_n$$

formel 2.18

Idet alle andre egenværdier end λ_0 er mindre end 1, bliver de nul når de opløftes til en "uendeligt" høj potens. I grænsetilfældet, er således kun første led i formel 2.17tilbage,

$$\begin{aligned} \boldsymbol{v}^{\infty} &= \lim_{k \to \infty} \left(c_0 \lambda_0^k \boldsymbol{R_0} + c_1 \lambda_1^k \boldsymbol{R_1} + \ldots c_n \lambda_n^k \boldsymbol{R_n} \right) \Rightarrow \\ \boldsymbol{v}^{\infty} &= c_0 \boldsymbol{R_0} \end{aligned}$$

formel 2.19

Idet det erindres, at det betragtede punkt b_i er det første element i $\boldsymbol{v}$, kan grænsepositionen for b_i findes som c_0 ganget med det første element i $\boldsymbol{R_0}$. Endnu et krav der skal være opfyldt for, at en subdivision-metode konvergerer er *[SCHWEITZER s.25]*, at den højre egenvektor

$$\boldsymbol{R_0} = \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix}$$

formel 2.20

- hvorved formel 2.19 reducerer til

$$\begin{aligned} \boldsymbol{v}^{\infty} &= \begin{pmatrix} c_0 \\ \vdots \\ c_0 \end{pmatrix} \Rightarrow \\ v_0^{\infty} &= c_0 \Leftrightarrow \\ b_i^{\infty} &= c_0 \end{aligned}$$

formel 2.21

Det kan vises, at c_0 kan udtrykkes ved $\boldsymbol{v}$ og den *venstre* egenvektor for subdivison-matricen *[SCHWEITZER, STOLLNITZ]*. Herved kan endelig grænsepositionen findes, udtrykt ved den oprindelige støtte

$$\begin{aligned} b_i^{\infty} &= \frac{w v_0 + v_1 + \ldots + v_n}{w+n} = \frac{w b_i^0 + adj_0(b_i^0) + \ldots + adj_n(b_i^0)}{w+n} \Leftrightarrow \\ b_i^{\infty} &= \frac{w}{w+n} b_i^0 + \frac{1}{w+n} \sum_{i=0}^{n} adj_i(b_i^0) \end{aligned}$$

formel 2.22

12 Det er ikke en trykfejl at $\lambda_1 \geqslant \lambda_2$. Argumentet for at dette skal gælde, er givet i *[SCHWEITZER]*.

- hvor $w=\frac{3}{8\beta(n)}$, og $\beta(n)$ er vægten anvendt ved udglatning, givet i afsnit 2.5.1. Ved omskrivning af formel 2.22, kan udledes formel 2.12, side 34. Grænsepositionen for en vertex findes altså ved at opløse en vektor til positionen efter egenvektorerne i subdivision-matricen, og finde komponenten med hensyn til den første egenvektor. Vægtene for tangenter findes ved at se på de resterende egenværdier for subdivision-matricen.

2.7 Skarpe kanter, spidse punkter og åbne flader

Det er som udgangspunkt ikke muligt at ændre på kontinuiteten af subdivision flader – at lave skarpe kanter, spidse punkter eller lignende. Man kan efterligne effekten, ved indledende, at lave multiplicitet i kanterne på kontrol-meshet, se illustration 2.22. Dette medfører dog beregning af et højt antal trekanter der i sidste ende ikke er synlige. Ikke desto mindre, er det den metode der i praksis er oftest benyttet, da det giver mulighed for, ved direkte manipulation af kontrolmodellen, at ændre skarpheden af en kant. Et tilsvarende trick til at få spidse punkter er, at lave en ring af kanter omkring den pågældende vertex. En anden er, at undlade at udglatte de pågældende punkter. Disse punkter bevarer således deres oprindelige position, og bliver dermed "spidse", da resten af fladen udglattes på sædvanlig vis.

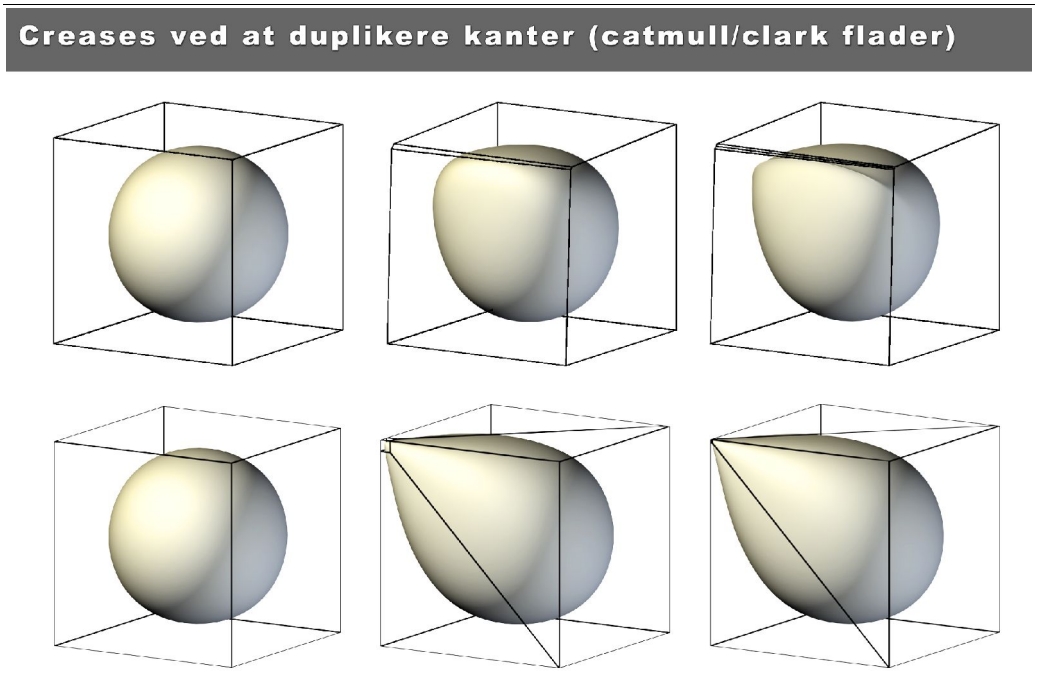


Illustration 2.22
Øverst duplikeres en kant 0, 1 og 2 gange for at opnå en skarp kant.
Nederst tilføjes kanter omkring en vertex for at gøre den spids.

En mere generel fremgangsmåde, er at udvide de grundlæggende subdivision regler, med en række specielle regler for kanter der ønskes skarpe. Denne metode er ofte anvendt, og der er udviklet regler for de fleste subdivision metoder.

I *[HOPPE1]* gives et eksempel på anvendelse af subdivision flader til fitting, der tager udgangspunkt i analyse af en polygon-model. Loops metode udvides i denne sammenhæng med en mulighed for på vertex-niveau, at angive skarphed, se illustration 2.23. Desuden præsenteres regler for grænse-positioner og -tangenter. Åbne flader, altså flader med huller i, håndteres ved at markere alle vertices omkring hullet som ”skarpe”. Det bemærkes, at vægtene for skarpe kanter giver en lineær interpolation mellem de to endepunkter.

Et generaliseret sæt regler for Loop og Catmull/Clark, der yderligere tager højde for fladens normaler i skarpe kanter, gives i *[BIERMANN]*. Denne metode tilføjer mulighed for, at styre subdivision-fladers form ved at ændre på normalerne, der sædvanligvis ellers ikke indgår i beregningen.

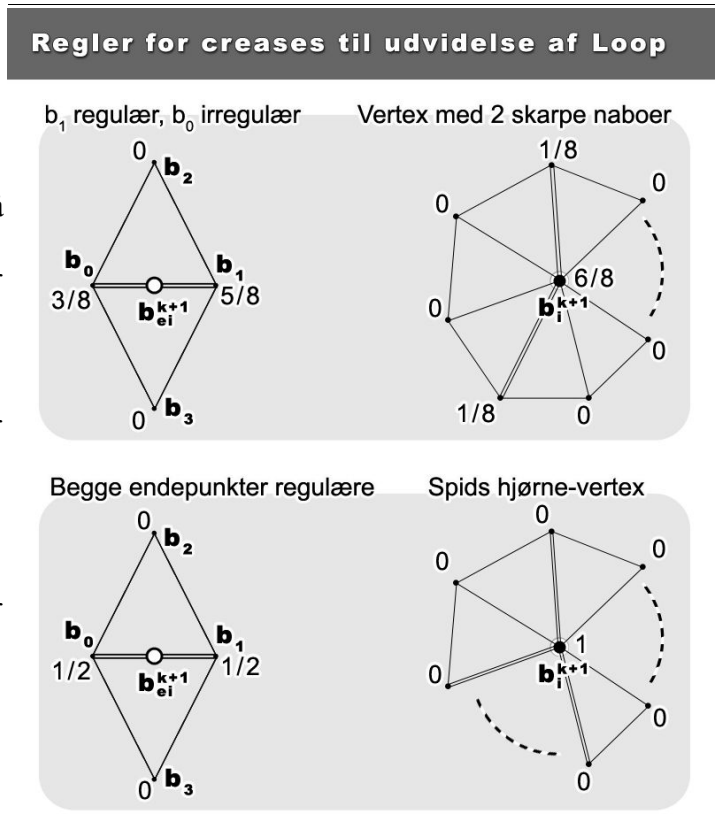


Illustration 2.23 (dobbelte kanter er skarpe)
Det bemærkes, at regulær og irregulær betegner antallet af ikke-skarpe naboer. En vertex med 7 naboer, hvoraf den ene er markeret som skarp, er således regulær. En vertex betegnes her ”spids” når den har flere end to skarpe naboer.

Kontinuiteten af fladerne ændres i områder omkring skarpe kanter – en skarp kant er i sagens natur kun C^0. Disse udvidede regler for Loop-flader, giver således flader der er C^2 i områder væk fra ekstraordinære punkter samt skarpe kanter og spidse punkter.

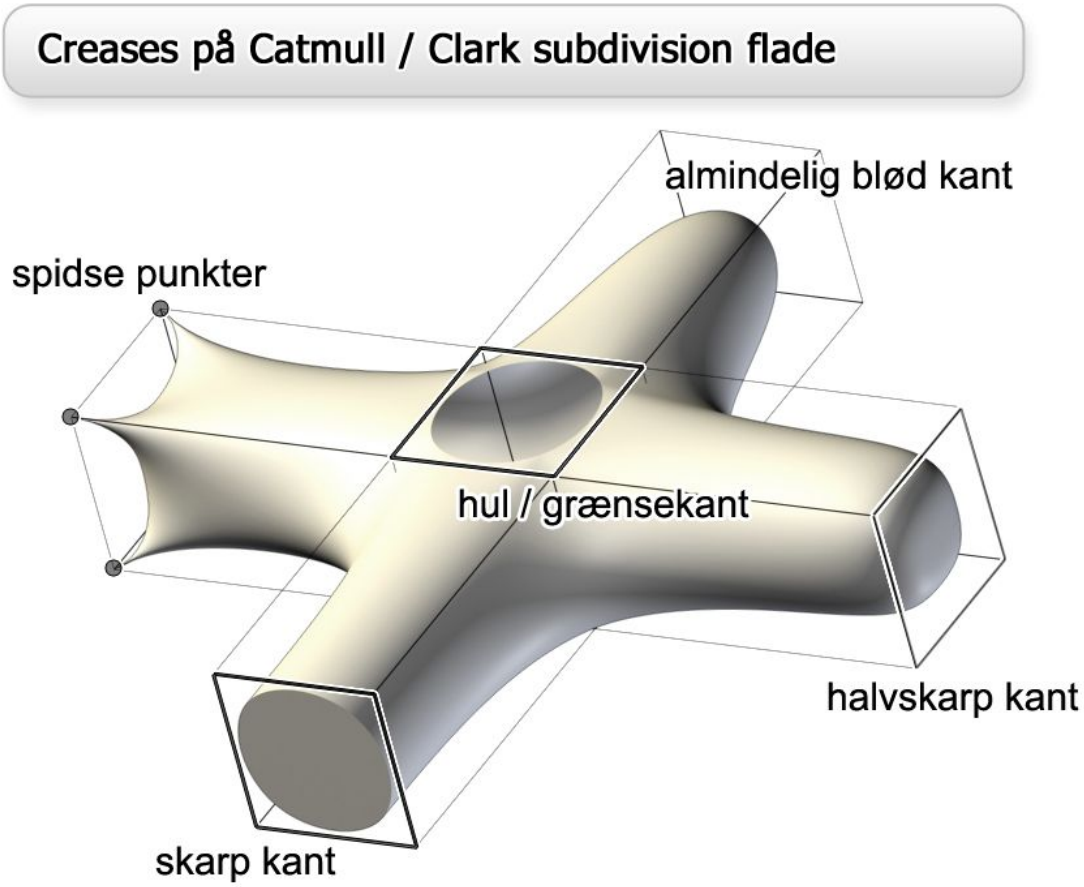


Illustration 2.24
Catmull-clark flader, med forskellige typer creases

Ovenstående regler giver mulighed for at markere kanter og vertices som skarpe. De giver dog ingen mulighed for at specificere graden af skarphed. Pixar foreslår, og har patenteret, en måde at lave delvist skarpe kanter, hvor man underinddeler et par iterationer med brug af regler for skarpe kanter, og anvender almindelige regler, eller en interpolation mellem de to sæt regler, i efterfølgende iterationer. *[SIGGRAPH]*

En stærkere, men mere kompleks fremgangsmåde beskrives i Sabin's NURSS[13]-metode *[SABIN2]*. Denne generaliserer idéen fra B-splines om anvendelsen af en knudefølge til at håndtere arbitrære rektangulære net. Metoden vises for (C^1) DooSabin- og (C^2) Catmull/Clark subdivision-flader. En udvidelse af denne metode kan findes i *[SEDERBERG1]*. Udvidelsen består af to dele, T-Splines og T-NURCCS (Non-Uniform Rational Catmull Clark Surfaces). T-spline flader er B-splines, hvor det er muligt at danne T-samlinger i fladen. Fladerne skal altså være rektangulære, men det er muligt at foretage lokal detaljering af fladen. T-NURCCS er en generalisering af T-splines til flader med arbitrær topologi, med udgangspunkt i Catmull-Clark underinddeling. Fordelen ved disse flader er, at det er muligt at lave knude-indsættelse, helt analogt til B-splines. Det betyder at det bliver muligt at ændre kontrolmodellen lokalt, uden at grænsefladen ændrer sig. Tidligere har været udviklet metoder med samme formål *[DISNEY, HALSTEAD, LAVOUE]*, men de har baseret sig på fitting af fladen efter "knudeindsættelsen".

13 NURSS – Non Uniform Recursive Subdivision Surfaces

2.8 Parametrisering af subdivision flader

Som nævnt, er en af de primære styrker ved subdivision surfaces, at de ikke kræver at der kan findes en parametrisering af fladerne. Desværre betyder det også, at anvendelser der kræver en parametricering ikke umiddelbart kan benyttes. Idet der er udviklet en stor mængde værktøjer der tager udgangspunkt i parametriske flader, vil en sådan parametrisering lette analysen af, og arbejdet med, subdivision surfaces.

Ud fra helt ækvivalente metoder, udledes i *[STAM1, STAM2]* en sådan parametrisering for henholdsvist Catmull/Clark og Loop subdivision flader. Metoderne tager udgangspunkt i analyse af subdivision matricen, og egenbasis-funktioner for denne udledes. Den direkte evaluering af subdivision fladerne foregår ud fra disse egenbasis-funktioner. For catmull-clark flader er disse basisfunktioner bikubiske polynomier – altså svarende til kubiske B-spline flader. Teorien for disse flader kan således anvendes direkte, hvilket medfører, at de afledte til fladerne tillige kan evalueres eksakt. Egenbasis funktionerne er givet alene ud fra subdivision-reglerne, og kan således forudberegnes. I *[ZORIN2]* udvides metoden til at håndtere hårde kanter og åbne flader. Udgangspunktet her er Loop-subdivision, og parameteriseringen af fladen dannes ved de barycentriske koordinater for hver trekant, samt en diskret fjerde koordinat givet ved et trekant-indeks.

Eksistensen af en parametrisering af fladerne, åbner op for en række anvendelser. En af de vigtigere er, at raytracing bliver umiddelbart tilgængeligt, hvor det tidligere har været nødvendigt enten at triangulere til subpixel-niveau eller approksimere fladen på anden vis *[KOBBELT2]*. Stam selv anvender dem til, at evaluere flader eksakt i områder der indeholder ekstraordinære punkter. En anden anvendelse kunne være, at foretage en vilkårlig tessellering af kontrolpolygonen, og beregne punkterne via denne metode.

3 Visualisering af subdivision surfaces i realtid

Der vil i dette afsnit blive beskrevet forskellige metoder til hurtig beregning af subdivision surfaces. Der vil ikke blive givet en gennemgang af alle metoder til visualisering i realtid. I stedet vil nogle af de væsentligste tendenser blive trukket frem, for herved at give en forståelse for de fremgangsmåder der kan anvendes. Den metode der er udviklet i dette projekt bygger på tabelbaseret subdivision, hvorfor en uddybende beskrivelse af dette emne vil blive givet.

3.1 Adaptiv subdivision

Det primære problem med visualisering af subdivision surfaces er det store antal trekanter der resulterer, samt de relativt tunge beregninger der skal til for at finde punkterne på disse trekanter. En åbenlys tilgang til problemet er, at undlade at beregne, og tegne, de dele af en flade der ikke gør nogen visuel forskel. Fladestykker der ikke kan ses, eller ikke krummer, er eksempler på områder der ikke er nødvendige at underinddele for at give et resultat der er visuelt identisk med grænsefladen.

På illustration 3.1 ses det, at kun en begrænset del af den subdividede støtte anvendes ved beregningen af de indre trekanter. Yderligere er det kun nødvendigt at underinddele til et niveau lavere end selve trekanten. Dette giver mulighed for at tilføje detaljer til modellen lokalt, og uden at splitte den op, så længe der holdes styr på hvilket niveau alle trekanter er på. Hvis subdivision af en trekant skal kunne foretages, må der ikke være større forskel end et enkelt subdivision-niveau på trekanten og dens naboer. Dette begrænser metodens fleksibilitet, men fremgangsmåden er enkel og stiller få krav til den underliggende datastruktur *[PULLI, KOHLER]*.

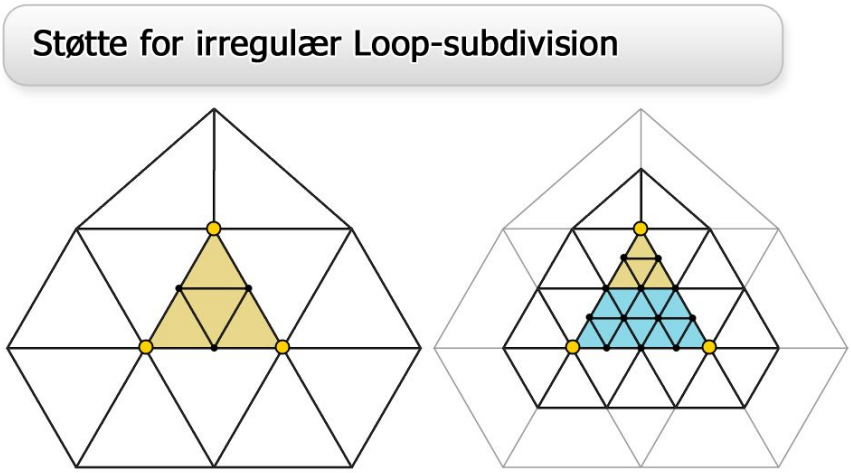


Illustration 3.1:
De blå indre trekanter er regulære, de gule er irregulære. Efter hver iteration kan 3/4 af de nye trekanter evalueres regulært. Første underinddeling er "gratis". For at foretage yderligere underinddelinger, er det nødvendigt delvist at underinddele støtten.

I *[HAVEMANN, HAVEMANN2]* vises det hvordan den begrænsede støtte kan anvendes til at splitte modellen op, og evaluere hvert patch separat. Det giver en algoritme der er mindre ressourcekrævende end almindelig rekursiv underinddeling. Det vises også hvordan det kan undgås

at foretage samme beregning flere gange, selvom der er overlap imellem de støtter modellen splittes op i.

En problemstilling der skal løses når der foretages non-uniform underinddeling, er triangulering af naboflader med ulige detaljegrad. I lighed med ulige NURBS-naboer, se afsnit 2.3, er det, for at undgå huller eller revner, nødvendigt at håndtere ulige naboer[14]. Den mest almindelige måde at gøre dette på, er at foretage en triviel triangulering af de lavinddelte trekanter.

Eksempler på triangulering af ulige naboflader

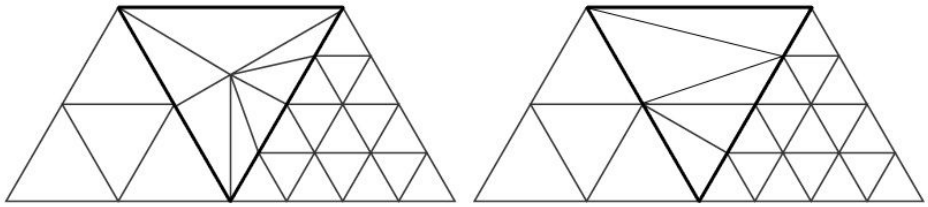


Illustration 3.2
To eksempler på triangualisering. Til venstre kræves beregning af et ekstra punkt i midten af fladen – til gengæld kan alle nabo-konfigurationer behandles ens.

En anden fremgangsmåde er, at udvikle et helt nyt subdivision-scheme specifikt til non-uniform underinddeling. Dette gøres i *[VELHO2]* med udgangspunkt i såkaldte "4-K meshes" *[VELHO1]*, her som 4-8 meshes. Den resulterende metode har den fordel, at non-uniform underinddeling af flader ikke propagerer ud til resten af meshet, se illustration 3.3. Yderligere er den mindste underinddeling givet ved bisektion, hvilket gør bedre styring af antallet af trekanter mulig. Efter n uniforme iterationer er antallet af trekanter således 2^n. Andet skridt i en 4-8 subdivision svarer topologisk til et skridt i Catmull/Clark. Metoden giver flader der er C^4, og C^1 i nærheden af ekstraordinære punkter. Til gengæld er støtten for metoden stor, hvilket har en række ulemper. For det første bliver metoden langsommere, idet flere punkter skal vægtes. For det andet betyder den større støtte en afhængighed af en større del af modellen – hvilket igen betyder at ekstraordinære punkter vil påvirke en tilsvarende større del af modellen.

En tilsvarende fremgangsmåde præsenteres i *[KOBBELT4]*. Her udvikles tillige en ny subdivision-type, der tre-dobler antallet af trekanter efter to iterationer. Dette er en fordel hvis man ønsker at styre antallet af trekanter i en model. Hvor Loop og Catmull/Clark for hver trekant giver 4^n trekanter efter n iterationer, giver denne kun $(\sqrt{3})^n$. Det foreslås, at lade subdivisionen propagere ud til nabokanter på sædvanlig vis, for at muliggøre beregning. Metoden er som Loop og Catmull/Clark C^2, og C^1 i nærheden af ekstraordinære punkter.

14 Modsat NURBS, er dette kun nødvendigt for non-uniform underinddeling. Med NURBS er det nødvendigt at tage disse forholdsregler blot en kompliceret flade skal beskrives – og altså selvom fladen tesseleres uniformt.

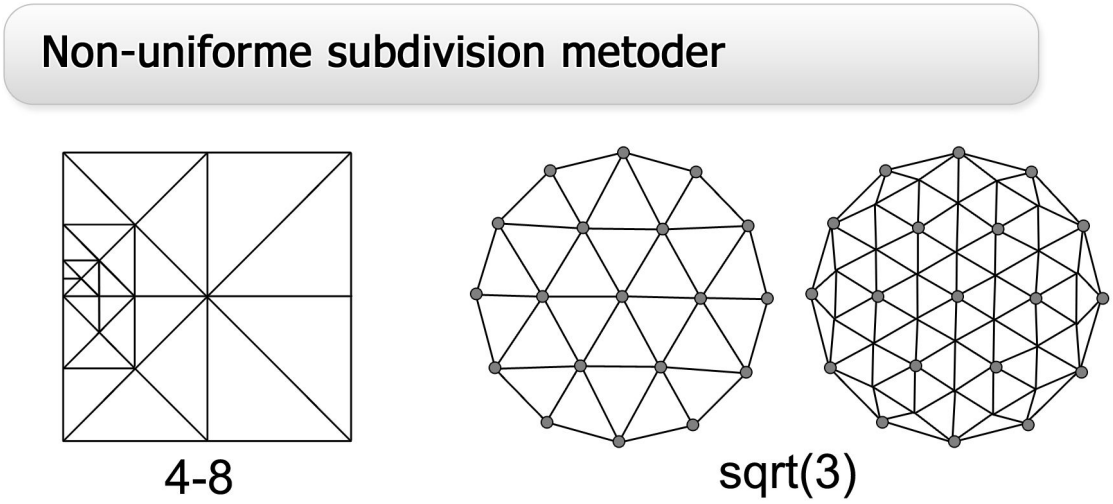


Illustration 3.3:
$\sqrt{3}$ *-metoden vises her ved uniform underinddeling, efter to iterationer. Billederne er lånt fra [KOBBELT4] og [VELHO1]*

Et godt estimat for, hvor en flade skal underinddeles, er fladens krumning. En metode til udregning af denne for arbitrære, trianguleredе, polygonmodeller er givet i *[MEYER]*. Dette er dog en tung metode at evaluere, hvorfor der i realtidssammenhænge ofte anvendes tilnærmede værdier – eksempelvis givet ved vinklen mellem nabofladers normaler. Præcisionen af underinddelingen kan så styres, ved at angive en maksimalt tilladelig vinkelforskel *[HAVEMANN1 , HAVEMANN2]*. Yderligere kriterier for underinddeling kan være størrelsen af den projicerede flade på skærmen, hvorvidt fladen kan ses, samt om den er med til at danne en kontur. En gennemgang af et par af de mere avancerede kriterier til bestemmelse af det ønskede niveau af subdivision, gives i afsnit 4.

3.2 Beregning af subdivision via tabelopslag

Da en stor del af projektet beskrevet i senere afsnit baserer sig på en afart af den i det følgende præsenterede metode, vil beskrivelsen her være væsentligt grundigere end for de andre realtidsalgoritmer. Først vil der blive givet lidt baggrundsinformation om udviklingen af metoden, og herefter vil det teoretiske fundament blive gennemgået, og der vil blive givet et rids af den praktiske implementering. Efter dette afsnit bør det være klart hvorfor tabelsubdivision er muligt, samt hvordan det skal gribes an i praksis.

Metoden præsenteret i *[BRICKHILL]* er baseret på Loop-subdivision, mens *[SCHRÖDER3, SCRHÖDER5]* baserer sig på Catmull-Clark subdivision. Da der i dette projekt bliver anvendt Loop-subdivision, vil den følgende gennemgang basere sig på denne metode.

3.2.1 Baggrund

Idéen om underinddeling via tabelopslag blev i første omgang udviklet til Sony's Playstation2 (ps2) i *[BRICKHILL]*. En ps2 er grundlæggende anderledes opbygget end en PC, og stiller således andre krav til anvendelsen. På denne platform er lager-ressourcer meget begrænsede, men til gengæld er systemhukommelse og grafikhukommelse delt. Det betyder at det bliver muligt, og praktisk, under selve visualiseringen af en model, at danne trekanter, tegne dem, og smide dem væk igen. Udnyttelsen af denne mulighed gør det muligt kun at gemme kontrol-modellen for alle geometriske

figurer, hvilket kan reducere pladsbehovet til geometri væsentligt. Yderligere bliver dyre per-vertex operationer som skinning muligt at foretage alene på kontrolmodellen.

På grund af den begrænsede lager-kapacitet, er traditionel rekursiv bredde-først underinddeling ikke en mulighed på konsoller som ps2. Dybde-først underinddeling *[PULLI,]*, kunne muligvis have været anvendt, men passer dårligt ind en streambaseret arkitektur *[SCHRÖDER5]*, og kræver desuden et afsluttende gennemløb af aller punkter for at skubbe dem til grænsefladen. I lighed med traditionelt hurtige algoritmer til evaluering af NURBS-flader i på forhånd bestemte parameterværdier, blev i stedet valgt en metode der baserede sig på sampling af de tilhørende basisfunktioner – i dette tilfælde reglerne for Loop-subdivision.

[SCHRÖDER3] beskriver en identisk metode til beregning af catmull-clark subdivision-modeller, men tilføjer muligheden for anvendelse af åbne modeller og hårde kanter. Metoden målrettes mod nyere Intel-artitektur – specifikt gøres der meget ud af, at cache-optimere beregningerne, for herved at minimere tilgangen til system-RAM. Yderligere angives her en mere intuitiv fremgangsmetode for opbygningen af de ønskede tabeller. I *[SCHRÖDER5]* implementeres metoden på programmerbar PC grafik-hardware i form af anden-generations pixelshader-hardware (se afsnit 5.5 om hardware).

3.2.2 Subdivision er linearkombinationer af støtten

Den grundlæggende observation, der leder til muligheden for at foretage underinddeling af flader via tabelopslag, er, at både reglerne[15] for placering af gamle punkter som nye punkter, er givet ved *lineære kombinationer* af nabopunkter (afsnit 2.5.1). Ethvert punkt i en trekant subdivided en gang, er således dannet ud fra en linearkombination af punkterne i støtten for denne trekant. Der mindes her om, at de punkter der er nødvendige for at foretage underinddeling af en trekant, betegnes ”støtten”. For Loop-subdivision er støtten givet ved alle polygoner der har en vertex til fælles med trekanten, se illustration 3.4.

Ethvert punkt på en given trekant, vil således kunne beregnes ud fra en følge af linearkombinationer af punkterne i støtten til trekanten. Idet sammensætningen af linearkombinationer igen giver en linearkombination, kan altså ethvert punkt på en underinddelt trekant gives direkte som en linearkombination af punkterne i støtten[16]. Et eksempel følger, med udgangspunkt i situationen på illustration 3.4.

15 En anden væsentlig observation er, at disse regler er stationære, og således ikke ændrer sig for hver iteration.
16 Det bemærkes at det ikke er tilstrækkeligt, at afbildningen er lineær, idet også resultatet skal være givet ved en linearkombination af støtten, for at være anvendelig.

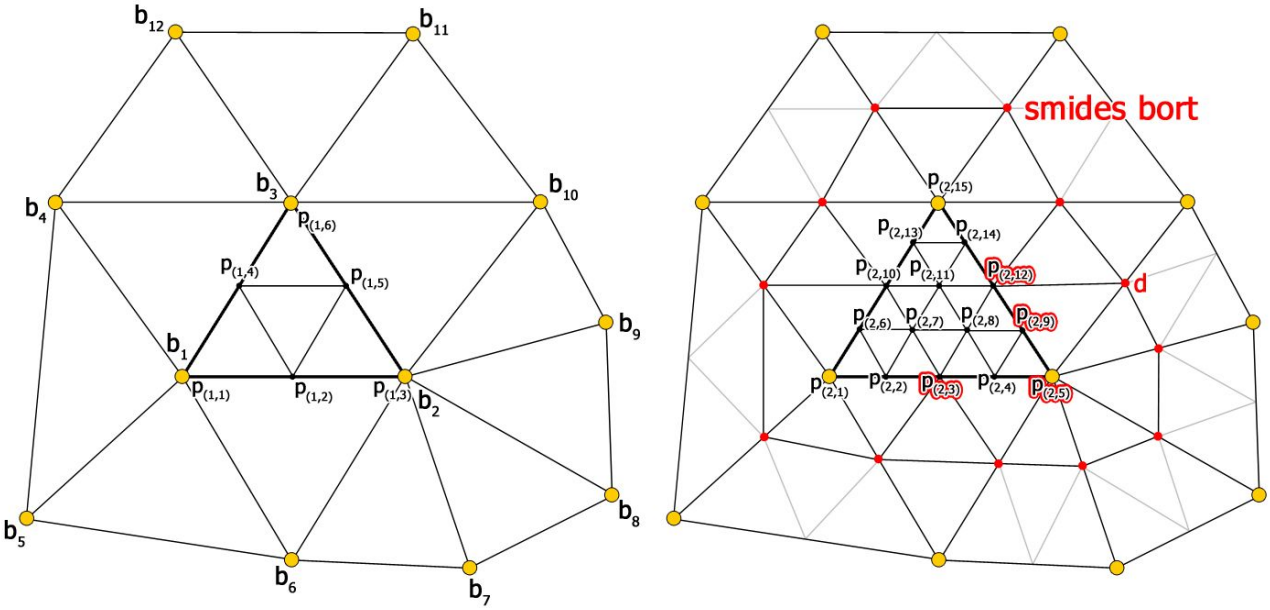


Illustration 3.4: støtten underinddeles på sædvanlig vis. For et kunne foretage yderligere subdivision, er det nødvendigt at underinddele uden for center-trekanten. Kun vægtene for de "indre" punkter gemmes dog i tabellen.

Punkterne i støtten betegnes med b_i , og et indre punkt på den underinddelte trekant betegnes med $p_{(n,k)}$, hvor *n* er subdivision iterationen, og *k* er punktets indeks. De indre punkter i trekanten efter en enkelt iteration kan findes direkte ud fra støtten. Ifølge Loop-reglerne fra afsnit 2.5.1 for nye punkter, findes punkterne efter første iteration som

$$
\begin{aligned}
p_{(1,2)} &= \frac{1}{8}b_3+\frac{1}{8}b_6+\frac{3}{8}b_1+\frac{3}{8}b_2 \\
p_{(1,4)} &= \frac{1}{8}b_2+\frac{1}{8}b_4+\frac{3}{8}b_1+\frac{3}{8}b_3 \\
p_{(1,5)} &= \frac{1}{8}b_1+\frac{1}{8}b_{10}+\frac{3}{8}b_2+\frac{3}{8}b_3 \\
p_{(1,1)} &= (1-5\beta(5))b_1+\beta(5)(b_2+b_3+b_4+b_5+b_6) \\
p_{(1,3)} &= (1-7\beta(7))b_2+\beta(7)(b_1+b_3+b_6+b_7+b_8+b_9+b_{10}) \\
p_{(1,6)} &= (1-6\beta(6))b_3+\beta(6)(b_1+b_2+b_{10}+b_{11}+b_{12}+b_4)
\end{aligned}
$$

formel 3.1

Ser vi herefter på punkterne i anden iteration, og ønsker vi specifikt at finde $p_{(2,9)}$ er dette givet som en vægtning af punkterne:

$$
p_{(2,9)} = \frac{1}{8}d+\frac{1}{8}p_{(1,2)}+\frac{3}{8}p_{(1,3)}+\frac{3}{8}p_{(1,5)}
$$

formel 3.2

d er et punkt der ikke ligger i den indre trekant, men som er nødvendigt at beregne alligevel, for at kunne foretage det andet subdivision-skridt. Punktet er givet direkte ved Loop's regler for indsættelse af punkter på kanter, som $d = \frac{1}{8}b_3+\frac{1}{8}b_9+\frac{3}{8}b_2+\frac{3}{8}b_{10}$. Idet vi nu kan indsætte udtrykkene for *d*, $p_{(1,2)}$, $p_{(1,3)}$ og $p_{(1,5)}$ i formel 3.2, er $p_{(2,9)}$ udtrykt direkte som en linearkombination af punkterne i støtten. Kun vægtningerne af de indre punkter gemmes, mens vægtene af de punkter vi blot beregner for at kunne foretage subdivision iterationerne, f.eks. *d*, smides bort. I praksis vil det være nødvendigt at beregne et enkelt bånd af punkter omkring den ønskede trekant, for hver subdivision-iteration – dette er vist som de røde punkter på illustration 3.4.

3.2.3 Generering af tabeller

I praksis er der en lettere måde at finde vægtene, end at foretage subdivision symbolsk. Ser vi igen på udtrykket i formel 3.2, er det muligt for hver af støttepunkterne at finde bidraget til det beregnede punkt $p_{(2,9)}$. Ønsker vi eksempelvis at finde ud af hvor meget støttepunktet b_1 påvirker $p_{(2,9)}$, kan vi blot reducere udtrykket efter indsættelse. Konkret betyder dette, at b_1 i ovenstående tilfælde, ved simpel isolering, får vægten

$$
\begin{aligned}
p_{(2,9)} &= \frac{1}{8}d+\frac{1}{8}p_{(1,2)}+\frac{3}{8}p_{(1,3)}+\frac{3}{8}p_{(1,5)} \\
&\Leftrightarrow \\
p_{(2,9)} &= \frac{1}{8}(0\,b_1)+\frac{1}{8}(\frac{3}{8}b_1)+\frac{3}{8}(\beta(7)b_1)+\frac{3}{8}(\frac{1}{8}b_1) \; + \; \ldots \\
&\Leftrightarrow \\
p_{(2,9)} &= \left(\frac{3}{32}+\frac{3}{8}\beta(7)\right)b_1 \; + \; \ldots
\end{aligned}
$$

formel 3.3

I praksis kan dette gøres ved at sætte værdien af et enkelt punkt, f.eks. b_1, i støtten til 1 og resten til 0. Efter subdivision af støtten, kan det direkte aflæses hvor stor vægt dette enkelte punkt har haft på hvert af de indre punkter i trekanten. Dette er analogt til at finde virkningen af at påtrykke en enheds-impuls til subdivision-filtret.

Det er naturligvis ikke nødvendigt at foretage en fuld underinddeling af støttens x, y, z værdier for at finde de søgte vægte. I stedet kan man nøjes med at tilknytte en enkelt værdi til hver vertex, et 1D-mesh, så at sige. Det er dog mere intuitivt at tænke på problemet i 3D, så illustration 3.5 viser denne situation.

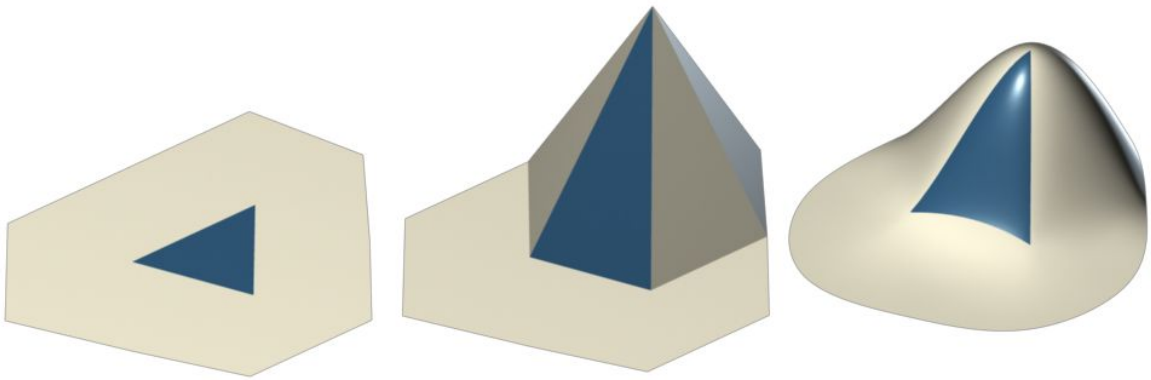


Illustration 3.5: Underinddeling af støtten for loop subdivision. Et enkelt punkt gives værdien 1, og støtten underinddeles. Efter underinddeling kan vægten af det flyttede støttepunkt aflæses direkte, for hvert af de indre punkter i trekanten. Figuren her er ikke underinddelt med loop-regler, så formen er ikke præcis.

En observation, der gør det yderligere anvendeligt at benytte tabelbaseret subdivision, er, at vægtene alene afhænger af konnektiviteten af en given trekants støtte – ikke af positionen af de specifikke punkter. To trekanter med ens støtte subdivides altså med ens vægte. Det betyder at det er muligt at danne et enkelt sæt tabeller, for alle mulige konfigurationer af valenser af en trekant, der således vil kunne anvendes på vilkårlige modeller – uafhængigt af deres konkrete geometri. Antallet af kombinationer af valenser for en trekant er dog af en sådan størrelse, at lagerressourcerne til at gemme disse i sig selv ville gøre metoden uanvendelig i praksis. For at begrænse antallet af muligheder, antages det derfor at alle trekanter har maksimalt en irregulær vertex. Dette er en antagelse der ikke holder for langt de fleste modeller, men som vil gælde for alle modeller efter en enkelt iteration af såvel trekant- som firkant-underinddeling. Antallet af værdier kan reduceres yderligere når det observeres *[BRICKHILL]*, at alle center-punkter ($p_{(2,7)}$, $p_{(2,8)}$, $p_{(2,11)}$ på illustration 3.4), dannes som en en kombination af punkter på kanterne af trekanten. Det betyder at de indre punkter altid har samme vægte i forhold til disse kantpunkter, uanset støtten. Denne ekstra optimering vil dog i de fleste tilfælde være unødvendig, idet størrelsen af tabellerne vil være moderat allerede efter de første tiltag.

Typisk vil man vælge et maksimalt ønsket niveau for subdivision, og gemme koefficienter for dette niveau. Det er dog stadig muligt at bruge samme koefficient-tabel til beregning af tidligere iterationer, så længe grænsepositioner beregnes direkte. Beregningen vil dog give mere segmenteret tilgang til hukommelsen i computeren, og således potentielt give dårligere cache-ydelse. Selv uden cache-optimering er metoden dog meget hurtig, hvorfor dette ikke er en væsentlig anke overfor brugen af det samme sæt tabeller.

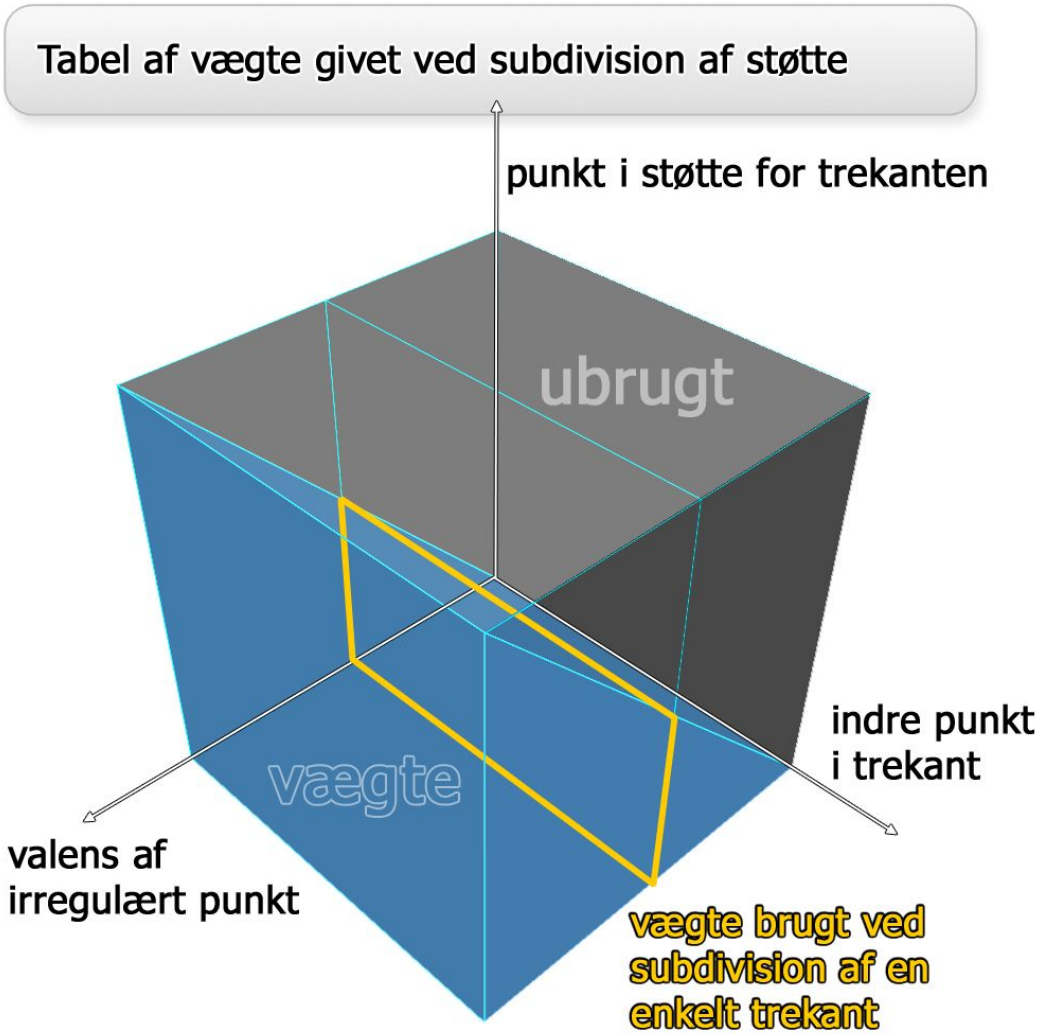


Illustration 3.6: Lagringen af vægte kan ses som en tre-dimensionel struktur, hvor kun halvdelen af pladsen anvendes. Ved beregning af en kant, anvendes vægte svarende til et tværsnit i denne struktur.

3.2.4 Beregning af subdivided trekant ved tabelopslag

Resultatet af den beskrevne underinddeling af støtten, er et tredimensionelt array af vægte for punkterne på trekanten, se illustration 3.6. For at undgå det store spild, allokeres dog ikke det viste 3D-array, men en række 2D-arrays med passende størrelse.

Antages det at vi har beregnet subdivision-tabeller for et givent subdivision-niveau, finder vi et punkt p_k i den underinddelte trekant som følger: Vægten for støttepunktet b_i, med hensyn til punktet p_k er givet ved $w_{i,k} = tabel(v, k, i)$, hvor v er valensen af det enkelte irregulære punkt i trekanten. Det nye punkt er således givet ved summen af hvert støttepunkts vægt multipliceret med dets position[17].

17 Beregningen er naturligvis ikke begrænset til positioner. Andre vertex-parametre der ønskes interpoleret via subdivision-regler, vægtes på samme måde. Dette kunne f.eks. være teksturkoordinater, som beskrevet i *[PIXAR]*.

Er *n* således antallet af punkter i støtten, er beregningen af det nye punkt givet ved

$$p_k = \sum_{i=0}^{n} w_{i,k} b_i$$

formel 3.4

Det er på sin plads at knytte en hurtig kommentar til denne ligning. Formlen ligner meget den til beregning af B-splines, hvilket nok ikke kan undre idet de to metoder er analoge. Forskellen er at ovenstående ligning tager udgangspunkt i en sampling af basisfunktionen for fladen, modsat en direkte evaluering af et aritmetisk udtryk. Yderligere varierer basisfunktion her i forhold til støtten af trekanten, hvor støtten altid er ens for B-splines.

En af de væsentlige pointer i *[SCHRÖDER3]* er, at ovenstående beregning kan cache-optimeres. Det er således væsentligt, i hvilken rækkefølge beregningerne sker. Den specifikke rækkefølge hænger tæt sammen med hvilken arkitektur der anvendes samt hvordan vægtene gemmes. Som en hovedregel bør man dog forsøge at gennemløbe hukommelsen så sekventielt som muligt, og forsøge at undgå store spring i hukommelsen.

3.2.5 Grænsepositioner og tangenter via tabeller

Det er unødvendigt at gemme separate vægte for beregning af grænsepositioner. Idet man som regel vil ønske at beregne punkter på overfladen, giver det mening at slå beregningen af punktet sammen med beregningen af grænsepositionen. Det betyder, at når tabellerne opbygges, underinddeles støtten til det niveau der ønskes beregnet, hvorefter punkterne i støtten skubbes til grænsefladen. Aflæsningen af vægte foregår præcis som tidligere. Argumentet for at dette er muligt er igen, at også beregningen af grænsepositioner sker ved en linearkombination, hvorfor sammensætningen med den almindelige subdivision også vil være en linearkombination, se afsnit 3.2.2. Det intuitive argument er, at beregningen af vægtene foregår ved helt sædvanlig subdivision af en model. Det betyder at hver iteration vil give en bedre tilnærmelse af grænsepositionerne. Støtten kan således, fuldstændig som en vilkårlig anden model, skubbes til grænsen for at finde grænsevægtene.

Beregningen af grænsetangenter er ligeledes givet som en lineær vægtning af naboerne – og vi kan således anvende samme fremgangsmåde for beregning af tangenterne som for positioner. Vi kan altså gemme to ekstra vægte for hvert punkt, for herved at kunne beregne tangenterne til alle nye punkter på trekanten. I forhold til visualisering af subdivision flader, er den hyppigste anvendelse af tangenterne at finde en normal til fladen, idet denne bruges til lysberegning på fladen. Denne normal kan findes som krydsproduktet imellem de to tangenter. Er således $w^u_{i,k}, w^v_{i,k}$ vægtene for henholdsvis t^u_p, t^v_p givet ved tabelopslag, findes til fladen en normal med enhedslængde, ved

$$n_p = \frac{t^u_p \times t^v_p}{|t^u_p \times t^v_p|}, \quad t^u_p = \sum_{i=0}^{n} w^u_{i,k} b_i, \quad t^v_p = \sum_{i=0}^{n} w^v_{i,k} b_i$$

formel 3.5

Dette betyder, at beregningen af en normal til lysberegning, kræver dobbelt så mange beregninger som positionen, og yderligere en normalisering af den resulterende normal. Det ville derfor være bekvemt, om også normalen kunne beskrives ved en direkte vægtning af støttepunkterne. Idet resultatet af et krydsprodukt ikke kan udtrykkes som en linearkombinationer af de indkommende vektorer, er dette dog ikke en mulighed.

Et krydsprodukt er lineært, hvorfor beregningen af normalen kan udtrykkes som en lineær afbildning af støttepunkterne. Resultatet kan dog ikke gives som en linearkombination af støttepunkter, men vil være kombinationer af produkter af disse. I praksis betydet det, at det er muligt at angive normalen direkte som en vægtning af støtten for en trekant, men at hvert punkt optræder med mange forskellige vægte, og at beregningen bliver uanvendeligt kompliceret. Idet normalisering yderligere er ulineær, kan det alligevel ikke undgås, at beregne denne efter normalen er fundet. Den bedste løsning er således at gemme trangenterne t_u og t_v, og beregne krydsprodukt og normalisering sammen med punktet.

I *[BRICKHILL]* er den her viste beregning af normaler via udregning af tangenterne en uacceptabel løsning ud fra en hastighedsmæssig betragtning. I stedet beregnes her normaler samt diffus belysning per vertex i kontrolpolygonen, og spekulært lys beregnes herefter for den underinddelte flade med lineært interpolerede normaler.

Tabelbaseret subdivision er primært rettet mod hurtig beregning af subdividede punkter. Idet punkter i en trekant kan beregnes uafhængigt af underinddelingen af naboerne, er metoden et ideelt udgangspunkt for adaptive metoder. Adaptivitet kræver dog beregning af et mål for, hvilke trekanter der skal underinddeles, og yderligere håndtering af naboer med ulige underinddeling. I *[SCHRÖDER3]* foreslås blot at lave triangle-fans fra nabotrekanter, for at undgå huller i modellen. Det nævnes dog samtidig, at metoden til subdivision anslås at være så hurtig, at en god heuristik for hvorvidt en trekant skal subdivides, let kommer til at tage længere tid end beregningen.

4 Metoder til forbedring af silhuetter

Der vil i dette afsnit blive præsenteret forskellige metoder der enten kan anvendes til, eller er decideret rettet imod, at forbedre silhuetten af et objekt. De af metoderne der er mest relevante for den senere udviklede metode, vil blive beskrevet mere nøje end de resterende. Visse af de præsenterede metoder har umiddelbart intet med silhuetter at gøre, men beskrives her, da de ligger til grund for andre af metoderne.

Der vil steder blive differentieret imellem to begreber – silhuetter og konturer. En silhuet er grænsen imellem objekt og baggrund. En kontur er dannet af punkter på en overflade, hvor normalen står vinkelret på øjevektoren. Et objekt kan således have flere konturer, og disse vil danne lukkede cykler henover overfladen. I litteraturen refereres der dog generelt til forbedring af silhuetter, selvom der reelt er tale om alle modellens konturer.

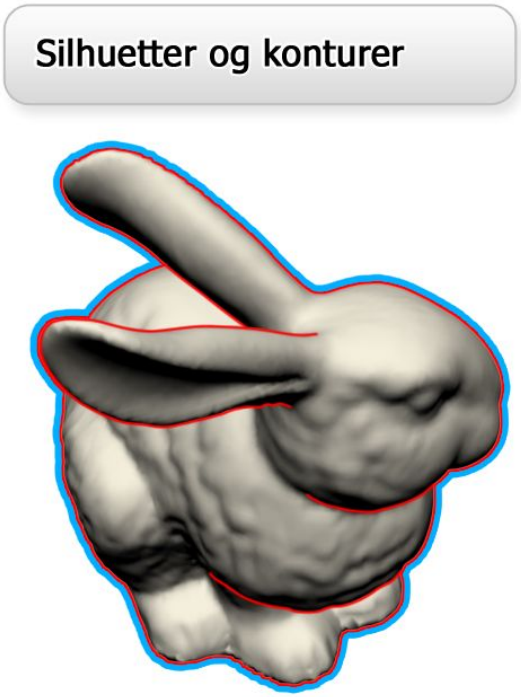


Illustration 4.1: Silhuetten er i blåt, konturerne er i rød. Det bemærkes, at billedet ikke viser konturer på bagsiden af figuren – det kan således ikke ses at konturerne danner lukkede cykler.

Der er to subtilt forskellige mål med de metoder der præsenteres i dette afsnit. Det ene er at tilføje detaljer til en model, og det andet er at approksimere en detaljeret flade, uden at skulle repræsentere alle detaljer som geometri. Hvis der tages udgangspunkt i en detaljeret model, vil der næsten altid blive foretaget en sampling af denne flade, som efterfølgende anvendes under rendering.

De her præsenterede metoder kan deles op i to typer: De geometriske, der baserer sig på at danne en passende detaljeret geometri. Den anden type refereres ofte til som ”image-space”, og dækker over at der gemmes en række relevante parametre, og ved shading af fladen findes den relevante værdi fra de gemte data.

4.1.1 Displacement Mapping

Displacement mapping baserer sig på at forskyde en flade langs sin egen normal, i forhold til et skalært højdekort. Den traditionelle måde at gøre dette *[COOK]*, er at tessellere objektet geometrisk og forskyde punkterne på den resulterende geometri direkte i forhold til højdekortet. Grundlæggende er det således i lige så høj grad en måde at konstruere geometri på, som at tilnærme en allerede givet detaljeret flade. Alle metoder der baserer sig på forskydning via højdekort, approksimerer således i en eller anden forstand displacement mapping – idet geometrien for den forskudte flade, her rent faktisk dannes.

Displacement mapping fungerer godt for anvendelser hvor mængden af geometri ikke er et problem – eksempelvis i sammenhæng med Pixar's Renderman der i forvejen underinddeler al geometri til

subpixel[18] niveau ved rendering. I realtids visualisering er mængden af geometri af afgørende betydning, da transformation af alle punkter, samt lysberegninger foretages per punkt i en model. For at gøre displacement mapping bredere anvendeligt, er der udviklet metoder til at mindske mængden af geometri metoden producerer *[DOGGET, MOULE]*. En af de afgørende parametre i disse metoder er at få silhuetten til at fremstå korrekt, da denne let afslører manglende detaljer. En anden udfordring i displacement mapping er, at få trianguleringen af den underliggende flade til at passe med det påførte højdekort. Hvis der vælges en grovere tesselering end subpixel, opstår nemt fejl hvis trianguleringen løber på tværs af højfrekvente ændringer i højdekortet. Der er lavet løsningsmodeller der foretager trianguleringen ud fra analyser af højdekortet, samt løsninger der helt undgår tesselering ved at raytrace højdekortet direkte.

Displacement mapping er vidt anvendt inden for computergrafik, men ikke brugt i ret vid udstrækning for real-tids applikationer. Den primære hindring er den meget store mængde geometri, samt at overfladen skal genberegnes hver gang enten den underliggende model eller højdekortet ændres. Visse nyere grafikkort, navnlig Parhelia-kortet fra producenten Matrox, giver mulighed for at foretage uniform triangulering af fladerne i hardware. Andre nyere grafikkort tilbyder andre features der potentielt kan anvendes til at foretage displacement mapping, se afsnit 5.5 om hardware.

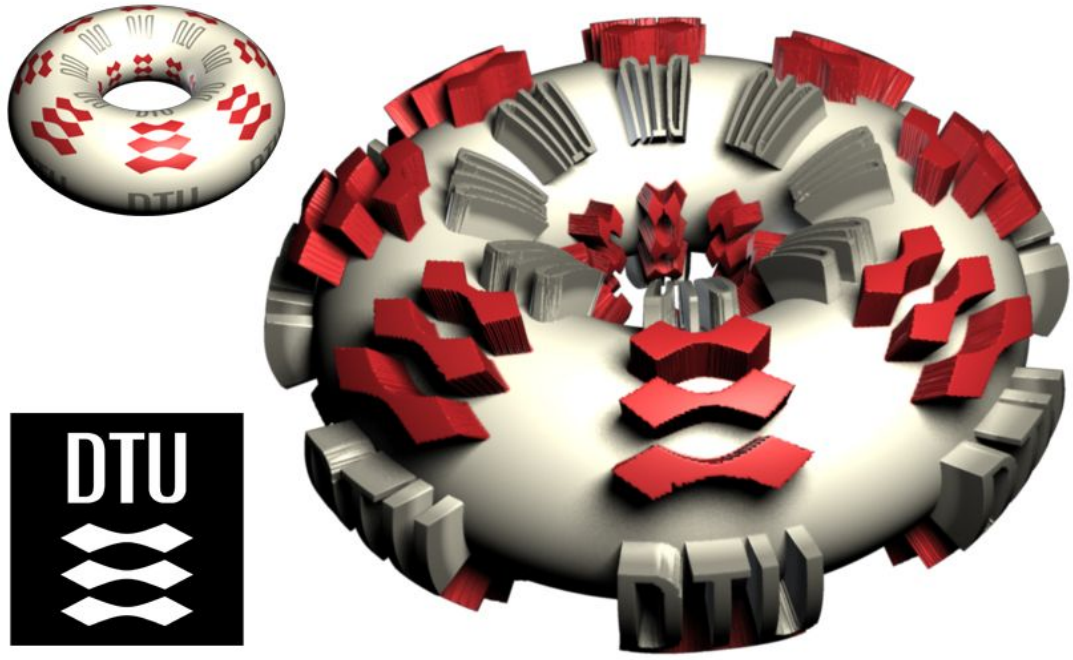


Illustration 4.2:
Billedet viser en torus påført højdekortet vist til venstre via displacement mapping. Bemærk artefakterne der optræder på grund af trianguleringen af fladen samt teksturens begrænsede opløsning .

18 I Renderman tesselleres al geometri så den største trekant er mindre end en pixel på skærmen. Dette giver god sampling, og gør det let at bruge displacement mapping.

4.1.2 Bumpmapping / normalmapping

I dette afsnit vil der blive givet en kort beskrivelse af bumpmapping. En fuld gennemgang ligger udenfor rammerne for denne rapport, og der henvises i stedet til forprojektet, eller til udførlige gennemgange i [*BLINN2*, KILGARD, WATTWATT, PEERCY].

Bumpmapping blev i sin tid præsenteret som en måde at simulere små detaljer på overflader, uden at tilføje ekstra geometri. Metoden til dette er, at variere normalen til fladen ud fra en tekstur.

En af de mest grundlæggende opgaver inden for computergrafik, er at beregne farven af en belyst flade. Belysning eller shading af en flade, giver i høj grad formen af den figur vi ser. I realtids-sammenhænge, anvendes ofte Blinns formel for tilnærmelse af lysintensiteten på en flade *[BLINN1]*. Her beregnes den isolerede lysintensitet af et punkt på en overflade, ud fra øjets position, lysets position, samt fladens normal i det givne punkt. Denne beregning betegnes lokal belysning. Følgende enhedsvektorer antages givet: Lysvektoren $\boldsymbol{L}$, fladens normal $\boldsymbol{N}$ i punktet hvor lysintensiteten beregnes, samt øjevektoren $\boldsymbol{V}$. Hvis konstanterne C_A, C_D, C_S og *specexp* beskriver materialeparametre for fladen, og I_A og I_P beskriver punktlyskildens intensitet, haves[19]

$$I = I_A C_A + I_P C_D(\boldsymbol{N}\cdot\boldsymbol{L}) + I_P C_S(\boldsymbol{N}\cdot\boldsymbol{H})^{specexp}$$

formel 4.1

$\boldsymbol{H}$ er her en "halvvektor", der er rettet midt imellem $\boldsymbol{V}$ og $\boldsymbol{L}$, og således kan findes ved:

$$\boldsymbol{H} = \frac{\boldsymbol{L}+\boldsymbol{V}}{\|\boldsymbol{L}+\boldsymbol{V}\|}$$

formel 4.2

19 Såvel $\boldsymbol{N}\cdot\boldsymbol{L}$ og $\boldsymbol{N}\cdot\boldsymbol{H}$ kan i praksis blive negative, hvorfor de resulterende værdier sædvanligvis begrænses til at ligge over 0. Det bemærkes yderligere at materiale- og lys-parametre varierer meget imellem implementeringer. Eksempelvis kan angives separate værdier for en lyskildes diffuse og spekulære intensitet, eller materialer have mulighed for at være selvlysende.

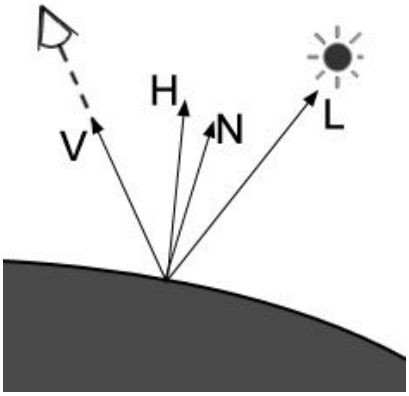


Illustration 4.3
V er viewing vektoren
L er er lysvektoren
N er normalen til fladen
H er halfvektoren

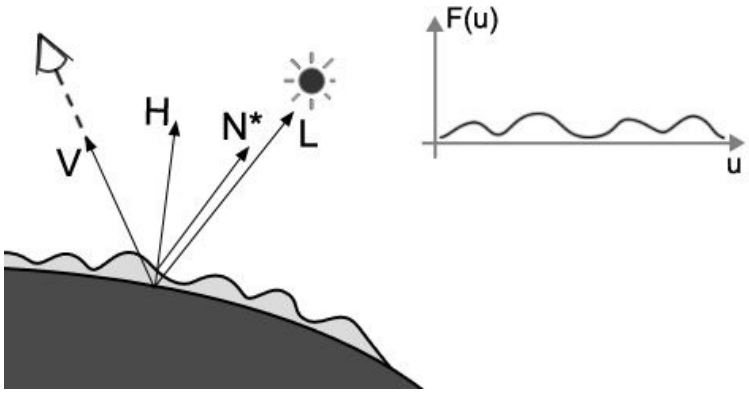


Illustration 4.4
Det bemærkes, at V, H og L gives i forhold til positionen på den gamle flade. Dette negligeres dog, da højdekortet antages meget mindre end fladen.

En måde at ændre udseendet af fladen, er at forandre lysberegningen. Idet normalen indgår som et væsentligt element i lysberegningen, vil en ændring af denne betydeligt ændre fladens fremtoning. En måde at gemme denne normalændring, er, at lagre en række normalforskydninger henover fladen, i en tekstur. Det er således muligt, at tilføje detaljer til en model uden at øge mængden af geometri. Metoden blev foreslået i *[BLINN2]* i 1978. Idéen var, at simulere at et højdekort forskyder fladen i normalens retning, ved at anvende normalen til den forskudte flade på den oprindelige flade. Modsat displacement mapping hvor selve geometrien ændres, forandres her kun normalen i lysberegningen. I praksis anvendes højdekortet sjældent direkte. I stedet præ-beregnes normalforandringerne der resulterer ved flade-forskydning, og disse gemmes i et ”normalmap”. Navnet kommer af, at forskydningerne er givet direkte ved normalerne til højdekortet.

En forudsætning for at bumpmapping fungerer er, at forskydningerne kun er ganske små. Antagelsen om en lille forskydning er fornuftig af flere grunde. Hvis forskydning er for stor holder illusionen om en rynket flade ikke i områder omkring silhuetten, da forskydningen her instinktivt burde ses. Et eksempel på dette ses på illustration 4.5. En ulempe ved kun at variere fladenormalen er, at områder af fladen der ellers ville ligge i skygge, bliver oplyst. Synligheden af dette problem kan dog mindskes, ved at kigge på den geometriske flades normal, og dæmpe lysintensiteten når denne peger væk fra lyset.

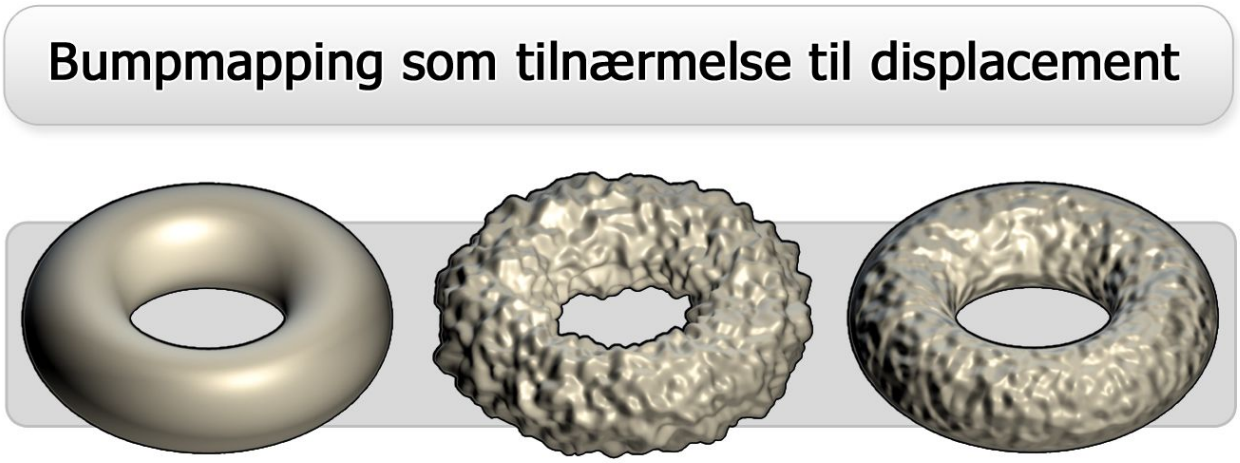


Illustration 4.5:
I midten ses displacement-mapping, og til højre tilnærmelsen via bumpmapping.

En anvendelse af bumpmapping der har fundet stor anvendelse, er det såkaldte ”normalmapping”. Denne idé blev først præsenteret i *[COHEN]*, der ud fra en detaljeret flade dannede en simplificeret lavpolygon-model og gemte normalerne til den fjernede geometri i et normalmap. Efterfølgende lysberegninger på fladen kunne således gendanne nogen af de tabte detaljer, hvilket gjorde det muligt at simplificere modeller betydeligt, uden større visuelt tab. Flere af de i dette afsnit følgende metoder, udvider dette koncept, ved at gemme yderligere parametre for fladen, så som højde (parallax / relief) og krumning (VDM / GDM). Dette gør det muligt at foretage en bedre tilnærmelse til fladen, på bekostning af større ressourceforbrug.

4.1.3 Runtime adaptive Subdivision Surfaces

Som det blev nævnt i afsnit 3, giver non-uniform subdivision mulighed for at tage særligt højde for underinddeling af silhuetten. Den mest normale fremgangsmåde er at kigge på alle konturer, uden at teste om de specifikt ligger på silhuetten. Der er to årsager til dette. Den første er, at det er hurtigt at foretage et estimat for, om et område af en flade ligger på konturen – mens det er vanskeligere at vurdere om en given del af en kontur indgår i silhuetten. Den anden årsag er, at det sædvanligvis er ønskeligt også at forbedre indre konturer, da der ellers er områder på den indre del af fladen der vil fremstå kantede.

I *[ALLIEZ]* præsenteres en metode baseret på $\sqrt{3}$ -subdivision *[KOBBELT]*, der tager specifikt hånd om konturer. Metoden baserer sig på iterativ analyse og underinddeling af modellen. Kanter antages her at tilhøre en kontur, hvis normalerne til nabofladerne peger i hver sin retning. Kontur-trekanter underinddeles ekstra, og de nye trekanter analyseres igen med samme heuristik. Idet $\sqrt{3}$ -subdivision giver mulighed for lokal detaljering, propagerer den høje detaljering omkring konturen kun langsomt til resten af modellen, se illustration 4.6. En lignende heuristik vises for Loop-subdivision i *[AKENINE]*, hvor der yderligere differentieres imellem retningen af en trekants naboer. Målet med metoden er ikke at forbedre silhuetten, men sikkert at undgå underinddeling af ikke-relevante områder af modellen.

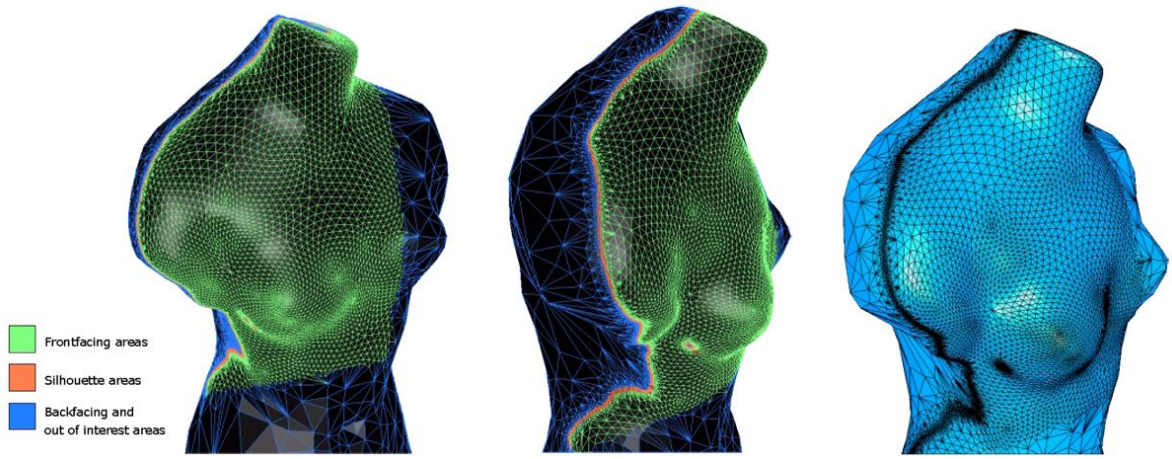


Illustration 4.6:
Her ses $\sqrt{3}$ *-subdivision anvendt til at forbedre silhuetten og fronten af en model. Bemærk at også konturen på undersiden af højre bryst identificeres. Metoden tager tillige højde for viewfrustum. Længst til højre ses figuren i wireframe, uden farvekodning.*
Billederne er lånt og let modificeret fra [ALLIEZ]

I *[AZUMA]* præsenteres en metode der er rettet imod visualisering af statiske subdivision-modeller. Metoden baserer sig på wavelet-dekomposition af en subdivision-model. Resultatet er en metode der hurtigt og præcist kan identificere kritiske områder af modellen, herunder silhuetten. For hver trekant forberegnes en kegle, med en radius sådan at alle normaler til den underinddelte trekant, er indeholdt i keglen. Dette giver en hurtig test for om en trekant, ved underinddeling, kan give geometri der vender fremad. Kontur-områder identificeres ved denne kegle, og den ønskede præcision sættes her i vejret.

En teknik der bør nævnes, men ikke er direkte rettet imod hurtig visualisering, er anvendelsen af displacement mapping på subdivison surfaces. I *[LEE]* vises hvorledes et højdekort kan anvendes til tilføjelse af detaljer til en subdivision flade . Metoden retter sig primært imod kompression, opnået ved at angive en model som en grov subdivision-model, og et tilhørende højdekort. I *[CHEN]* vises en alternativ fremgangsmåde, der forskyder vertices som en del af subdivision-processen, frem for bagefter. Idet subdivision kan ses som en slags lavpasfiltrering, og højdekortet udglattes sammen med resten af modellen, bliver de resulterende forskydninger meget bløde. En fordel er dog, at der kan gemmes forskellige detaljer for de enkelte subdivision-iterationer.

4.1.4 PN-triangles

Målet med denne metode er, at forbedre udseendet af allerede eksisterende computerspil-modeller. Fokus er således på, at danne en pæn, blød flade, ud fra en lavpolygon model der ikke er konstrueret specifikt til metoden. Konkret var det målet, at fjerne de hårde silhuet-kanter og forbedre shadingen af modellerne. Det var et krav til PN-Triangle metoden, at beregningerne skulle passe ind i den daværende pipeline på grafikkort.
[VLACHOS]

PN-triangles står for Punkt/Normal-trekant. Navnet stammer fra, at metoden danner bløde flader alene ud fra punkter og normaler for en enkelt trekant. Der er flere fordele ved dette. Muligheden

for at behandle hver trekant for sig, gør metoden trivielt paralleliserbar, hvilket igen gør den velegnet til implementering i en grafik-pipeline. Yderligere vil normaler næsten altid indgå i de data der er givet for geometriske modeller til brug i spil, hvorfor metoden vil fungere på de fleste, også ældre, modeller.

Et PN-triangle-patch er en kubisk trekant beziér-flade, hvor kontrol-punkterne er dannet alene ud fra punkter og normaler til en trekant i den oprindelige flade. Idet der findes effektive algoritmer til evaluering af beziér-flader *[GRAVESEN]*, er den primære udfordring at danne kontrolpunkterne til disse flader. Et kubisk beziér-patch kræver 10 koefficienter, se illustration4.7 a). Problemet er således underbestemt, når der kun haves tre punkter og normaler, hvorfor fladen kan dannes på flere måder. I artiklen præsenteres følgende heuristik for generering af kontrol-punkter, der giver bløde flader og interpolerer den oprindelige model:

1. Kontrolpunkterne i hjørnerne placeres i hjørnerne af trekanten.
2. Kontrolpunkterne på kanterne placeres indledningsvist uniformt over kanten – dvs. $\frac{1}{3}$ og $\frac{2}{3}$ langs kanten.
3. Det midterste kontrolpunkt, $\boldsymbol{b}_{111}$, placeres som gennemsnittet af hjørnepunkterne.
4. Til hvert midlertidigt placeret kant-kontrolpunkt, findes det nærmeste punkt på normalplanet til nabo-hjørnepunktet. Se *c)* på illustration 4.7.
5. Kontrolpunktet i midten, $\boldsymbol{b}_{111}$, forskydes nu fra den midlertidige position, til gennemsnittet af de seks kant og hjørne-punkter. Den endelige position findes ved at fortsætte forskydning med en halv gang ekstra.

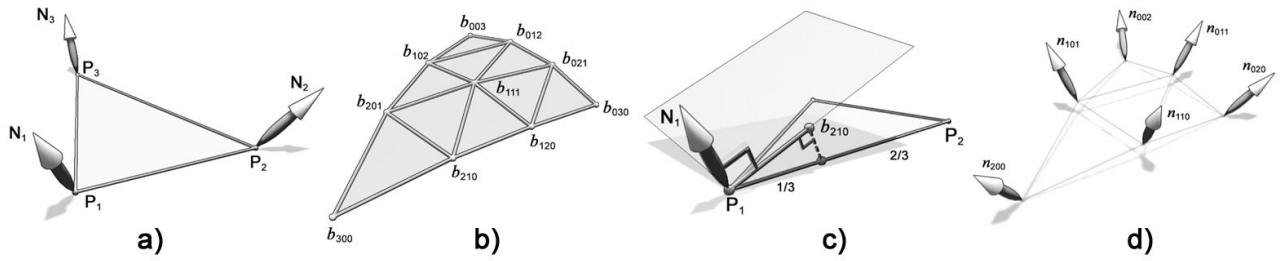


Illustration 4.7:
a) er den trekant der tages udgangspunkt i. b) er kontrolpolygonen for et kubisk trekant-bezier patch. c) er heuristikken for dannelse af de kontrolpunkter der ligger på kanterne.
Billedet er lånt fra [VLACHOS], og let modificeret.

For at sikre kontinuert shading imellem flader, dannes en normal der er uafhængig af punkterne i beziér-fladen. Normalerne approksimeres med en kvadratisk funktion, givet som endnu en trekantet beziér-flade. At fladen er kvadratisk betyder, at der kun skal findes seks kontrolpunkter, illustration 4.7 d). De tre hjørnepunkter gives som før direkte ud fra hjørnerne i trekanten, men nu ved normalerne. Herefter mangler kun de tre kontrolpunkter på midten af kanterne. Disse dannes ved at beregne et gennemsnit af de to hjørnepunkter til hver kant, og spejle dette i planet der står vinkelret på kanten. Det endelige kontrolpunkt fås ved normalisering af denne normal.

Muligheden for skarpe kanter tilføjes til metoden, ved først at detektere kanter med splittede normaler, og tilføje et ekstra sæt kanter omkring denne. Dette er samme fremgangsmåde som beskrevet for subdivision surfaces i afsnit 2.7. En væsentlig ulempe ved denne metode er som

nævnt, at den giver mange overflødige trekanter på et meget lille område.

Metoden var målrettet imod meget lavpolygon-modeller der beskrev organiske former og ikke var beregnet på subdivision. På denne type modeller giver denne teknik ganske pæne flader, der ikke svinger for meget, og ikke gør modellerne for runde.

Der er dog en kraftig tendens til at fladerne bøjer skarpere omkring de oprindelige kanter i modellerne, hvilket tydeliggører det grove udgangspunkt. Metoden indgik en overgang, under navnet "TRUFORM", som en del af ATI's grafikkort på linje med andre højniveau-flader så som NURBS. Denne anvendelse er aldrig rigtig blevet en succes – men kunne måske blive det, hvis modellerne blev målrettet imod metoden.

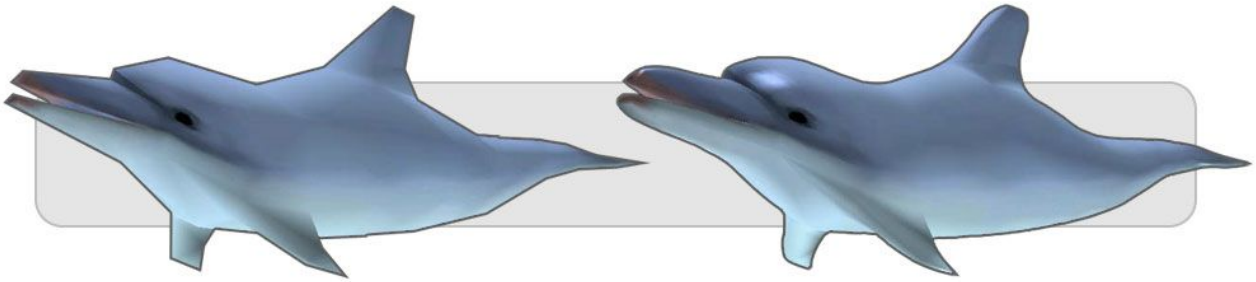


Illustration 4.8:
Modellen her er taget fra ATI's TRUFORM-demo. Modellen er således lavet til brug med PN-triangles, hvilket må betragtes som snyd.

4.1.5 Viewdependent Progressive meshes

"Progressive meshes" *[HOPPE3]* er en metode til realtime Level-of-Detail, der giver mulighed for at ændre detaljegraden af et objekt kontinuert. Dette sker ved, ud fra en detaljeret model, at danne en simpel model via kant-kollaps *[HOPPE4]*, og for hver kant gemme information om hvilke punkter der er kollapset. Metoden baserer sig på, at der findes en topologisk invers operation af et kant-kollaps, nemlig et vertex-split, se illustration 4.9. Et "progressive mesh" (PM) er således givet ved en meget simpel model, samt en ordnet følge af vertex-split. Udførelsen af en vertex-split operation vil forbedre tilnærmelsen til den detaljerede model. Det er herved muligt at styre den ønskede præcision, blot ved at vælge et antal vertex-split operationer der ønskes udført.

Metoden er udvidet til at være tage højde for hvor modellen ses fra *[HOPPE2]*. Udgangspunktet er primært modellens normaler, samt kameraets view-frustum. Yderligere specialiseres den underliggende PM-datastruktur til at inkorporere hurtigere beregning på normaler og geomorphing. Metoden tager ikke specifikt højde for at forbedre silhuetten af et objekt, men gør det alligevel implicit, idet der anvendes et mål for den fejl en simplificering vil forårsage i skærmkoordinater.

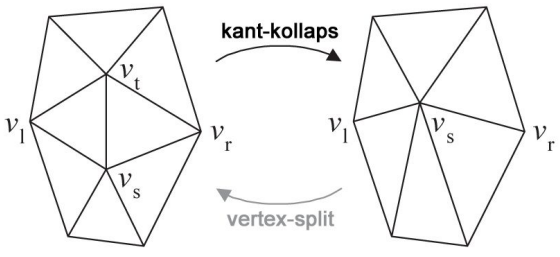


Illustration 4.9:
Illustrationen er lånt fra [HOPPE4].

Områder omkring silhuetten vil således blive mere underinddelt, om end metoden i højere grad baserer sig på krumningen af fladerne.

4.1.6 Silhouette-clipping

Denne metode er specifikt rettet mod at korrigere problemet med kantede silhuetter. Ud fra en højdetaljeret model dannes et lavdetaljeret ydre-konvekst hylster for modellen. Samtidig genereres en mængde af træer, bestående af polygon-kanter, hvori der hurtigt kan søges efter silhuet-kurver *[SANDER]*. Under rendering findes silhuetkurverne af den højdetaljerede model, ved en traversering af disse træer. Silhuetkurverne projicerer herefter til billedplanet, hvor der tegnes en trianguleret 2D-polygon. Denne 2D-polygon repræsenterer således silhuetten for den højdetaljerede model. Den endelige rendering af objektet, sker ved at rendere et ydre konvekst hylster og bruge 2D-silhuetkurven til, via stencil-bufferen, at fjerne alle pixels der ikke ligger indenfor silhuetten. Det vises yderligere hvordan metoden kan kombineres med "appearance preserving simplification" *[COHEN]* så også normalerne på objektet bliver korrekte.

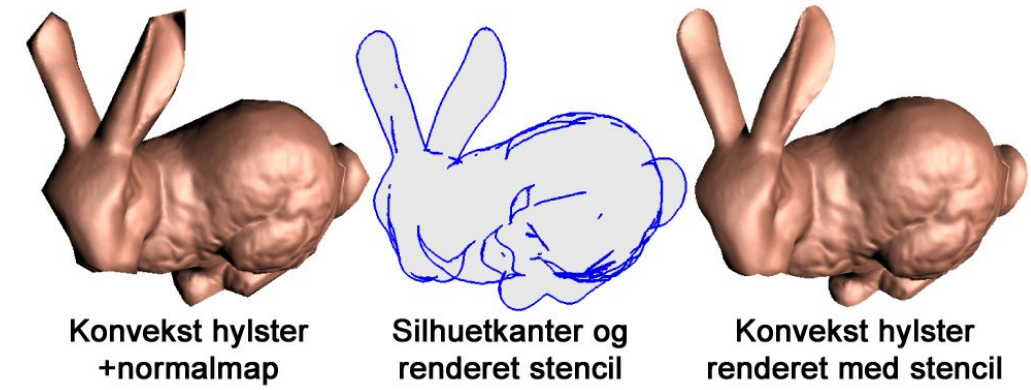


Illustration 4.10:
(billederne er lånt fra [SANDER], og let modifieret.

Metoden kan håndtere objekter med arbitrær genus, og fungerer uanset krumning af fladen. Hvis der er overlap imellem forskellige dele af objektet fejler metoden dog. Dette bunder i, at 2D-polygonen dannes strengt ud fra silhuetten af objektet, og således ikke løser problemer for indre konturer. Et eksempel er når ørerne på stanford-kaninen skygger for noget af kroppen. I dette tilfælde vil ørerne fremstå i deres grove, kantede form, som til venstre på illustration 4.10. Der er umiddelbart ingen muligheder for at animere de resulterende objekter.

En tidligere metode med samme formål, kaldet silhuette-mapping *[HOPPE5]*, beskriver en metode til, ud fra et stort antal renderinger af modellen fra forskellige retninger, at genskabe silhuetten. Dette sker ved at interpolere renderingerne til det nuværende øjepunkt, og bruge alpha-kanalen som stencil-buffer. Som før renderes et ydre konvekst hylster af modellen. Metoden begrænser øjepunktet til at bevæge sig på en kugleskal, hvis der anvendes perspektivisk projektion. Hvis øjet skal afvige fra denne, skal silhuet-maps for flere afstande beregnes. Denne metode anvender ikke normalmaps, men projicerer i stedet renderingerne direkte på modellen.

4.1.7 Parallax mapping

Parallax mapping er egentlig ikke noget forsøg på at forbedre silhuetten af objekter. Det er taget med alligevel, fordi det er en af de simpleste metoder, der tager udgangspunkt i normalmapping, og forbedrer udseendet ved at tilføje lidt information – her i form af et højdekort. Formålet med metoden er at tilføje effekten af, at ting der er tættere på kameraet bevæger sig hurtigere, når kameratet bevæger sig, end ting der er langt fra – dette betegnes sædvanligvis "parallakse" *[CARSTENSEN]*.

Antagelsen fra bumpmapping om, at højdeforskydningen ikke er ret stor, gør at denne effekt ikke optræder for traditionel normalmapping, se afsnit 4.1.2. Parallax mapping *[KANEKO, WELSH]* forsøger at rette op på dette problem ved, udover normalerne til hvert punkt, også at gemme et højdekort for fladen - typisk som alpha-kanalen i teksturen for normalmappet. Ved shading foretages et enkelt opslag i højdekortet med de teksturkoordinater som fladen har i det shadede punkt. Den forskudte flade approksimeres herefter med et plan i den fundne højde, parallelt med den oprindelige trekant. Teksturkoordinaterne i punktet hvor øjevektoren skærer dette plan findes, og diffus farve og normaler for dette nye punkt anvendes herefter til lysberegning i punktet.

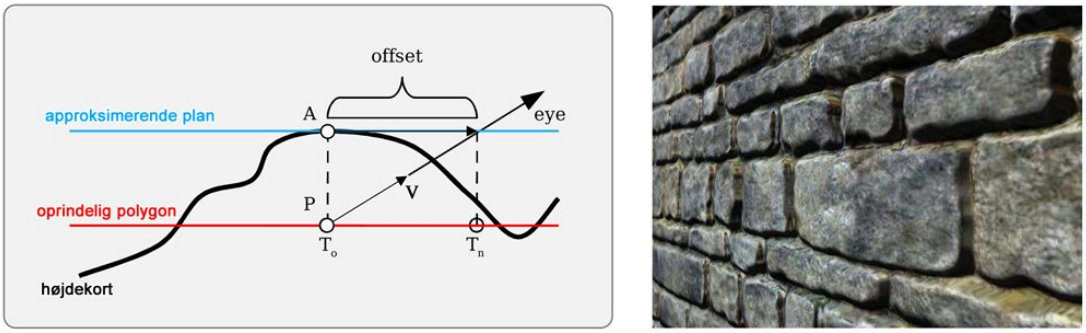


Illustration 4.11:
Billeder lånt fra [WELSH] og let modificeret

Parallax-mapping er en meget billig operation, og kan trivielt forbedres ved at foretage et par iterationer i stedet for blot et enkelt spring. En måde at styre antallet af iterationer er givet i *[MCGUIRE]*, hvor vinklen til fladen samt afstanden direkte giver antallet af iterationer. Parallax mapping er grundlæggende en meget simpel søgning der foretages i højdekortet, og den er som sådan meget begrænset, og særligt følsom overfor hvilken vinkel fladen ses i.

Parallax-mapping gør ikke noget forsøg på at forbedre silhuetten for flader. For simple, åbne flader kan kanterne dog trivielt forbedres – dette gælder f.eks. for et plan, hvor alle opslag der ryger uden for teksturen blot smides bort. I *[BRAWLEY, TATARCHUK]* vises en udvidelse af metoden der forbedrer præcisionen af metoden ved at tage flere samples i højdekortet. Metoden garanterer ikke at den korrekte skæring med højdekortet findes, men kan gøres vilkårligt præcis ved at øge antallet af samples. I *[TATARCHUK]* beskrives også måder at håndtere en række af de problemer der kan opstå ved praktisk anvendelse af parallax-mapping – f.eks. problemer omkring diskontinuert texturemapping. Det skal dog bemærkes at der ikke præsenteres løsninger på problemerne, men hovedsageligt måder til at gøre dem mindre synlige[20].

20 Eksempelvis ved at sætte en stor kasse i vejen, så problemet ikke kan ses!

4.1.8 Relief texture-mapping

To forskellige metoder går under nogenlunde samme navn, relief-mapping. Den første metode *[OLIVEIRA]* tager udgangspunkt i almindelige bitmap-billeder med et tilknyttet højdekort. I en to-delt process, foretages et 1D-warp af hver pixel i billederne i forhold til deres højdekort og vinklen som billedet ses fra. Metoden virker kun for planer, men det vises hvordan "vinger" kan tilføjes så også pixels kan tegnes, selvom de warpes til et område udenfor det renderede plan. Metoden har traditionelt været anvendt til visualisering af geologiske højdekort.

Den anden metode er en hardware implementering med samme udgangspunkt *[POLICARPO]*, nemlig et bitmap med tilhørende højdeforskydninger. Denne metode finder skæringspunktet imellem højdekort og øjevektor ved at foretage to søgninger i højdekortet. Dette er en traditionel tilgang til at finde globale minima – hvor der først foretages en søgning der skal give et punkt i nærheden af det globale minimum, hvorefter en mere præcis søgning finder det eksakte minimum *[LOAN]*. Fremgangsmåden i hardware-accelereret relief-mapping er først at sample højdekortet lineært langs øjevektoren, efterfulgt af en binær søgning for at finde det præcise skæringspunkt. Metoden garanterer ikke at det korrekte punkt findes, idet små toppe i højdekortet let overses i den lineære søgning. Eneste umiddelbare løsning er at øge antallet af samples – og i sidste ende at sample alle texels i højdekortet, langs med øjevektoren. En udvidelse af metoden er under udvikling under navnet "curved relief mapping". Den tilføjer forbedret rendering af silhuetter, ved at tage højde for krumningen af den flade som højdekortet tilføjes til *[GL1]*.

4.1.9 VDM / GDM

De to forkortelser i overskriften står for henholdsvis "View-dependent Displacement Mapping" *[WANG1]* og "Generalized Displacement Mapping" *[WANG2]*. VDM er baseret på højdekort med forskydninger ind i fladen, mens GDM udvider teknikken til at omfatte muligheden for at tilføje arbitrær geometri til overfladen. Det betyder altså at den "displacede" flade ikke nødvendigvis skal kunne angives som en forskydning i forhold til den oprindelige flade, men godt kan have overlap[21], se illustration 4.12. Heraf kommer det "generaliserede"[22]. Metoden tager altså ikke udgangspunkt i et højdekort, men decideret geometri.

Begge metoder baserer sig på at præ-beregne en sampling af displacement-mappet, der anvendes ved rendering af objekterne. I lighed med silhuette-mapping, samples overfladen fra en række øjepositioner fordelt på en kugleskal. For et givet punkt på displacement-mappet, findes for en række anskuelsesretninger den korrekte forskydning til den approksimerede flade – se illustration 4.12. For hver retning gemmes teksturkoordinaterne for skæringspunktet. Dette sæt teksturkoordinater anvendes under rendering til at tilgå diffuse farve, normalmaps, "selv-skygge" ("Precomputed Radiance Transfer", *[SLOAN]*) etc. I VDM anvendes krumningen af objekterne i beregningen. Det er derfor nødvendigt, at foretage en sampling af højdekortet ikke kun for alle retninger i alle punkter, men også for alle krumninger. Den eneste objekt-specifikke information der præ-beregnes, er netop denne krumning i alle punkter. Samplingen af højdekortet giver en forholdsvist grov approksimation af silhuetten. For at forbedre denne, gemmes yderligere sæt data, der for hvert punkt, for hver anskuelsesretning, angiver den præcise retning for silhuetten. GDM er i stand til at rendere vilkårlige forskydning, såvel indad som udad, for en given geometri. Dette sker

21 Der nævnes intet om huller eller tunneller i paperet.

22 I beskrivelsen her vil der dog blive refereret til et "displacement map". Det er dog vigtigt at være opmærksom på, at dette ikke er et sædvanligt højdekort.

ved at danne vinger ud fra alle trekanter i modellen, og sætte disse sammen til prismer der efterfølgende renderes. Der dannes således et ydre konvekst hylster omkring den ”virtuelle”, forskudte flade.

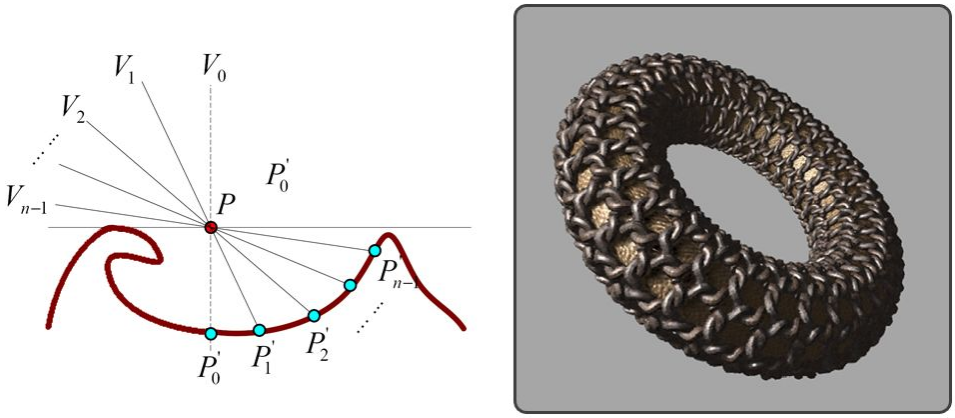


Illustration 4.12:
Til venstre vises samplingen af fladen for hvert punkt i Billedet til højre viser GDM i en situation der ikke kan beskrive som en forskydning af den grundlæggende flade langs med normalen. Billederne er lånt fra [WANG1, WANG2] og er let modificerede.

Såvel VDM som GDM baserer sig på tilføjelsen af mønstre der gentager sig henover en flade. Der er således ikke gjort nogen overvejelser omkring hvad der sker i kantområder af ”højdekortet”, eller ved samlinger imellem to teksturer. Metoderne kræver yderligere ganske meget data: For VDM anvendes i artiklen et 128x128 højdekort, der samples 32x8 gange for hver pixel. Dette giver 64MB data, som det dog vises hvordan kan pakkes til ca. 4MB. GDM anvender en anelse mere data. Et højdekort på 128x128 er dog ikke ret meget, og giver kun acceptable resultater fordi det gentages mange gange henover de viste flader.

Animation er ikke nævnt i nogen af de to artikler, men morphing imellem flere sæt beregninger burde være mulig – hvilket dog yderligere ville øge mængden af data. Yderligere virker det som om, at for lokal belysning, burde GDM kunne modificeres til at virke med generel animation. VDM foretager en beregning af krumningen for fladen, der dog kan tilnærmes ganske hurtigt, hvorfor det ikke burde være en decideret hindring for at anvende metoden på animerede flader.

4.1.10 Displacement mapping med afstandsfunktioner

Metoden baserer sig på sphere-tracing, der blev vist i *[HART]* til raytracing af implicitte flader. Denne fremgangsmåde giver mulighed for at anvende arbitrær volumendata som tilføjelse af geometri til modellen *[DONELLY]*. I lighed med GDM er displacement mapping via afstandsfunktioner således ikke begrænset til at tage udgangspunkt i højdekort. Det betyder at alle typer geometri kan tilføjes til en overflade, inklusive overlappende flader, tunneller og huller. Al geometri skal dog ligge indenfor det oprindelige objekt.

Ideen er, at danne et 3D-afstandskort ud fra den geometri der ønskes tilføjet til fladen. Når fladen skal visualiseres, raytraces igennem afstandsvolumenet. For hvert punkt i volumenet kendes

afstanden til den nærmeste flade. Givet en øjevektor V og et punkt p_0, vil punktet p_1 således være tættere på det korrekte skæringspunkt med fladen – om end ikke tættere på fladen, se illustration 4.13. I kontrast til den lineære søgning foretaget i relief-mapping, er hver iteration her garanteret ikke at bevæge sig igennem detaljer i højdekortet. Metoden konvergerer således imod det korrekte skæringspunkt med den tilnærmede flade. Kriterier for afslutning af algoritmen kan enten være et maksimalt antal skridt, eller når afstanden bliver tilstrækkeligt lille.

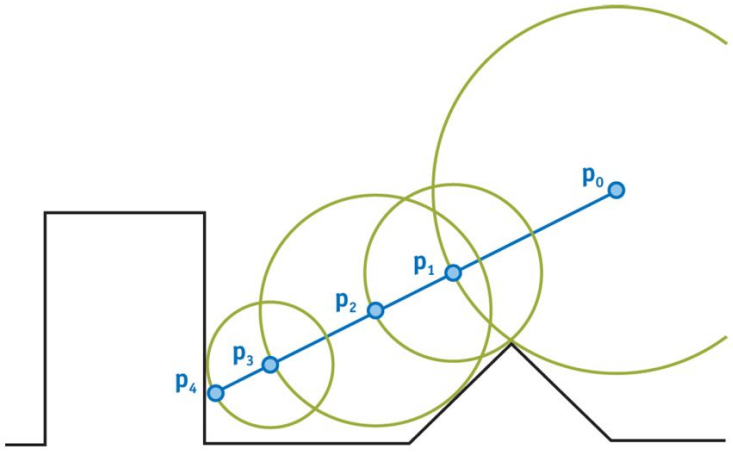


Illustration 4.13:
Hver iteration laver et opslag i en afstandsfunktion for højdekortet.
Billedet er lånt fra [DONELLY].

Rendering med afstandsfunktioner er ganske hurtig, idet en iteration begrænser sig til et opslag i afstandsteksturen, en multiplikation af øjevektoren med denne afstand, og en summation med det sidst fundne punkt. Som for VDM/GDM er det relativt meget data der skal gemmes. Yderligere fungerer metoden ikke for krumme flader, da der ikke findes nogen umiddelbar mulighed for at warpe et afstandskort i forhold til ændret geometri. En mulighed ville være, som for GDM, at lave afstandsberegninger for alle krumninger. Dette ville dog, uden kompression af afstandsdata, resultere i uhåndterligt store mængder data. Metoden giver således ingen anvendelig forbedring af silhuetten, men giver, som parallax-mapping, alene mulighed for afvise samples der slet ikke skærer et højdekort påført et plan.

4.2 Opsummering

Der er i dette afsnit beskrevet en række metoder der tidligere har været anvendt til at forbedre udseendet af silhuetter.

En række af disse baserer sig på beskrivelse af en detaljeret model ved en model med en lav detaljegrad, samt et højdekort. En anden trend er dog at sample fladen uniformt fra en række retninger, og interpolere imellem disse samplinger under rendering. Dette giver mulighed for at tilføje arbitrær geometri frem for blot forskydninger langs normalerne.

Generelt set kan identificers to muligheder at forbedre silhuetten af en model. Den ene er at tilføje geometri lokalt, afhængigt af synsretningen. Den anden er at danne et lav-detaljeret konvekst hylster

for den model der ønskes approksimeret, og undlade at rendere pixels der falder udenfor modellen.

Metoderne præsenteret i dette afsnit, skal ses i den kontekst de er lavet, og ikke mindst den tid de er lavet i. Udviklingen af grafikkort der foretager transformationer og beregning af belysning, har i stigende grad gjort det fordelagtigt at placere geometri statisk på grafikkortet. Flere af de her viste metoder, baserer sig på udvælgelse af geometri, der herefter sendes til grafikkortet og tegnes. For statisk geometri, vil dette i dag ofte ikke kunne betale sig. Evalueringen af komplicerede heuristikker, eller konstruktionen af geometri, kombineret med overheadet ved at læse data fra systemhukommelse og sende det til grafikkortet, er en væsentlig begrænsning på mængden af geometri (afsnit 5.5). For dynamiske eller procedurale modeller, er statisk lagring på grafikkortet dog ikke en mulighed, hvilket retfærdiggør metodernes relevans. Yderligere er situationen meget anderledes på andre platforme end PCen, hvilket gør adaptive metoder relevante i en overordnet sammenhæng.

De nyere af de præsenterede metoder er helt GPU-baserede, og retter sig hovedsagelig mod tilføjelsen af såkaldt ”mesostruktur” *[WANG2]*. Hermed menes former der er for store til med en rimelig nøjagtighed at kunne repræsenteres ved bumpmapping, men for små til, at det er praktisk at danne dem som geometri. Metoderne gør det muligt at tegne meget detaljeret geometri, men er til gengæld ikke anvendelige for animerede objekter, endnu.

5 Observationer og hypotese

I dette afsnit vil der først blive redegjort for silhuetters, og dermed projektets, relevans for visualisering af subdivision surfaces. Dernæst vil der blive præsenteret en række observationer, væsentlige for valget af metode. Til sidst præsenteres en hypotese for forbedring af silhuetter ved visualisering af subdivision surfaces.

5.1 Overordnede betragtninger

En subdivision-flade konvergerer, ved rekursiv underinddeling, mod en grænseflade. Den kan således betragtes som bestående af et uendeligt antal flader. Visualisering kan herefter ske ved enten at evaluere fladen parametrisk, via raytracing *[STAM1]*, eller ved at underinddele fladen tilstrækkeligt. For computerskærme er en mulighed at underinddele fladen til subpixel-niveau. Med subpixel-niveau menes, at der underinddeles så meget, at hver pixel på skærmen, der indeholder en del af objektet, indeholder mindst en vertex. Dette vil dog give for mange trekanter til at nuværende hardware kan visualisere modellen i realtid. Er interaktion med fladerne et mål, er det således en nødvendighed, at danne en tilnærmelse til grænsefladerne.

I forprojektet til dette projekt blev det vist, hvordan lysberegningerne kan bruges til at forbedre den visuelle approksimation til grænsefladen. Det blev her bemærket, at det væsentligste problem ved visualiseringen var at silhuetten fremstod kantet, mens lysberegningerne på det indre af modellen gav en god illusion af en blød flade. I *[BIEDERMAN]* gennemgås en teori for hvordan det menneskelige syns-system fungerer. Det nævnes her, at silhuetter er en så kraftig indikator for den rumlige form af et objekt, at deres effekt overskygger belysningen på en flade *[WITKIN]*. Dette underbygger det observerede problem, hvor illusionen af bløde flader brydes, når silhuetten fremstår kantet. I *[AZUMA]* bemærkes tilsvarende, at præcisionen ved modelsimplificering i den indre del af en model er mindre væsentlig, så længe silhuetten fremstår korrekt.

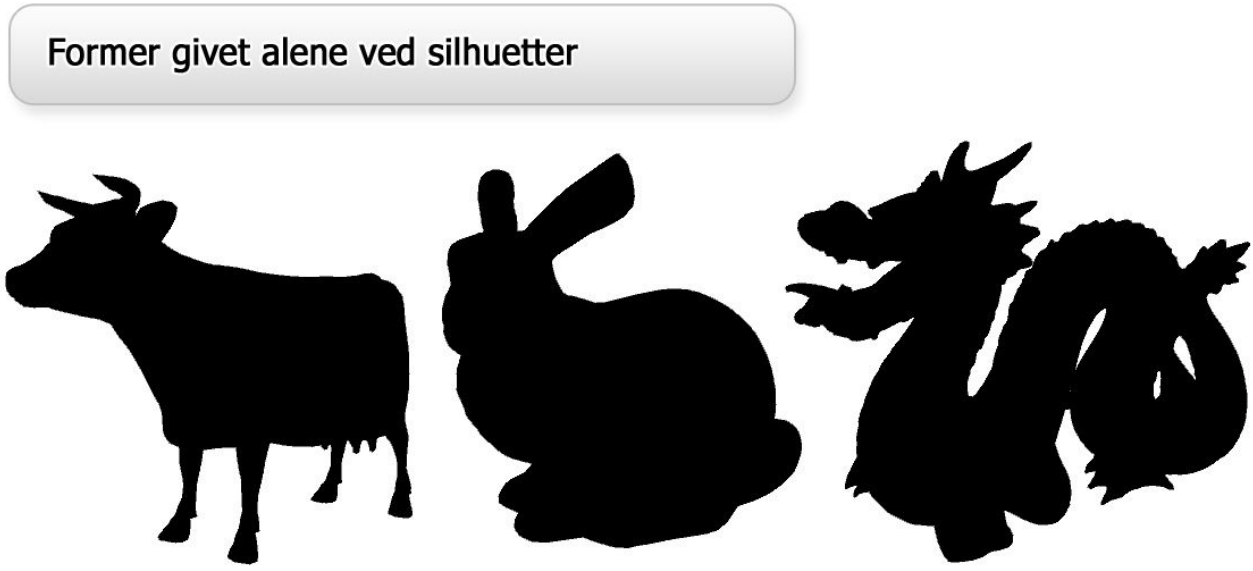


Illustration 5.1:
Silhuetten er en meget stærk, visuel indikator for formen af et objekt – her eksemplificeret ved kendte cgi-silhuetter fra dyreriget. Kan du finde koen?

Det er altså væsentligt at forbedre silhuetten af modellen, hvis vi skal konstruere en overbevisende

visualisering af en subdivision flade. I det foregående afsnit blev gennemgået en række metoder til forbedring af silhuetter. Metoderne kan som nævnt overordnet deles op i to typer: Geometriske metoder, der direkte ændrer geometrien af objekterne og efterfølgende tegner den, og "image-space" metoder, der ved at gemme forskellige data kan korrigere renderingen af objektet.

Problemet med de geometriske metoder er, at de som regel baserer sig på, at der ingen begrænsninger er for den resulterende geometri. Adaptive metoder der forsøger at mindske mængden af geometri, ender alligevel ofte med mere data end det er praktisk at rendere i realtid. Således giver de fleste klassiske adaptive subdivision-metoder en geometrisk detaljegrad der, kombineret med tiden det tager at beregne dem og det tilhørende bogholderi, gør dem uanvendelige i de fleste reelle applikationer. Yderligere giver de kun lille reel mulighed for at styre detaljegraden for silhuetten specifikt, idet underinddelingen af fladen propagerer ud til resten af modellen.

På trods af de præsenterede image-space metoders forskellighed, er det muligt at kategorisere dem i en række hovedgrupper. En tilgang til visualisering af komplekse flader er, at gemme øjepunktsafhængig data fra mange synsvinkler, og udvælge de rigtige data ved rendering. En afart af dette betegnes BDTF, Bi-Directional Texture Functions *[DANA, WANG]*, og går ud på at gemme udseendet af en overflade for forskellige lys-betingelser og anskuelsesretninger. Den tidligere nævnte metode, "Silhuet-mapping", anvender denne metode til lyssætning af en model, og udvider metoden til også at gemme silhuetten for hver synsretning. VDM/GDM tager også udgangspunkt i denne fremgangsmåde, men gemmer yderligere information om geometrien for modellen der renderes. Distancefield-tracing gemmer omvendt kun data om et objekts geometri, og renderer en model ved hurtigt at finde skæringspunkter med denne. Disse metoder har en tendens til at gøre den visualiserede model fuldstændigt statisk. Idet mængden af data, og den tilhørende pre-processering per model er ganske omfangsrig, er ændringer af modellen ikke tilnærmelsesvist interaktive.

En anden tilgang til visualisering af detaljerede flader, er at gemme lidt mindre data, og foretage en mere udførlig søgning i disse data. Denne fremgangsmåde anvendes eksempelvis i parallax-mapping og relief-mapping, der gemmer mere beskedne mængder data, men til gengæld foretager flere beregninger ved rendering. Fælles for disse er at animation er mulig, men at objektets silhuet ikke forbedres. Udvidelsen af relief-mapping til at tage højde for krumningen af fladen gør silhuetkorrektion muligt, men bringer samtidig metoden over i kategorien af svært-animérbare metoder.

Viewdependent progressive meshes og silhuette-clipping er en form for hybrid imellem image-space og geometriske metoder. Disse gemmer en masse information om geometrien, og gennemsøger denne inden rendering, hvorved en geometrisk model hurtigt kan konstrueres og efterfølgende renderes. Også disse resulterer i fuldstændigt statiske visualiseringer.

For subdivision flader har den primære tilgang til at forbedre silhuetten været, at foretage non-uniform underinddeling ud fra øjevektorens vinkel med normalen. En ikke tidligere udforsket mulighed er, at fokusere på at finde den eksakte silhuet, og herved helt undgå underinddeling af fladen. Subdivision flader er matematiske af natur, og besidder som sådan visse analytiske egenskaber. I *[STAM, ZORIN2]* vises hvordan fladerne kan parameteriseres og evalueres eksakt for alle dele af fladen. Dette giver mulighed for at triangulere fladerne helt uafhængigt af den oprindelige struktur i kontrolmodellen. Udover positioner, kan også de afledede af fladen beregnes, såvel normaler som krumning. Idet en kontur er givet ved punkter på fladen hvor normalen står

vinkelret på øjevektoren, kan kontursøgningen betragtes som en nulpunktssøgning henover fladen. Da det er muligt at beregne også de afledte til normalen, kan traditionelle algoritmer så som Newtons metode til nulpunktssøgning anvendes *[LOAN]*. Selv med gode algoritmer til optimering, er det dog ikke sandsynligt, at dette er en operation der kan udføres til fornuftig præcision. Givet at konturkurverne skal genberegnes hver gang øjepunktet flyttes i forhold til en model, eller denne ændrer sig, skal silhuetsøgningen ske meget hurtigt for at give mening til brug i realtidsvisualisering.

5.2 Observationer om silhuetter

De følgende afsnit beskriver essentielle observationer, der ligger til grund for de senere algoritmevalg. Det er ikke umiddelbart indlysende hvordan en silhuet bedst følges rundt på en subdivision flade, og der ligger en række fundamentale observationer til grund for den valgte metode.

En lokaliseret vektor siges at pege imod et punkt hvis prikproduktet imellem vektoren og en vektor fra punktet til vektorens udgangspunkt er positivt. En vektor siges at pege væk fra et punkt, hvis dette prikprodukt er negativt.

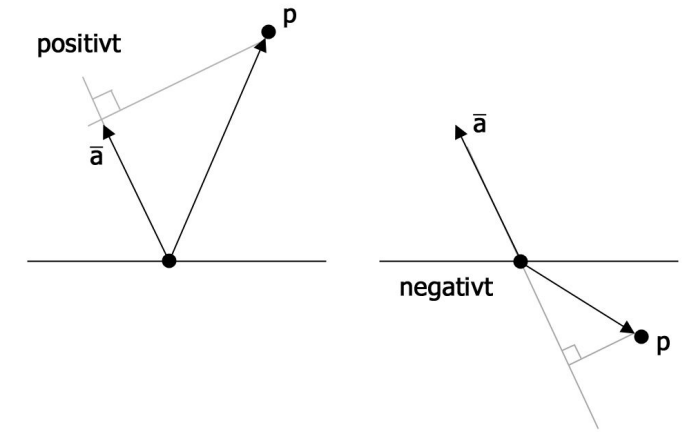


Illustration 5.2

For polygonmodeller er der flere muligheder for at definere hvornår en kant ligger på silhuetten, i forhold til et givet øjepunkt.

1. Kantens flade-naboer har normaler der peger i hver sin retning, i forhold til øjepunktet.
2. Normalerne til fladen, i kanternes endepunkter, peger i hver sin retning i forhold til øjepunktet.

Der er en grundlæggende forskel på disse to definitioner. Den første ser udelukkende på en models geometri, i form af den trianguleredе models fladenormaler, imens den anden ser på de angivne normaler per vertex. Normaler til polygonmodeller angives i computergrafik, som oftest separat fra modellens geometri. Dette gøres for at give illusionen af, at fladen har visse karakteristika der ikke ligger i geometrien, f.eks. at fladen er blød og rund. Dette efterlignes eksempelvis ved, per vertex, at angive normalen som et gennemsnit af nabofladernes normaler. En mulighed der følger er, at angive forskellige normaler til en vertex, afhængigt af hvilken flade den indgår i. Dette giver mulighed for, at tilføje illusion om skarpe kanter til en ellers blød flade, idet lyset vil udvise en diskontinuitet hvor normalen gør det. Der kan således siges at være to forskellige repræsentationer af en model: En der er givet ved den rå geometri, og en, der er en slags ”efterlignet flade”, som er den flade vi forsøger

at danne et billede af. Kigger vi på en lavpolygon subdivision model, hvor de angivne normaler er grænsenormalerne til fladen, er denne ”efterlignede flade” grænsefladen.

Antages det at normaler og fladen har en fornuftig sammenhæng, er forskellen på de ovenstående to definitioner, at den første finder de kanter der rent faktisk udgør silhuetten i en polygon-model (illustration 5.3) imens den anden finder kanter på tværs af silhuetten, se illustration 5.4. Ser vi på den første definition, vil silhuetten til polygonaliserede modeller altid udgøres af en eller flere følger af kanter, der danner lukkede cykler *[HALL]*, som på illustration 5.3. Vælges definitionen der kigger på normaler i endepunkter til en kant, vil der altid dannes lukkede cyker for trianguleredе modeller, med følgende argument: En trekant identificeres som liggende på silhuetten hvis den har normaler der vender i hver sin retning i forhold til øjevektoren. Idet der kun er to muligheder, imod øjepunktet eller væk fra øjepunktet, vil to af en silhuettrekants hjørnepunkter altid vende i samme retning. Samme argument gælder for en nabotrekant der deler silhuetkanten, hvorfor den tilsvarende må have to silhuetkanter. Føres dette argument til ende betyder dette, at følgerne af kanter, for modeller med et endeligt antal kanter, vil danne sammenhængende, lukkede konturer rundt i en model.

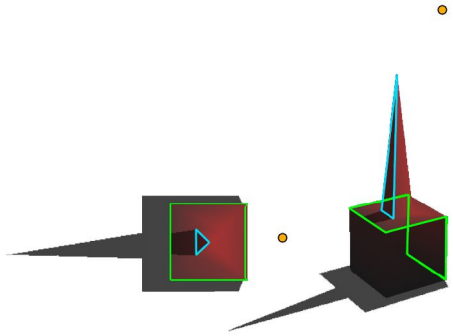


Illustration 5.3:
Det ses at der dannes to konturer på objektet, og at de samlet giver objektets silhuet. Her kigges alene på fladers geometriske normaler.
Billedet er lånt og let modificeret fra [HALL].

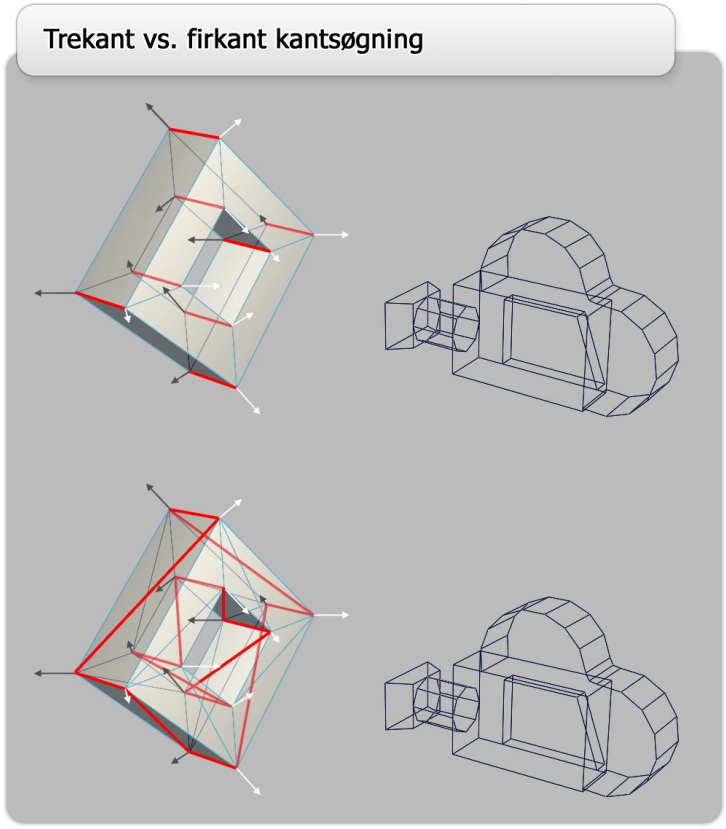


Illustration 5.4:
Øverst søges efter silhuetkanter i model bestående af firkanter – kanterne hænger ikke sammen. Nederst søges i en trianguleret model – alle konturer består af cykler.

Det skal hele tiden erindres, at vi behandler silhuetter i forhold til en perspektivisk projektion. Det betyder at der er et par antagelser vi ikke kan gøre:

- Vi kan generelt *ikke* lade øjevektoren være en vektor fra øjet til øjets fokuspunkt.
- Vi kan *ikke* antage at normalerne vil ligge i et plan, se illustration 5.5.

Modsat ortografiske projektioner, kan to normaler i hver sin side af view-frustum således være vind-skæve, selvom de begge står vinkelret på vektoren til øjet. På grund af den perspektiviske projektion, vil øjevektoren i to punkter generelt ikke vil være ens. Det bemærkes yderligere, at konturpunkterne heller ikke generelt ligger i et plan.

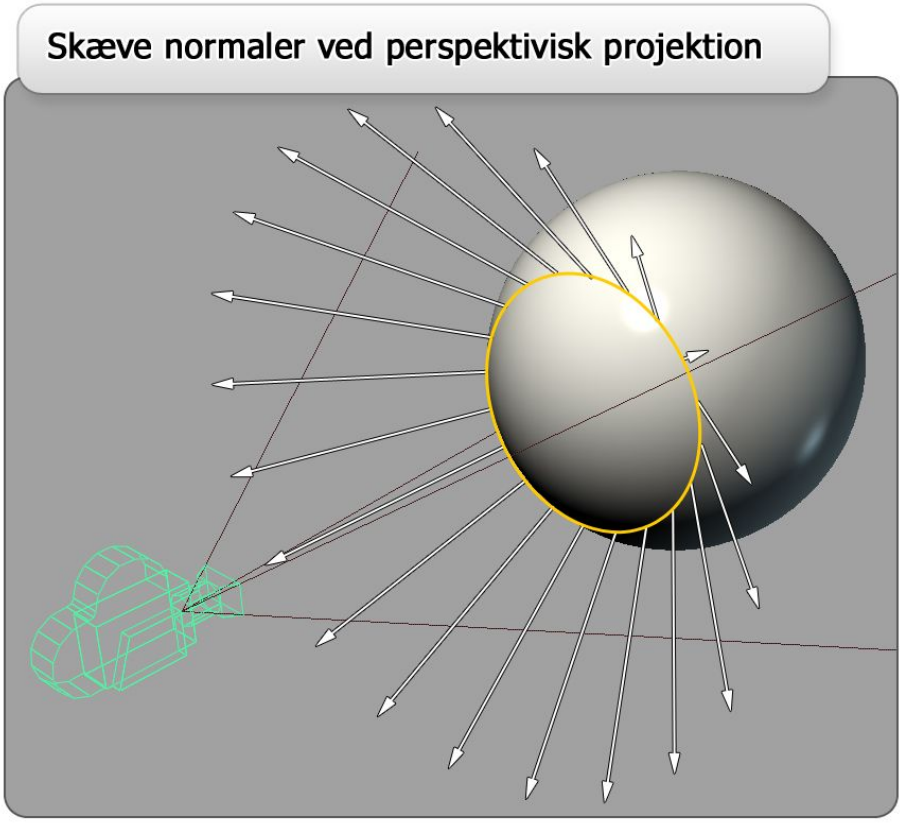
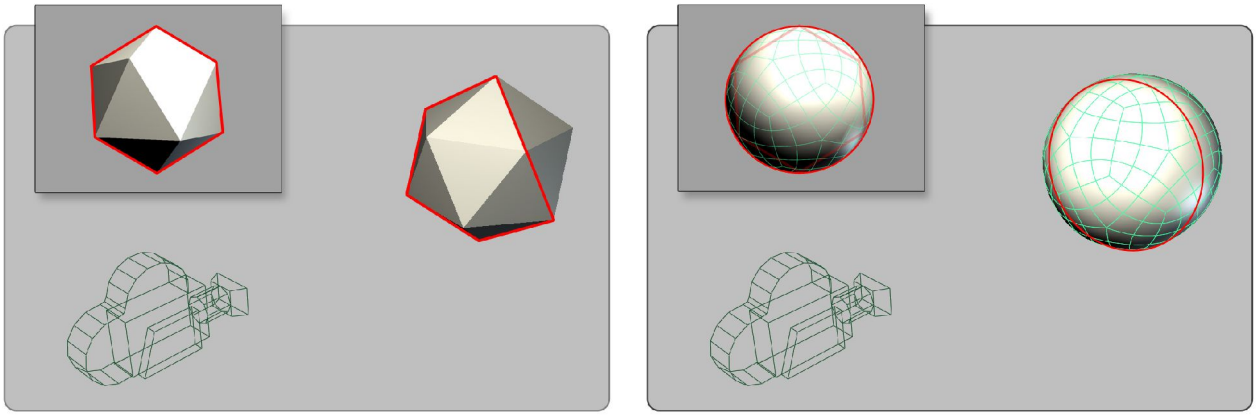


Illustration 5.5:
Bemærk at normalerne til kuglen, fra fod til spids, her danner en stump kegle. De ligger således ikke i et plan, og kan dermed generelt være vindskæve.

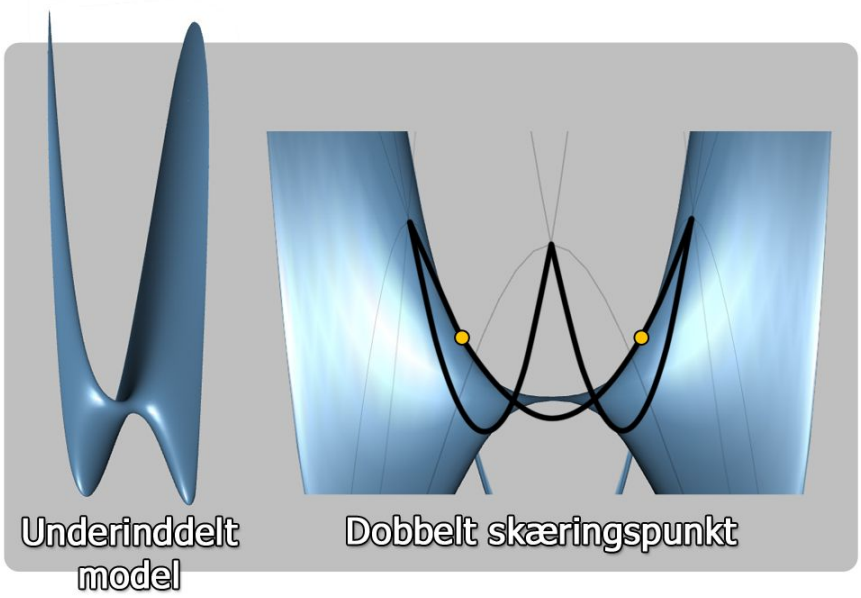
5.3 Om silhuetter til subdivision surfaces

Silhuetten for subdivision surfaces udgøres ikke af kanterne i den oprindelige model. Betragtes en trekant på kontrolmodellen for en kugle, vil ethvert subdividet punkt ligge forskudt udad i forhold til den flade, lineære trekant. Dette betyder at de subdividede punkter kan indgå i silhuetten for objektet. En silhuet på en subdivision-flade udgøres således af en række punkter tværs over trekanterne fra den oprindelige kontrolpolygon – ikke af kanterne i kontrolpolygonen. Dette skal ses i relation til gængse polygon-modeller, hvor silhuetten altid udgøres af kanterne i modellen.



Silhuetten til en vilkårlig flade vil danne en eller flere lukkede kurver der løber rundt langs fladen. For en kubisk B-spline flade af grad $m \times n$, vil en ret linie i parameterrum, der ikke løber langs parameterlinierne i fladen, generelt have grad $m+n$ *[GRAVESEN]*. En lignende regel er ikke fundet for subdivision flader, men det er sandsynligt, at det tilsvarende er nødvendigt at approksimere en kurve på en kubisk flade, med en kurve der har højere grad end tre.

Nu betragtes punkter på en subdividet flade, og vi ser på grænsepositionen for kanterne i kontrolmodellen. Ud fra ovenstående betragtninger ses det, at silhuetkurven vil skære kanterne fra den oprindelige kontrolmodel, skubbet til grænsefladen. Da vi ikke kender graden af silhuetkurven, er det ikke umiddelbart muligt at sige hvor mange gange den kan skære en af disse kanter fra den oprindelige kontrolpolygon. På figur 5.6 ses en situation hvor en kant fra kontrolpolygonen, efter subdivision, skæres to gange af silhuetten. I praksis er der ikke fundet eksempler på flere end to skæringspunkter for en kant.



Figur 5.6:
På billedet ses en situation hvor silhuetten skærer en kant to gange.

5.4 Betragtninger omkring tabelbaseret subdivision

Vi ved fra kapitel 3, at det er muligt for en given model, at finde et tilstrækkeligt sæt koefficienter for beregning af alle subdividede punkter ned til et givet niveau. Da punkterne på en enkelt kant er en delmængde af punkterne i trekanten, er det således også muligt at beregne disse punkter. Restriktionen til kun at betragte punkterne på kanten gør, at beregningen kan foretages hurtigere, idet støtten er mindre, og færre punkter således skal vægtes for at beregne et nyt punkt. Yderligere er de nødvendige tabeller mindre, idet støtten er mindre, og der kun skal gemmes vægte for punkter på kanten.

Tabelbaseret subdivision giver også mulighed for beregning af normaler til fladen. Det er således muligt at vurdere om silhuetten skærer kurven imellem to punkter på kanten – nemlig hvis normalerne til fladen i de to endepunkter peger i hver sin retning. Dette giver mulighed for hurtigt at finde silhuetpunkter på kanten, idet punkter og normaler hurtigt kan udregnes, og den efterfølgende søgning begrænser sig til en triviel endimensionel søgning.

5.5 Om hardware

Dette afsnit indeholder betragtninger omkring det nuværende niveau af hardware. Vi forholder os til PC-hardware der eksisterer i dag, og anvendes på forbrugerniveau. I det følgende vil de blive beskrevet, en række af de hardware-features der er relevante ihht. visualisering af subdivision surfaces.

5.5.1 Shadere

Den vigtigste tilføjelse til hardware indenfor de sidste par år, er muligheden for at programmere dele

af pipelinen. Det er blevet muligt fuldt ud at kontrollere transformationer, tekstur- og lysberegninger, hvad enten det sker per vertex eller per pixel. Programmer der afvikles på grafikkortet betegnes generelt ”shadere”. En shader fungerer som en erstatning for en del af grafikkortets pipeline. Det betyder at de funktioner der hidtil blev udført automatisk, nu skal udføres manuelt. Grundet hardware-mæssige hensyn til parallelitet, er det yderligere ikke muligt, at gemme information fra en kørsel til den næste – data sendes en vej igennem pipelinen og smides herefter væk. Det er heller ikke muligt at tilgå resultater på tværs af en kørsel, f.eks. at se på en vertex naboers nye position.

En nyligt tilføjet mulighed er, at vertex-shadere kan tilgå teksturhukommelse. Idet det længe har været muligt at rendere til en tekstur, giver dette mulighed for at ændre en vertex's position i forhold til en tidligere rendering. Idet floatingpoint-teksturer ligeledes er blevet almindelige, er der på denne måde givet mulighed for en type feedback i pipelinen. Et problem med tekstur-tilgang fra vertex-shadere er, at deres hastighed stadig er uforudsigelig, og langsom i sammenligning med pixelshadere.

Det er ikke alle dele af pipelinen der kan modificeres. I pixelshaderen er det muligt at programmere såvel farven og gennemsigtigheden af en pixel, samt dens dybde. Det er dog ikke muligt frit at ændre måden den resulterende pixel blendes. Hvis dette er nødvendigt, foretages normalt flere renderinger, der efterfølgende kombineres ved hjælp af multi-texturing. Det eneste sted der er adgang til primitiv information, er under culling/rasterisering, se illustration 5.7. Dannelsen af primitiver er ligeledes fastlåst – det er således ikke muligt at bestemme hvilke vertices der hænger sammen i polygoner. En væsentlig observation er, at der ingen steder er tilgang til generel naboinformation for punkter eller polygoner. Er denne type information nødvendig, skal den således gemmes manuelt. Typisk sker dette ved at gemme fladekonstanter i alle vertices i fladen, mens vertex-konstanter kan gemmes direkte per vertex.

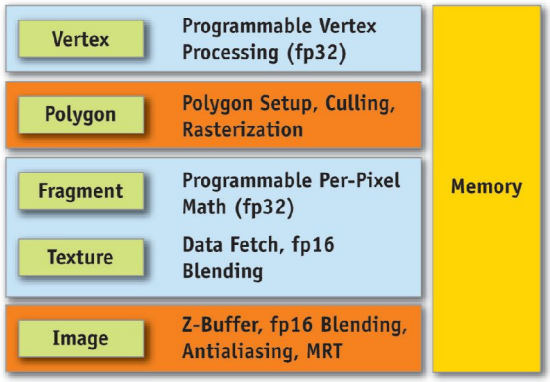


Illustration 5.7:
Billedet er lånt fra [NV40]. Blå felter er programmerbare, røde er ikke. Det ses at det er muligt at tilgå hukommelsen fra alle dele af pipelinen.

Et problem med vertex-shadere er, at alle beregninger smides væk så snart geometrien er ændret. Det betyder, at selvom data kun sjældent ændres, skal de enten beregnes hver gang billedet tegnes, eller alternativt ændres på CPUen og uploades igen. En generel løsning på dette problem, ville være at give mulighed for at lade grafikkortet skrive tilbage i vertex-data. I praksis er dette implementeret

i OpenGL via såkaldte Pixel-Buffer-Objects (PBO), der gør det muligt at anvende buffere af en type, som en anden type[23]. Dette giver mulighed for, at en buffer med vertex-data kan anvendes som tekstur, og at der tilsvarende kan kopieres (floating-point) pixels til vertexdata. I praksis betyder det, at det bliver muligt at ændre i de vertex-data der ligger på kortet, uden at det skal sendes over CPU'en. Metoden til at opdatere geometri er ikke triviel, men ligger ret tæt op af de metoder der anvendes i GPGPU, se afsnit 5.5.2.

Anvendelsen af PBO giver mulighed for, at vertex-data kan opdateres når de ændres, i stedet for når modellen tegnes. Ændres data således kun sjældent, opnås en væsentlig besparelse. En triviel anvendelse er "rigtig" per-vertex displacement, hvor hver vertex forskydes langs sin normal i forhold til en værdi gemt i en tekstur. I sammenhæng med subdivision surfaces betyder det, at subdivision-interpolationen af data, kan foretages på grafikkortet *[SCHRÖDER5]*, idet vertex naborelationer kan lægges i en tekstur på linie med andre vertex-data. Da det ikke er muligt at danne punkter i en shader, er det nødvendig først at allokere plads til alle positioner, samt at foretage den topologiske underinddeling af modellen på grafikkortet. Herefter køres en shader der opdaterer positionerne i forhold til de gemte naborelationer. Tidligere var de fleste grafikkort i stand til at evaluere og triangulere NURBS-flader ud fra deres kontrolpolygoner. Denne feature blev dog fjernet igen, da den reelt ikke blev anvendt. På de fleste ATi-kort består PN-triangles stadig, og Matrox giver mulighed for at tesselere geometri til brug med displacement mapping, men generelt er det ikke længere muligt at skabe trekanter på grafikkortet.

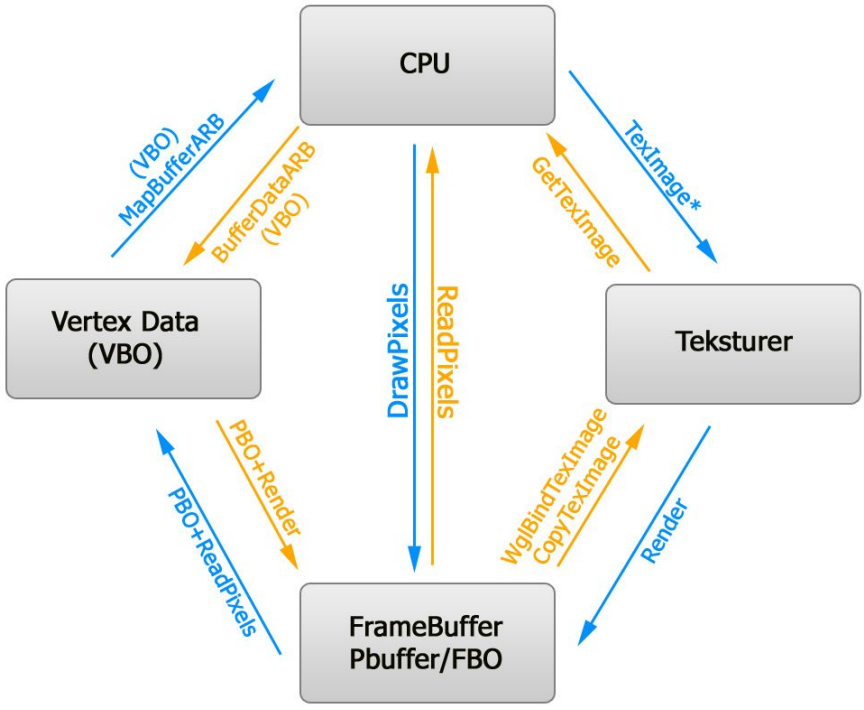


Illustration 5.8:
Bortset fra vertex data, kan præcisionen alle steder vælges til 8/16/32bit.
Diagrammet er ikke udtømmende.

23 Dette svarer til sædvanlig typecasting. Der er ingen mulighed for at foretage konvertering imellem forskellige formater, eller floating-point præcision.

5.5.2 GPGPU

Efterhånden som grafikkort er blevet mere programmérbare, er begrebet GPGPU[24] dukket op *[GPGPU.ORG]*. Den nyvundne fleksibilitet på grafikkort, gør dem i stand til at udføre mere generelle opgaver end de grafik-specifikke funktioner de tidligere varetog. Udover blot at være "endnu en processor" der kan anvendes til beregninger, er GPU'er specialiserede til floatingpoint- og vektor-beregninger. Dette betyder, at nogle opgaver kan foretages meget hurtigt på en GPU i forhold til en CPU. Yderligere går udviklingen for grafikkort hurtigere, hvilket betyder at hastighedsforskellen imellem de to områder kun bliver større i fremtiden – hvilket samlet set giver gode fremtidsudsigter for GPU-baserede algoritmer. På trods af at programmering af grafikkort er et forholdsvist nyt område, er det allerede anvendt på ganske mange områder – dog hovedsageligt centreret omkring computergrafik. I flæng kan nævnes fluid-, partikel og stof- (cloth) simulationer, raytracing samt generelle matrix-beregninger.

Trenden de sidste par år, er gået i retning mod forholdsvist større regnekraft per pixel, sammenlignet med den afsat til vertex-shadere. Således udregnes der på nuværende hardware 16 pixels af gangen *[NV40]*, imens vertex-programmer er begrænset til 6 parallelle beregninger. Ønsker man således den maksimale regnekraft fra grafikkortet, bør man i videst muligt omfang udnytte fragment-programmer. En yderligere gevinst ved dette er, at tekstur-tilgang er hurtigere for fragment-programmer. Generelt udnyttes dette i GPGPU-sammenhænge, ved at tegne en firkant påført en række teksturer med data, der fylder hele "skærmen". Det bør her bemærkes, at en væsentlig udfordring er, at det ikke er umiddelbart muligt at skrive til flere pixels. Dog kan en pixelshader skrive til flere buffere ("Multiple-Render-Targets" MRT), hvilket i visse henseender afhjælper dette problem.

Bussen der forbinder CPU med GPU er en væsentligt begrænsende faktor. Hastighedsforbedringerne på denne bus, senest i form af "PCI-express", følger på ingen måde de øvrige fremskridt i hastighed for GPU og grafik-hukommelse. Dette dikterer en væsentlig restriktion i henhold til algoritmer hvor hastighed er en afgørende faktor og bussen således ikke må udgøre en flaskehals. Generelt bør således kun små mængder data overføres i tidskritiske perioder, hvorfor datatilgang bør foretages internt i system-området eller på videokortet. Dette gælder såvel upload af data, som download fra grafikkortet.

Data der anvendes under afvikling af en GPGPU-algoritme, eller visualisering af en model generelt, bør således i videst muligt omfang også beregnes på grafikkortet. Dette resulterer ofte i, at metoder der kan afgrænses til et specifikt område af maskinen foretrækkes, selv om bedre og teoretisk hurtigere algoritmer eksisterer *[MCGUIRE2]*. Herved opnås i praksis en bedre ydeevne, selv om metoden er teoretisk langsommere.

5.6 Hypotese

Målet med dette projekt er at konstruere en metode der forbedrer silhuetten ved visualisering af en subdivision flade. En metode til at gøre dette, er at finde konturkurver på fladen, og forskyde disse til grænsepositionerne for subdivision fladen. Det betyder, at vi skal finde alle punkter på fladen, hvor normalen står vinkelret på øjevektoren. For at begrænse det gennemsøgte område, foretages en

24 GPGPU, "General-Purpose computation on GPUs", GPU står for "Graphics Processing Unit"

søgning alene på kanterne af basis-modellen. Grænseflade-punkter beregnes ved hjælp af tabelbaseret subdivision. For at finde punkter hvor silhuetten skærer kanterne, beregnes normaler til subdivision-fladen langs med kanten, og der foretages en nulpunkts-søgning i disse. Når alle kant-silhuetpunkter er fundet, forbindes de med en spline-kurve der ideelt set tangerer subdivision fladen. På denne måde bliver det muligt at foretage en minimal triangulering af fladen, idet kun trekanter der direkte indgår i silhuetten berøres.

For at gøre den resulterende visualisering mest muligt fleksibel, baseres metoden ikke på pre-generering af et hierarki af trekanter som i *[AZUMA, HOPPE1, HOPPE2]*. I stedet tages der for silhuet-søgningen alene udgangspunkt i kontrolpolygonen til subdivision-fladen. Ved brug af tabelbaseret subdivision, kan punkterne på potentielle silhuetkanter effektivt beregnes, hvorefter der kan foretages en søgning henover kanterne.

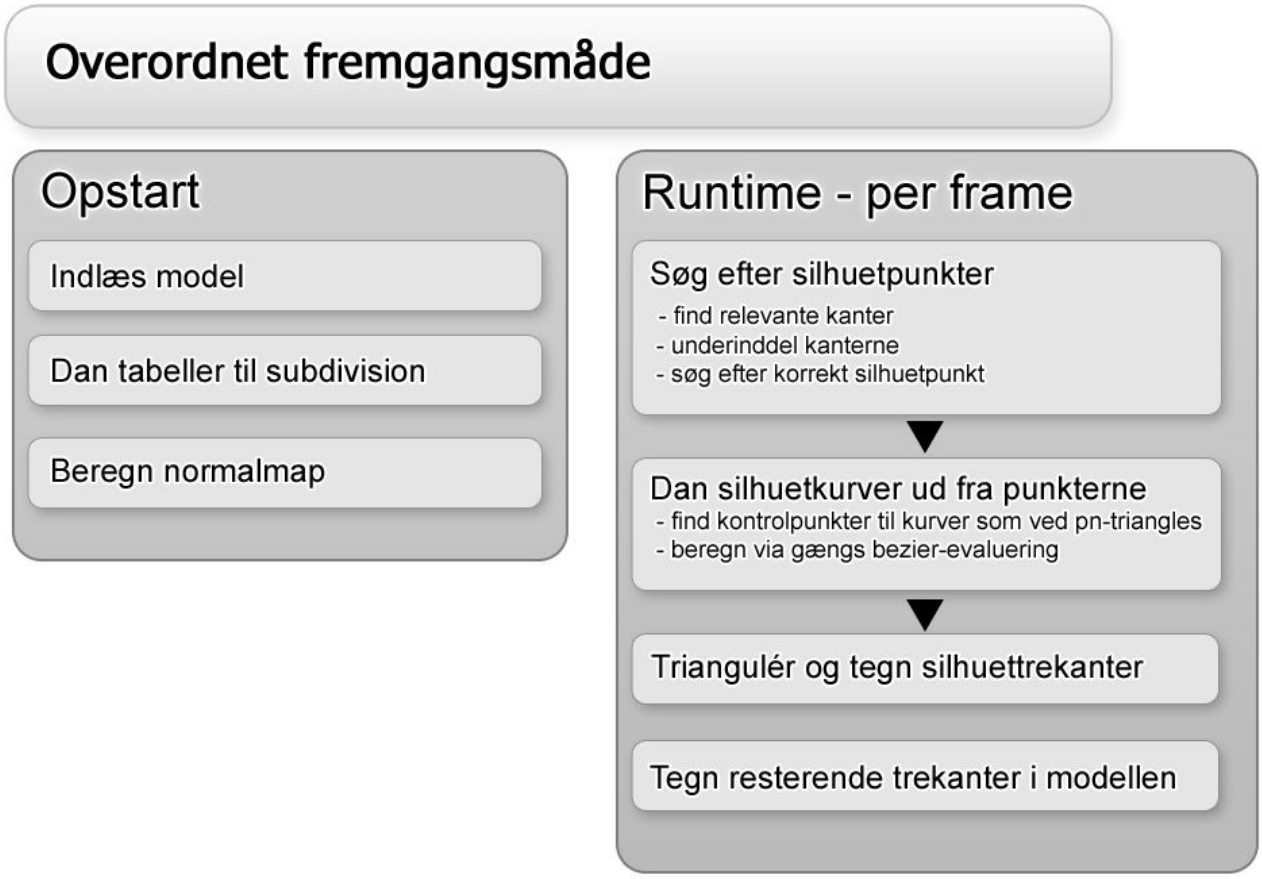


Illustration 5.9:
Det bemærkes, at tabellerne ikke generelt er modelafhængige, men her begrænses til de i modellen eksisterende valenser. Se afsnit 6.

Den overordnede fremgangsmåde minder om silhuetclipping (afsnit 4.1.6)**.** Begge baserer sig på at finde silhuetten til et givet detaljeret objekt, ved at søge efter kanter der vender hver sin vej i forhold til øjevektoren. Den primære forskel ligger i at silhuette-clipping tager udgangspunkt i en højdetaljeret model, hvor den her fremsatte hypotese tager udgangspunkt i subdivision flader, og beregner de nødvendige data under søgningen. Dette betyder at metoden gemmer mindre data, og at

de resulterende modeller bliver mindre statiske. Idet søgningen og dannelsen af silhuet-kanter sker uden model-specifik pre-beregning, er det muligt at ændre modellen uden at det har nogen indflydelse på metodens hastighed.

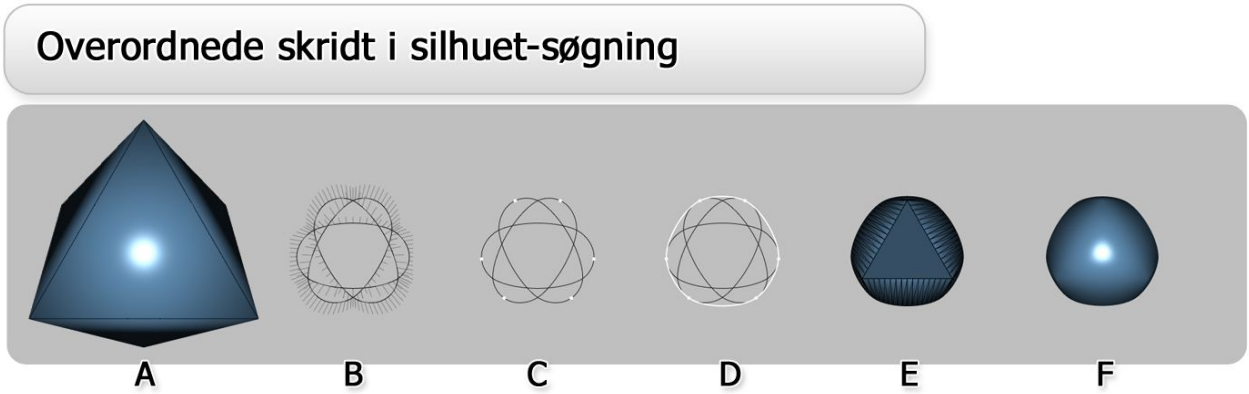


Illustration 5.10:
Med risiko for at tage forskud på glæderne, ses her billeder af den overordnede fremgangsmåde.
A viser den grove model der tages udgangspunkt i.
B viser modellen, med kun kanterne underinddelt, og normalerne til grænsefladen i disse punkter.
C viser de fundne silhuetpunkter på de neddelte kanter (men ikke de fundne normaler).
D viser silhuetkurven, dannet ud fra de fundne silhuetpunkter og normaler.
E viser den lokale triangulering af området omkring silhuetten.
F viser fladen påført et normalmap.

5.6.1 Om anvendelsen af normalmapping til shading

Anvendelsen af polygonmodeller til visualisering af bløde flader er altid en approksimation. Inden for computergrafik angives normalen til vertices ofte separat fra modellens geometri, for på denne måde at kunne tilføje information til lysberegningen der ikke findes direkte i fladen. Normalmaps generaliserer denne tankegang til at fungere per texel i en model, frem for per vertex. Forprojektet gav gode resultater med shading af fladerne ved hjælp af normalmapping, så denne fremgangsmåde vil også blive anvendt her. Dette giver de samme muligheder og begrænsninger på metoden som beskrevet der – primært at per vertex-animation vil kræve genberegning af normalmappet.

Anvendelsen af subdivision-normalmapping begrænser muligheden for fri animation af objekterne. Idet vi gemmer normalerne for et objekt, og disse afhænger direkte af den subdividede geometri, bliver det nødvendigt at genberegne dele af objektet der er ændret. Som beskrevet i forprojektet er det dog muligt at foretage de mest almindelige typer animation på objekterne. Da de grundlæggende modeller som regel vil indeholde få polygoner, er den bedste fremgangsmåde at foretage denne animation på CPUen, inden søgningen efter silhuetten. Umiddelbart interessant virker det, at approksimere normalen med en kvadratisk funktion. Idet vi kan beregne normaler i indre punkter på en trekant, er det muligt at danne eksakte koefficienter for et kvadratisk polynomium. Da de fleste subdivision metoder giver kubiske flader, giver det mening at foretage denne tilnærmelse. Dog vil det betyde, at skarpe kanter og lignende subdivision-specifikke features ikke umiddelbart kan indgå i metoden. Med udgangspunkt i ligheden mellem per-pixel belysning og normalmaps, er det dog sandsynligt at forskellen i mange tilfælde vil være negligérbar.

Det er ikke direkte muligt at anvende korrigerende algoritmer så som relief- eller parallax-mapping sammen med silhuette-subdivision, da det kræver brugen af et højdekort. Dette højdekort vil, hvis metoden kombineres med silhuetforskydelse, så at sige blive forskudt sammen med fladen. Dette er et generelt problem med metoder der ændrer modellen geometrisk, idet de fleste metoder til korrektion af normalmapping antager en statisk, eller begrænset animeret, model. En mulighed er, at gemme den forskydelse af fladen der sker ved silhuette-subdivision. Dette ville give mulighed for at anvende et højdekort, og en efterfølgende søgning i dette.

En anden mulighed er kun at anvende parallakse-korrektionen i det indre af fladen, og foretage en blød udfasning mod silhuetten. Silhuet-korrektionen forbedrer tilnærmelsen til fladen nær silhuetten sammenlignet med den grove polygon – til gengæld vil parallakse-fejlen generelt være større nær silhuetten. Anvendes denne metode vil en del af fejlen således bestå nær silhuetten.

6 Implementering

I dette afsnit vil blive beskrevet den anvendte fremgangsmåde til implementering af en prototype på hypotesen om underinddeling af silhuetter. De specifikke optimeringer og algoritmer der er lavet i forbindelse med implementering af metoden vil yderligere blive beskrevet. Afsnittene her vil flere steder hænge tæt sammen med den tidligere beskrevne teori, og det anbefales således at læseren refererer til disse afsnit for uddybende teoretiske forklaringer. I det følgende gennemgås først det grundlæggende framework, og derefter den specifikke implementering af den udviklede metode til silhuetsøgning.

De steder hvor der præsenteres diagrammer for klassestrukturer, angiver pile nedarvning fra det der peges på, mens cirkler angiver instanser af objekter.

6.1 Geometriske strukturer

I projektet anvendes en kant-baseret datastruktur til beskrivelse af de topologiske relationer i en model. De fleste sædvanlige subdivision-metoder kan foretages på enklere datastrukturer, som f.eks. et indexed face-set. Der er dog en række fordele ved at bruge en udvidet datastruktur. Kantbaserede datastrukturer er til stor hjælp når avancerede søgninger i modeller foretages. En fleksibel datastruktur er ligeledes en støtte under udviklingen af metoden, idet det bliver muligt, hurtigt at prøve forskellige muligheder af. Yderligere fjerner det en usikkerhedsfaktor omkring hvorvidt datastrukturen vil være i stand til at foretage de ønskede operationer, idet langt de fleste ønskelige operationer er mulige.

6.2 Om wavefront-mesh importer

Der er i projektet implementeret en funktion til læsning af filer med modeldata. En ”loader” er i stand til at læse et specifikt filformat, til en struktur der er tæt knyttet til filen. For at koble denne data med en model, konverteres den via en ”factory” til et generelt modelformat, se afsnit 6.2.3.

Formatet der indlæses er Wavefronts *.obj*-format. Dette format gemmes i klartekst, og er kompatibelt med næsten alle programmer til 3D-modellering. Det indeholder information om punkt-positioner, teksturkoordinater, normaler samt sammenhænge i polygoner. Dette giver den størst mulige fleksibilitet i forhold til skarpe normaler og opdelt texturemapping. I importeren antages det, at alle flader er ”ens” - således at hvis en flade har teksturkoordinater har alle flader teksturkoordinater. Formatet giver også mulighed for at specificere transformationsmatricer og angive objekt-sammenhænge. Desuden kan det indeholde NURBS- og andre højniveau flade-definitioner. Disse funktioner er dog ikke implementeret i dette projekt. Materiale-definitioner læses, men anvendes ikke konkret i dette projekt.

En specifikation af obj-formatet kan findes i *[PBOURKE1]*.

6.2.1 Den kantbaserede datastruktur ”Lath”

I prototypen anvendes en kantbaseret datastruktur til den topologiske beskrivelse af modellerne - altså naborelationer punkter og flader imellem. Datastrukturen er baseret på et element der kaldes en

”lath”. En lath er en enhed, der har tre variable: To referencer til andre laths, samt en reference til et geometrisk element, f.eks. en flade eller et punkt. Datastrukturen, der generaliserer tidligere kant-baserede datastrukturer som f.eks. Half-Edges[25], er beskrevet i *[LEGAKIS]*. I dette paper vises alle mulige geometriske strukturer der kan beskrives ved anvendelse af laths. I OpenGL er det ofte nyttigt at kunne traversere flader i en model i en orientering mod uret, hvorfor der er valgt en datastruktur, der anvender links ligeledes orienteret mod uret, se illustration 6.1. En fordel ved at anvende præcis denne udgave af den lath-baserede datastruktur er, at det duale mesh kan konstrueres umiddelbart. Det duale mesh findes blot ved at udskifte det geometriske link, med det duale geometriske link, eksempelvis ved at udskifte alle vertex-links med et link til en face – lath-strukturen forbliver således uændret. Da en del subdivision-metoder anvender det duale mesh, kan dette være en nyttig egenskab, men det udnyttes dog ikke i det implementerede program.

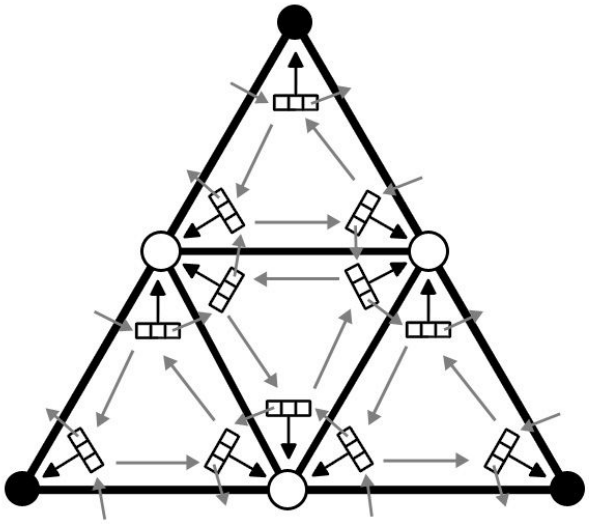


Illustration 6.1
Struktur inden for en neddelt trekant. En lath peger på næste lath rundt om en vertex, samt næste lath rundt i en face

Som strukturen er beskrevet i *[LEGAKIS]*, kan den kun beskrive lukkede 2-mangfoldige objekter. Det betyder at alle kanter skal deles af præcis to fladenaboer, samt, at alle flader der deler en vertex skal indgå i et sammenhængende følge. For åbne flader må kanter godt have kun en nabo. I den foreliggende implementering, er strukturen udvidet til også at kunne beskrive modeller med huller, dog kun huller i forstanden manglende flader – ikke frit definerede huller som indre huller i en trekant, eller over flere trekanter (så som eksempelvis NURBS-trimkurver). Denne udvidelse er nødvendig for senere at kunne foretage neddeling af støtterne til tabelsubdivision, men har også været anvendt i undersøgelsen af subdivision-regler for åbne modeller. Udvidelsen er triviel, og betyder essentielt blot at *counter_clockwise_vertex* for en lath kan være udefineret – hvilket angives med værdien *-1*. Det er stadig et krav at fladen er en mangfoldighed, hvilket betyder at enhver vertex i modellen kun kan indgå i et enkelt hul.

Alle lath-relaterede data ligger i en std::vector, pakket ind i klassen *cLathvec* – også funktioner til at

25 For andre kant-baserede datastrukturer, se f.eks. *[DOBKIN]* for Winged Edge, eller *[KETTNER]* samt *[WEILER]* for en sammenligning af strukturer.

traversere datastrukturen er implementeret i denne klasse. En ”lath” ses således i denne implementering som en slags ”proxy”-objekt *[GAMMA]*, hvor igennem de rigtige data kan tilgås. Et *cLath*-objekt indeholder ingen egentlig data, men kun en reference til en *cLathvec*, samt et index ind i denne vektor. Dette giver et elegant interface til laths, der fjerner de fleste muligheder for at lave fejl – primært fordi det muliggør check for fejl på flere niveauer. Yderligere er det næsten lige så effektivt som pointere, da et *cLath*-objekt kun vil indeholde ganske lidt data, og således er hurtigt at oprette og kopiere rundt.

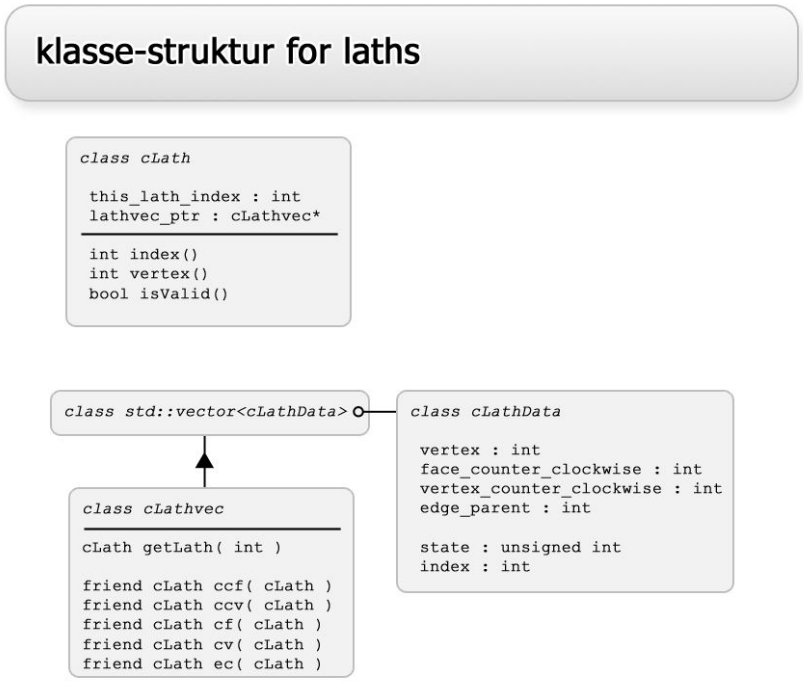


Illustration 6.2:
Operationer til traversering af modellen ligger ikke i nogen klasse, men har fuld adgang til cLathvec.

Selvom alle kanter i en model udgøres af to laths, er det på enkel vis muligt at identificere en kant entydigt. En kant identificeres unikt ved det laveste indeks af de to laths der udgør kanten. Det betyder at der vil være præcis halvt så mange kant-indices som lath-indices, men at disse vil være fordelt over samme interval.

En kantbaseret struktur er bekvem når man vil knytte forskellig data til to sider af en kant. Dette er ofte et ønske, eksempelvis hvis normalerne til en flade ikke er kontinuerte hen over en kant. Anvendes en kantbaseret datastruktur som f.eks. laths, kan normalen blot lægges i lath'en, idet en kant udgøres af to laths, og således kan gemme separate normaler. Konverteres denne struktur til en vertex-baseret struktur, som eksempelvis kræves inden den data kan anvendes med OpenGL, er det nødvendigt at danne unikke vertices. Der laves således flere sæt data for hver ”logisk” vertex, se illustration 6.3. Dette kaldes ofte at ”eksplodere” en model. Denne operation sørger for, at hvert sæt vertex-data kun tilhører en flade. Dette gøres ved at traversere alle laths omkring en vertex, og oprette en ny vertex for hver ny flade der findes. Det bemærkes at kun lath-vertex linket ændres. Idet OpenGL anvendes til at tegne modellen, og denne kræver unikt identificerede punkter, ville denne ”eksplodér”-operation skulle foretages hver gang modellen skulle tegnes, hvis vi gemte data

direkte i laths. For at undgå dette, gemmes alligevel al data i eksploderede vertices, hvor det er nødvendigt, og der tages efterfølgende højde for dette, når der laves topologiske operationer, så som tilføjelse af kanter til modellen.

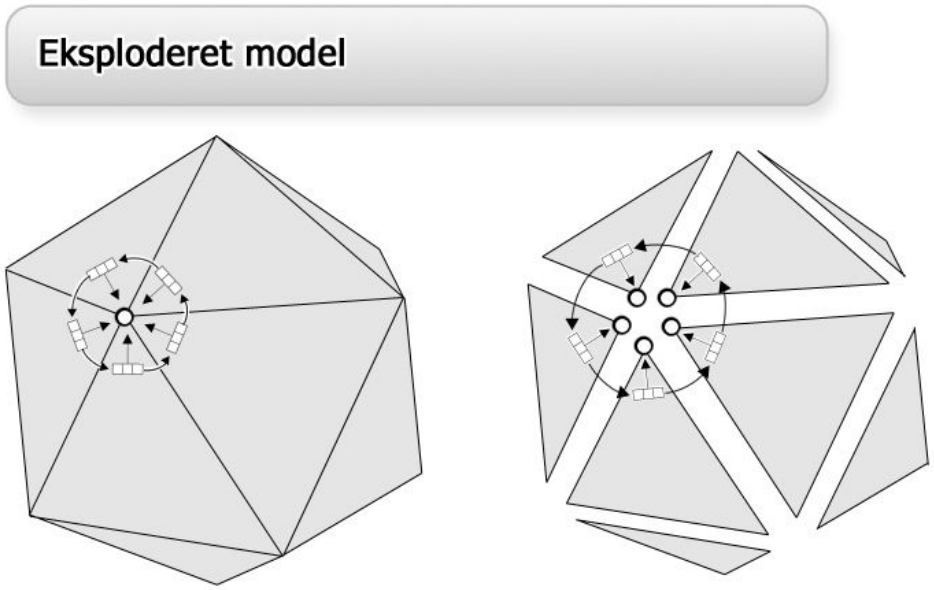


Illustration 6.3:
Det bemærkes at kun vertex-linket ændres. Den topologiske struktur forbliver intakt.

6.2.2 Geometriske operationer og subdivision

For at implementere sædvanlige subdivision-metoder som Loop og Catmull/Clark, er kun to topologiske operationer nødvendige; *split-kant*, og *tilføj-kant*. For at holde styr på nye og gamle punkter, er det nødvendigt for hver lath at markere dens tilstand. For at holde styr på nye og gamle punkter, er der til lath-strukturen tilføjete en *state*-variabel, der kan bruges til at angive hvordan en lath er tilføjet – f.eks. ved hvilken af de to topologiske operationer den er oprettet.

Fremgangsmåden for Loop-subdivision er som følger, idet ”gamle” laths betegner de laths der eksisterede i modellen inden subdivision. For at underinddele modellen udføres overordnet følgende operationer, se illustration 6.4.

- Tilføj et nyt punkt til alle kanter. Dette gøres ved at anvende *split-kant* operationen en gang på hver kant.
- Tilføj kanter imellem nye punkter. Dette sker ved, for alle trekanter i modellen, at anvende *tilføj-kant* operationen fra hvert nye punkt i en trekant, til det næste nye punkt mod uret.

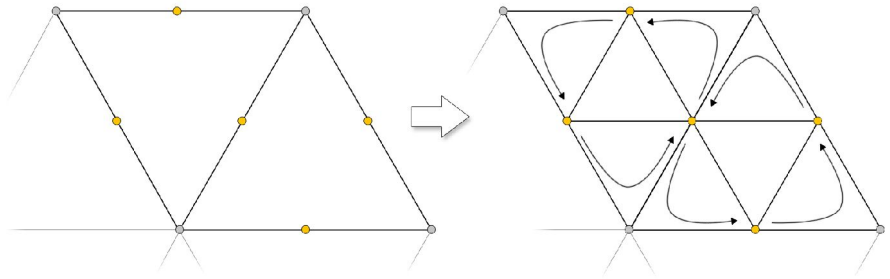


Illustration 6.4:
Underinddeling af trekanter. Modellen behandles ikke i par, men trekant for trekant. Der holdes hele tiden styr på hvilke laths der er behandlet.

Herefter foretages udglatning på punkterne, ved at traversere gamle laths. Idet nye laths tilføjes i slutningen af lath-vektoren, og ingen operationer ændrer på rækkefølgen af laths, kan vi trivielt afgøre om en lath eksisterede i modellen inden subdivision: Antallet af laths inden underinddeling giver skellet imellem den sidste gamle lath, og den første nye. Fordi laths kun dannes ved indsættelsen af nye punkter, vil enhver lath der er dannet ved ovenstående underinddeling, pege på et punkt der ligeledes er indsat ved underinddelingen. Vi kan på denne måde gennemløbe sekvenser af alene nye eller gamle punkter.

Udglatningen foretages således som følger:

- Gennemløb alle nye punkter, og vægt dem ud fra Loops regler. Når et nyt punkt er vægtet, markeres alle laths pegende på dette punkt som afsluttede.
- Gennemløb gamle punkter, og vægt dem i forhold til deres naboer inden underinddelingen.

Det er en nødvendighed, at der tages en kopi af alle ”gamle” positioner, for at kunne foretage Loop-vægtningen. Ligeledes kopieres laths-vektoren inden underinddeling – dette er ikke strengt nødvendigt, men det simplificerer søgningen efter ”gamle” naboer betydeligt.

Udvidelsen til åbne modeller betyder som nævnt, at *counter_clockwise_vertex* linket kan være ugyldigt. Det medfører at der skal foretages et check for, om en given kant ligger på fladens ydergrænse. Ved *tilføj-kant* operationen, kan dette check dog udgås, ved at vælge at lade den indsatte lath pege på den potentielt ikke-eksisterende lath. Dette sker trivielt ved at kopiere linket fra den gamle lath, eksempelvis fra ***L1*** til ***NL1***, se illustration 6.6.

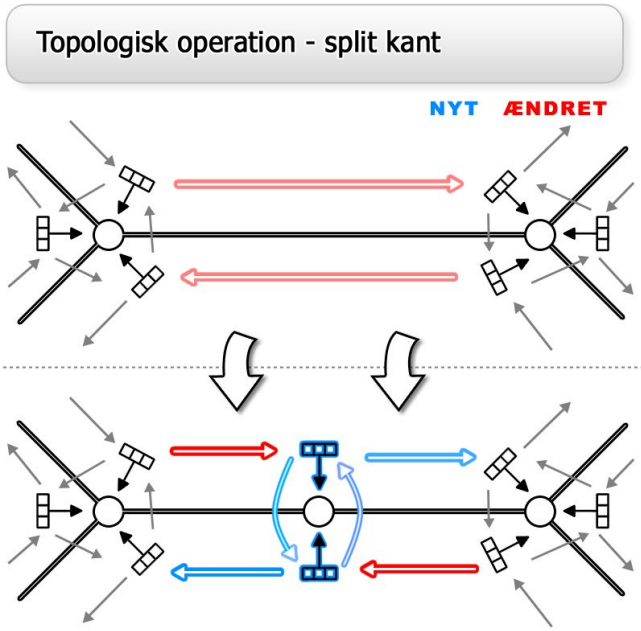


Illustration 6.5:
En kant opdeles ved at tilføje to laths samt en vertex.

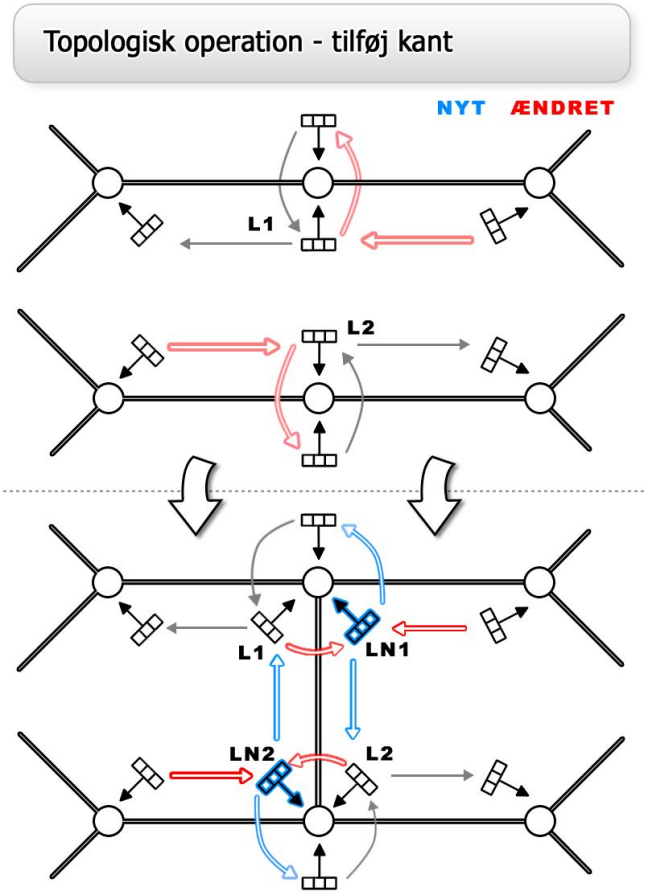


Illustration 6.6:
En kant tilføjes ved at tilføje to laths. Bemærk at intet boundary check er nødvendigt.

I den foreliggende prototype, foregår de fleste operationer direkte på lath-datastrukturen. Denne struktur nødvendiggør en del manuelt bogholderi, der burde samles i bekvemmelighedsfunktioner bygget oven på de eksisterende laths. Et par af disse funktioner er implementeret – eksempelvis er det muligt at foretage en række højniveau forespørgsler på naborelationer. Specifikt er der tilføjet funktioner til at finde alle naboer til et punkt og alle laths der peger på et punkt, samt valensen for et punkt. Denne fremgangsmåde bør dog udvides, og det ville være oplagt at lave et abstraktionslag, der kunne tilføje eller ændre punkter og trekanter, direkte på vertices i stedet for laths.

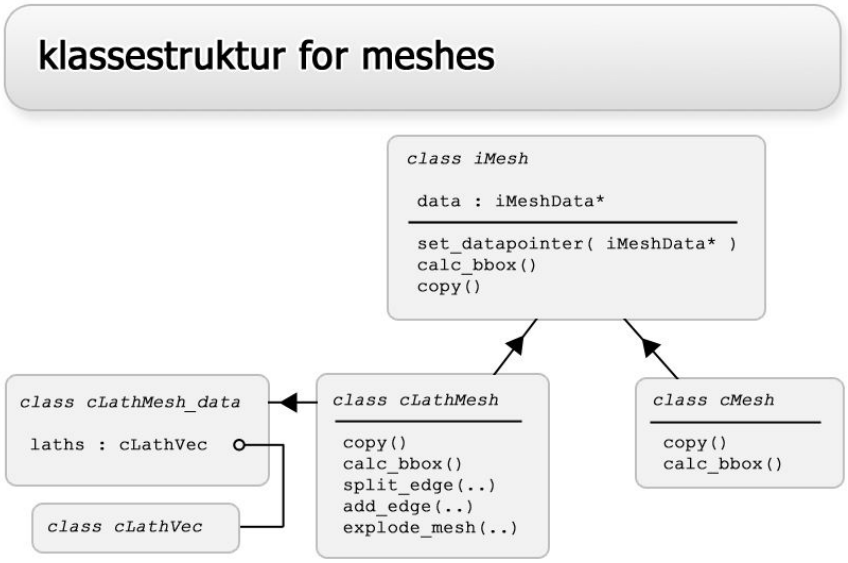


Illustration 6.7:
De eneste krav til meshes er, at de skal kunne kopiere sig selv, og udregne boundingboksen af deres data.

6.2.3 Typer af data i en model

Der differentieres i den implementerede prototype imellem to typer data, i forhold til måden de interpoleres: Lineær data, og subdivision-data. Når en model subdivides, foretages således to separate sæt interpolationer: En der anvender subdivision-regler, og en der foretager sædvanlig lineær interpolation. For udglatning af "gamle" punkter, tildeles lineær interpolation ingen effekt.

Implementeringen af dette baserer sig klasser kun indeholdende data, og funktioner til at kopiere denne data. Der gemmes to lister med sæt af data. Listerne indeholder referencer til arrays af data, samt information om hvor stort et enkelt element er (vec2, vec4 etc.). Typen er låst til at være floatingpoint-data. Selve dataen ligger i dataklasser, der udover at kunne tilføje disse referencer, også skal være i stand til at kopiere sig selv. Bekvemmelighedsfunktioner til at hente specifikke typer data, eksempelvis positioner eller normaler, er yderligere implementeret.

Udgangspunktet for en model, vil som regel være data om en model, der hentes fra en fil. Denne data konstrueres ved hjælpe af en såkaldt factory klasse, der læser den format-specifikke data ud fra en fil-loader, afsnit 6.2, og danner et dataobjekt med ovenstående egenskaber. "Loadere" kan på denne måde specificeres uafhængigt af mesh-strukturen, hvorved nye loadere let kan tilføjes.

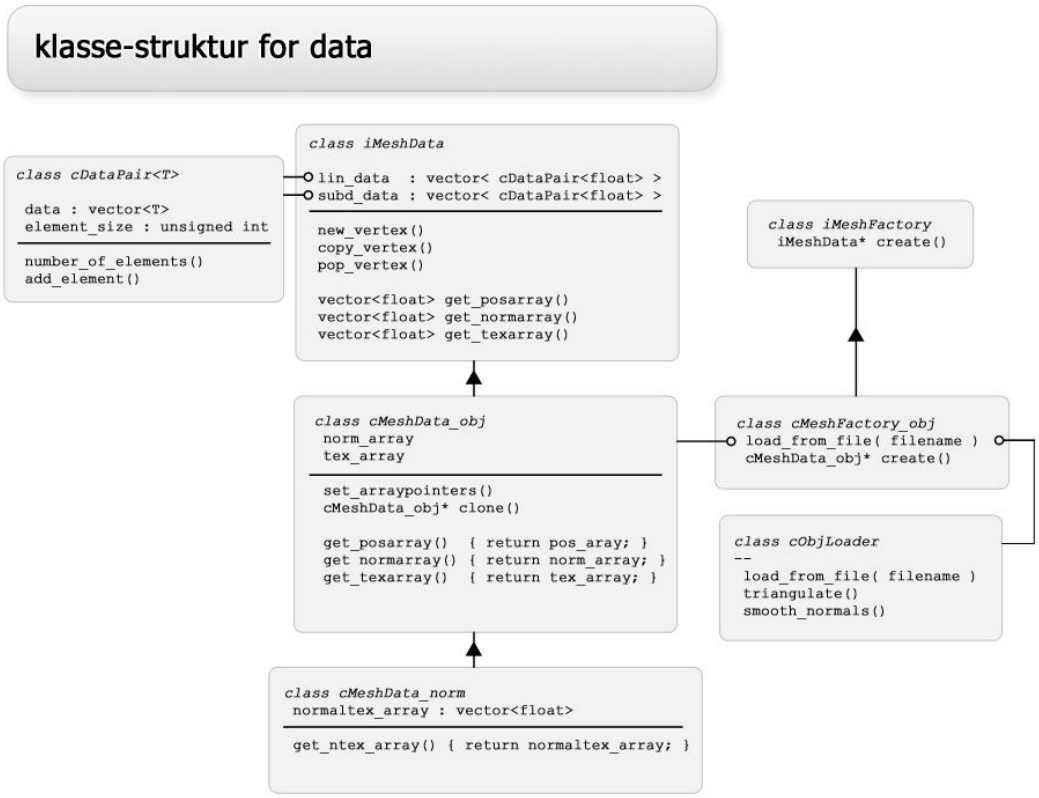


Illustration 6.8:

I praksis er den lineære interpolation implementeret som en subdivision-metode, hvilket ville gøre det let at generalisere yderligere, og knytte generelle subdivision-regler til sæt af data, frem for den nuværende begrænsning til lineær eller ikke-lineær data. En anden relevant udvidelse ville være, at give mulighed for at knytte data sammen runtime. Den nuværende løsning, der giver mulighed for at angive datasammenhænge ved kompilering, er rigeligt fleksibelt til den givne anvendelse, men kan sætte begrænsninger hvis strukturen ønskes anvendt i en mere generel sammenhæng.

6.3 Rendering af modeller

For at gøre det lettere at skifte imellem forskellige måder at tegne en model, er alle OpenGL-relaterede kald isoleret i separate klasser. Der er lavet et iRenderer interface der udbyder højniveau funktioner – *draw()*, *upload()*, *init()*, *deinit()*. Eneste krav til implementeringen er, at den ikke må ændre på objektet, og at *draw* ikke må ændre på renderen. Det er således helt legalt at *upload* intet gør reelt – hvilket f.eks. er tilfældet for OpenGL-immediate mode rendering. Data der skal renderes gives som konstante referencer via en render-klasses contructor.

Et naturligt yderligere abstraktionsniveau ville være en klasse der samlede en render og en data-klasse under et fælles interface.

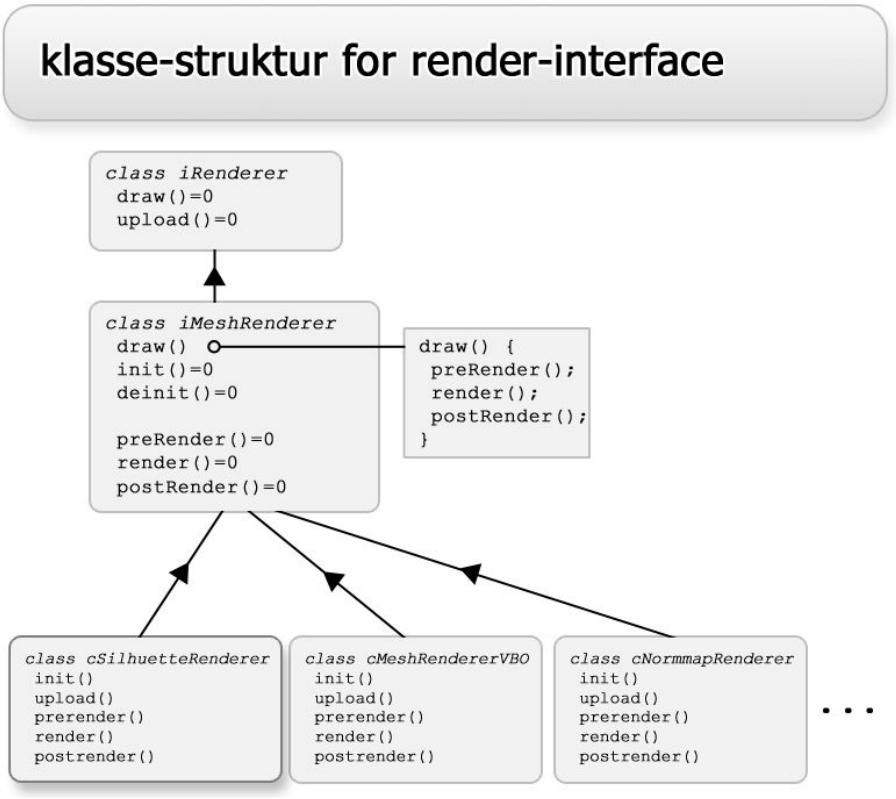


Illustration 6.9:

6.4 Subdivision af kanter alene

I dette afsnit forklares det hvordan beregning af underinddelte kanter via tabeller foretages i praksis. Fremgangsmåden er fuldstændig analog til fremgangsmåden beskrevet i teori-afsnit, bortset fra at der her kun betragtes punkter på kanten. Det anbefales derfor at teori-afsnittet (afsnit 3.2) læses forud for dette. Udover den praktiske forskel på de to metoder, vil der i dette afsnit blive givet en mere detaljeret gennemgang af den konkrete fremgangsmåde for generering af tabellerne.

6.4.1 Tabelgenerering for kant-underinddeling

Målet er hurtigt at kunne beregne grænsepunkter på kanter i en subdivision model. Vi er ikke interesserede i at finde indre punkter på trekanterne, men ser kun på kanterne angivet i den oprindelige kontrolmodel. Vi antager at modellen er en trianguleret, lukket to-mangfoldighed, således at der kun skal genereres tabeller for de almindelige loop-regler – og ikke for åbne modeller, skarpe kanter og spidse punkter. En udvidelse vil dog være triviel, og vil resultere i større tabeller og lidt langsommere beregninger af flade-punkter *[SCHRÖDER3]*. Kravet til triangulering hænger sammen med anvendelsen af loop-subdivision. Hvis metoden baserede sig på catmull-clark subdivision i stedet, ville alle flader bestå af firkanter efter en enkelt subdivision iteration.

Genereringen af tabellerne foretages på samme måde som i kapitel 2.5.1. Forskellen er, at kun punkter på kanten beregnes, mens de indre punkter i trekanten udelades. Dette gør at tabellerne kan

begrænses betydeligt, da der kun skal gemmes vægte for punkterne på kanten. Yderligere er støtten også en smule mindre, om end ikke ret meget, hvilket igen gør at færre vægte skal gemmes. På illustration 6.10 ses støtten for en kant, der er givet ved foreningsmængden af støtterne for de to endepunkter til kanten, se afsnit 2.5.2.

For at begrænse antallet af kombinationer af valenser for kantens endepunkter, antages det, at kun det ene endepunkt kan være irregulært. Denne antagelse er ikke begrænsende for hvilke modeller metoden kan anvendes på, idet enhver model kan bringes til at opfylde kravet ved initielt at foretage en enkelt subdivision-iteration. I afsnit 6.5.1 vises det, at denne indledende subdivision ikke resulterer i nogen væsentlig ulempe i den aktuelle anvendelse.

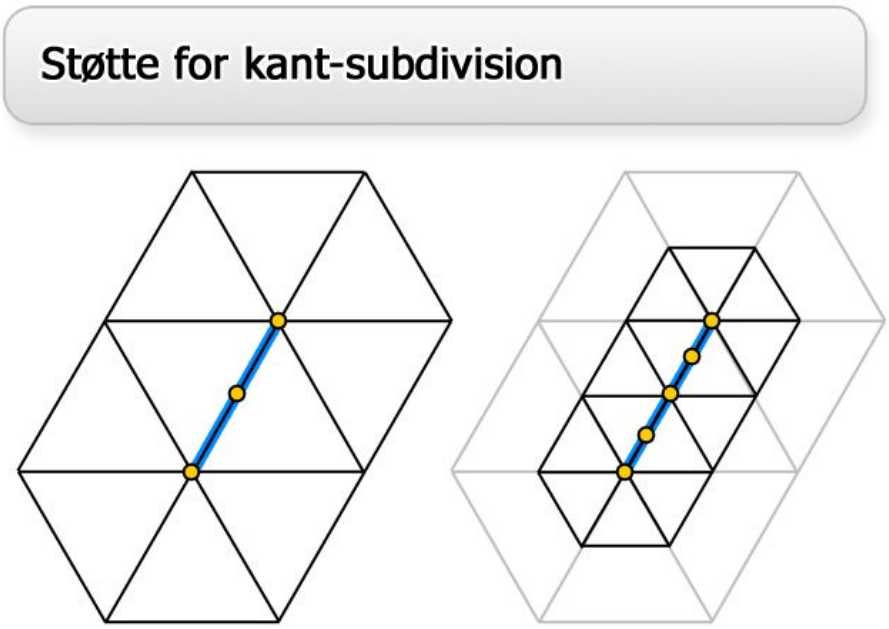


Illustration 6.10:
Støtten for en beregning af kanter på en kant ved første og anden iteration. Det bemærkes at støtten kun har to punkter mindre end for den fulde trekant.

Der genereres kun tabeller for valenser op til den højeste valens i den nuværende model. Idet vi antager at modellen er en trianguleret, lukket, 2-manifold, er det ikke meningsfyldt at tale om punkter med valens et eller to. For en model med maksimal valens *n*, vil der altså dannes *n-2* sæt af tabeller. Tabelstørrelsen kan reduceres yderligere, ved kun at gemme vægte for valenser der rent faktisk optræder i den givne model. Dette vil mindske kravene til lager, og tiden det tager at generere tabellerne. Det anses dog ikke for nødvendigt i den givne sammenhæng, da tiden det tager at generere tabellerne i forvejen er ganske overskuelig.

Med de givne antagelser, vil den samlede størrelse af tabellerne over vægte, være givet ved

$$tabelstørrelse = \sum_{i=3}^{\text{max valens}} N_{\text{inner points}} M_{\text{support size}}$$

$$N_{innner\ points} = (2^{max\_subd} - 1), \quad antal\ punkter\ på\ kan\ underinddelt\ \text{max\_subd}\ gange$$

$$M_{support\ size} = 7 + i - 3, \quad 7\ er\ antal\ punkter\ i\ første\ støtte.\ 3\ er\ valensen\ af\ første\ støtte$$

formel 6.1

Som beskrevet i afsnit 3.2, beregnes kun tabeller for et bestemt subdivision niveau. Det er dog stadig muligt at beregne grænsepunkter for andre subdivision-iterationer. Dette sker trivielt ved at springe nogle af punkterne over i beregningen. Det efterfølgende gennemløb af tabellerne med vægte bliver dog mindre lineært, hvilket potentielt gør beregningen mindre cache-venlig.

Det første der skal gøres når vægtene af støtten til en kant skal beregnes, er at støtten for denne kant skal dannes. Givet antagelsen om kun et irregulært punkt per kant, er støtten entydigt bestemt ud fra valensen af dette måske irregulære punkt. I praksis er der gemt en "basis"-støtte, der er den simpleste støtte mulig – nemlig for en kant med et irregulært punkt der har valens tre, se illustration 6.11. For at danne en støtte med valens *n*, splittes den "sidste" trekant *n-3* gange – se illustration 6.12. Med brug af lath-strukturen gøres dette trivielt som følger. Den kant, i den sidste trekant, der ønskes splittet, er associeret med *en* lath – på illustration 6.11 er det lath nr. 19. Lad *cf(L)* og *ccf(L)* betegne den lath der findes ved at gå henholdsvis med og mod uret rundt i trekanten. Så dannes støtten for en kant med valens *n*, ud fra en støtte med valens tre, som følger.

$$\begin{array}{l} CREATE-SUPPORTMESH(n) \\ \quad lath\ L \leftarrow ccf(lath_{19}) \\ \quad \textbf{for}\ i = 0 \ldots n-3 \\ \quad\quad split\_edge(cf(L)) \\ \quad\quad add\_edge(ccf(L),\ cf(L)) \end{array}$$

formel 6.2

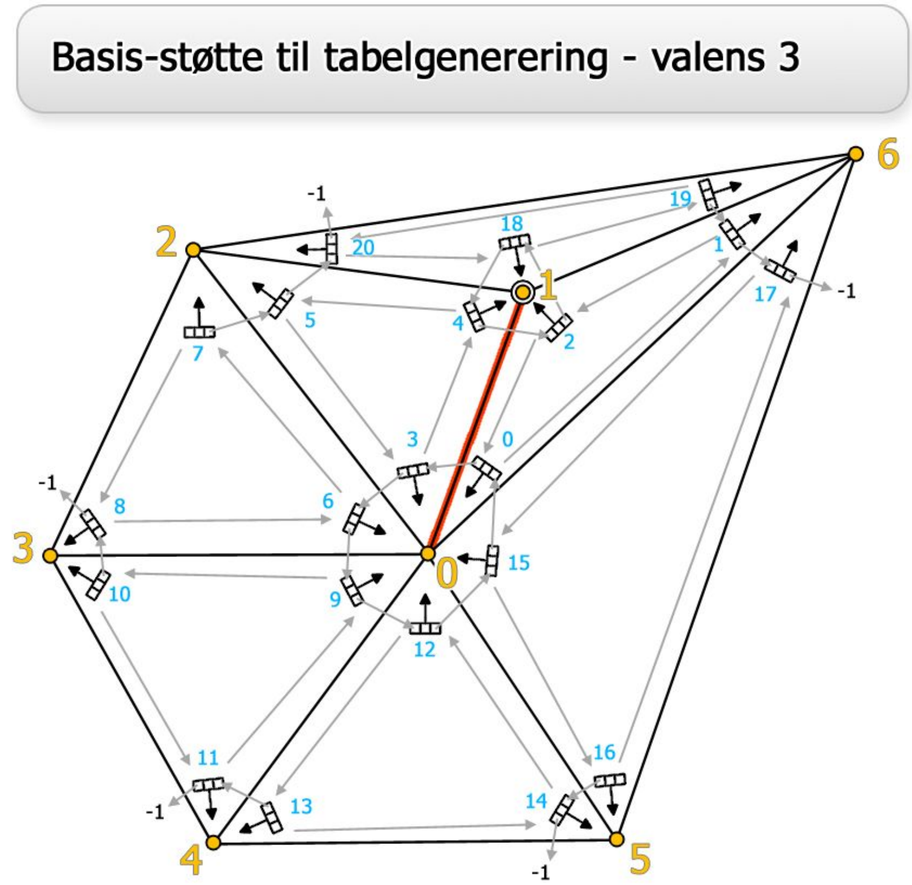


Illustration 6.11:
Udgangspunkt for generering af støtten. Punkt nr. 1 er det irregulære punkt hvis valens skal ændres. Den underinddelte kant er markeret med rødt.

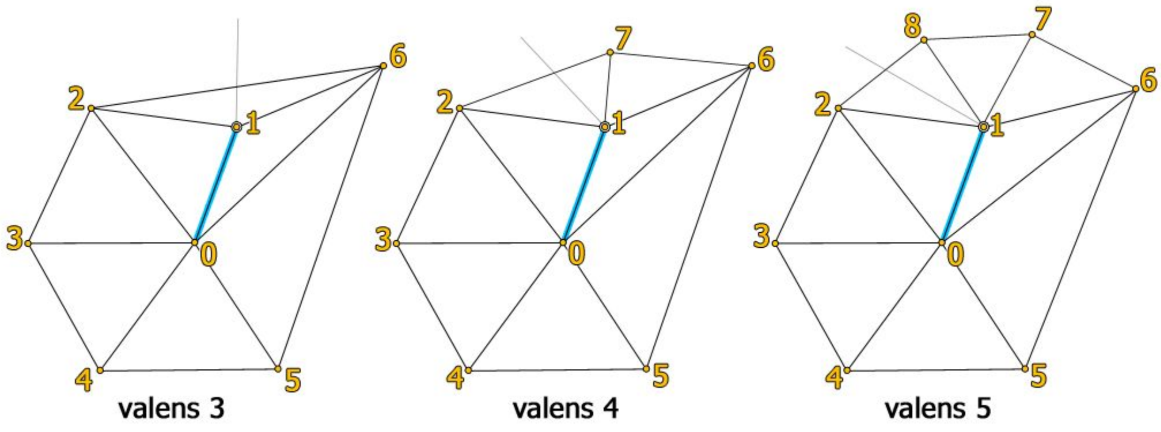


Illustration 6.12:
Eneste irregulære vertex er nr. 1. For at danne støtter med højere valens, deles trekanten efter sidste vertex.

Nu hvor vi har modeller for alle ønskede støtter, skal vi finde vægten af hvert punkt i støtten, med hensyn til et underinddelt punkt på kanten. Måden vi gør dette er at fortage en "en-dimensionel" subdivision af støtten – et punkt har altså kun en enkelt værdi tilknyttet. Alle punkter tildeles værdien *0.0* bortset fra det punkt i støtten vi ønsker at undersøge – det får værdien *1.0* – se illustration 6.13. Herefter subdivides hele støtten det ønskede antal gange, og alle punkter skubbes til deres grænsepositioner. Vi kan nu for hvert punkt på kanten aflæse den nye position som vægten af det undersøgte støttepunkt. Dette gentages for alle punkter i støtten, for alle valenser. Dette giver følgende fremgangsmåde for generering af støttevægte, når modellen maksimalt kan underinddeles *k* gange, og *SIZE* angiver antal punkter i en model.

```
BUILD-WEIGHT-TABLES( max_valens )
 1   for n = 3…max_valens
 2     B ← CREATE−SUPPORTMESH(n)
 3     for p = 0…SIZE(B)
 4       S ← B
 5       <flyt et enkelt punkt i S til 1.0>
 6       SUBDIVIDE(S, k)
 7       CALC−LIMIT−TANGENTS(S)
 8       GET−TANGENT−WEIGHTS−FROM−MESH(S)
 9       PUSH−TO−LIMIT−SURFACE(S)
10       GET−WEIGHTS−FROM−MESH(S)
```

formel 6.3

(forklaring af tangentvægte præsenteres i næste afsnit)

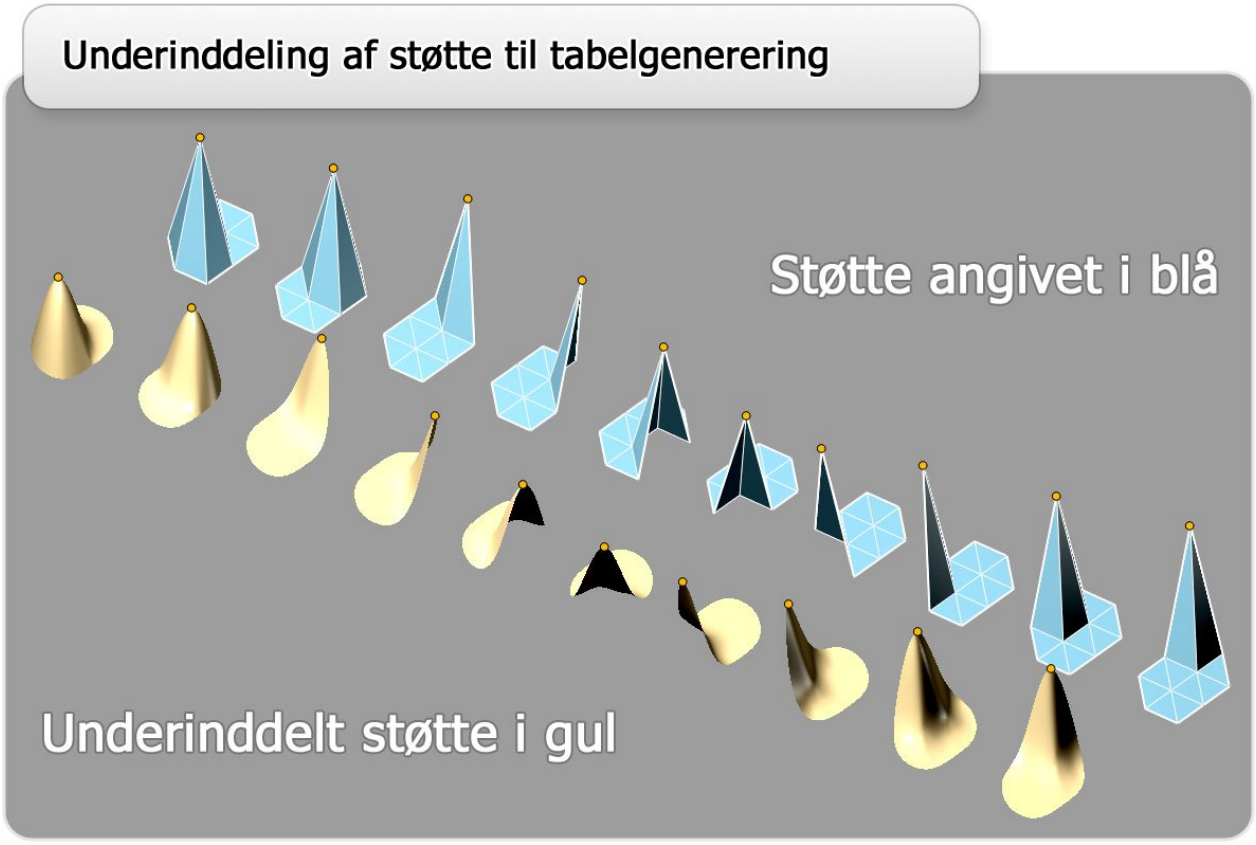


Illustration 6.13:
Processen kan visualiseres som hvert punkt i støtten på skift flyttes til 1.0 langs en af akserne, hvorefter modellen underinddeles. Her vises for det regulære tilfælde.

Det er ikke interessant hvilke regler der anvendes for ”åbne” kanter, selvom støtten er en åben flade. På illustration 6.10 s. 90, er forsøgt illustreret, at den relevante støtte for underinddeling af en kant, ”skrumper” for hver iteration. Beregningen af de yderste punkter i støtten er således ikke relevant i forhold til de indre punkter på kanten. En mulig optimering i tabel-genereringen kunne være at springe denne beregning over, og helt smide de yderste punkter væk efter hver subdivisioniteration.

6.4.2 Tabeller til normal-opslag

Det er ikke umiddelbart indlysende hvordan en tabel til normalgenerering skal dannes. Grænsenormaler til en subdivision flade er som nævnt i afsnit 2.5.1 givet ved krydsproduktet mellem de to tangenter til grænsefladen. Som forklaret i afsnit 3.2 er den bedste løsning, at gemme koefficienterne for hver af de to tangenter, og foretage krydsproduktet samt en eventuel normalisering under beregningen af selve kanten. Dette gør ikke alene tabellerne større, men også beregning af normaler væsentligt dyrere end for positioner alene. Køretiden for beregning af en kant med valens n, er dog stadig $O(n)$ – blot med en konstant til forskel: Der foretages dobbelt så mange vægtninger som for positionsberegningen, og et supplerende krydsprodukt samt en normalisering af den beregnede normal. Køretiden for beregning af punkter på en kant, O_{pos}, afhænger af henholdsvis valensen, og antallet af punkter på en kant.

- Der beregnes k subdividede punkter på kanten. Dette er en compile-time konstant.

- Hvert punkt vægtes i forhold til hvert punkt i støtten til kanten.
- Antallet af punkter i støtten, S, er direkte afhængigt af valensen, n, af det irregulære punkt på kanten, nemlig ved $S=n+4$, se afsnit 6.4.1.

Køretiden for beregningen af punkter på en kant er således

$$
\begin{aligned}
& O_{norm} = 2O_{pos}+c \\
\Leftrightarrow\ & O_{norm} = O_{pos}
\end{aligned}
$$

$$
\begin{aligned}
& O_{pos} = O(kS) \\
\Leftrightarrow\ & O_{norm} = O(k(n+4)) \\
\Leftrightarrow\ & O_{norm} = O(kn+4k) \\
\Leftrightarrow\ & O_{norm} = O(kn) \\
\Leftrightarrow\ & O_{norm} = O(n), \quad \textit{hvis k holdes kontant}
\end{aligned}
$$

formel 6.4

Fremgangsmåden for beregning af tangenttabeller er meget den samme som for positioner. For hver valens dannes kantens støtte, og et enkelt støttepunkt flyttes til *1.0*. Modellen underinddeles, men skubbes nu ikke til grænsepositionen. Herefter beregnes for hvert indre punkt på kanten de to 1d-tangenter,og disse gemmes som tangent-vægte for det flyttede punkt. I praksis slås tabelberegningen sammen med den for positioner, som i formel 6.3, hvorved overheadet ved at lave støtten flere gange undgås. Samlet fås således en tabel med tre vægte for hvert punkt i støtten – en for position, og to for tangenter.

6.4.3 Beregning af grænsepunkter på kanten

Beregningen af punkter på kanter via tabelopslag foregår præcis som trekantberegningen beskrevet i afsnit 3.2. Det er væsentligt at være opmærksom på hvilken rækkefølge vægtene traverseres, af hensyn til sammenhængende tilgang til hukommelsen. I den implementerede prototype foretages beregningen af samtlige N punkter, til et array d, på en underinddelt kant med støtte bestående af S punkter, som følger:

```
CALC-SUBD-EDGE-POINTS(L)
1  support ← GET-SUPPORT(L)
2  for i = 0...N
3    d[i] ← 0
4    for j = 0...S
5      d[i] ← d[i] + tabel[iS+j] positioner[support[j]]
```

formel 6.5

Beregningen af tangenter på kanten er fuldstændig analog til ovenstående beregning for positioner. Efter beregningen af tangenter, findes normalen i det givne punkt ved et vektorprodukt imellem tangenterne, hvorefter normalen normaliseres til enhedslængde.

For ikke at skulle finde støtten for en given kant hver gang modellen skal tegnes, dannes en liste med index til støtte-punkterne, der anvendes som cache. Søgning efter støtten er ikke langsomt, den består trivielt af to traverseringer af nabopunkter til hvert endepunkt, men for ikke at ødelægge den sekventielle tilgang til hukommelsen under beregningen af punkterne, lægges støtten i en liste for hver kant. Cachen er valid selvom modellen animeres eller morphes. Hvis netværkstopologien i modellen ændres, kan disse lister dannes for hver animationsopdatering, uden større hastighedsforringelse.

6.5 Søgning efter silhuetpunkt over kanter

I dette afsnit beskrives hvordan skæringspunkter imellem silhuetten og kanter i kontrolmodellen findes. Det vi ønsker, er punkter på grænsefladen, hvorfra en vektor til øjet står vinkelret på normalen. Vi begrænser søgningen til at foregå på kanterne fra den oprindelige kontrolmodel, men ønsker stadig punkter på grænsefladen. Fremgangsmåden for søgning på en kant er altså, at finde punkter på grænsefladen via subdivision af kanten, og søge i de resulterende punkter med tilhørende normaler. Søgningen sker som vist på illustration 6.14, ved hjælpe af simpel bisektion af den valgte kant.

Indledningsvist undersøges det, om normalerne til kanten peger i hver sin retning i forhold til kameraet. Idet det antages at silhuetten kun løber igennem hver kant maksimalt en gang, er dette et første gæt på hvorvidt en kant skæres af silhuetten. Hvis normalerne vender i hver sin retning i forhold til silhuetten, foretages en fuld søgning på kanten, ved at beregne punkter og normaler på den subdividede kant. Under bisektionen undersøges for hvert segment, om normalerne til endepunkterne peger i hver sin retning. Antagelsen om at kanten kun kan skæres en enkelt gang, dikterer at dette kun kan gælde for det ene af to segmenter. Når det mest detaljerede niveau er undersøgt, og det segment af den subdividede kant som silhuetten skærer, er fundet, approksimeres skæringspunktet med en lineær interpolation. Denne foretages over det sidste segment, i forhold til normalerne i endepunkterne.

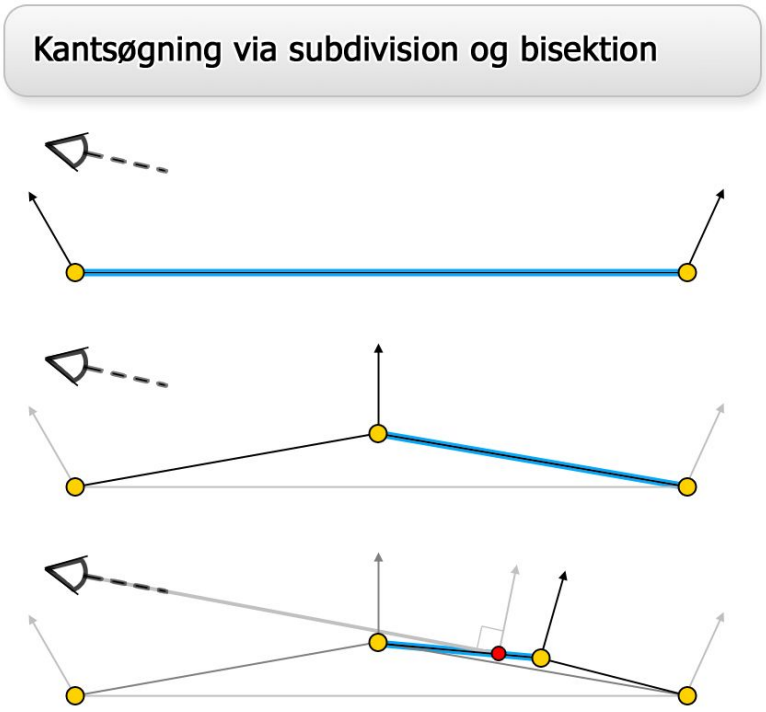


Illustration 6.14:
Søgningen foretages ved iterativ underinddeling af kanten. Efter sidste underinddeling foretages en lineær interpolation hen over kanten, hvorved det endelige punkt findes, her markeret med rødt.

For et øjepunkt med position E, og en kant med endepunkterne p_1 og p_2, er algoritmen til at finde et skæringspunkt givet som følger. Algortmen kører m iterationer, og k er antal punkter på en subdivided kant, hvilket skal være mindst 2^m+1 punkter.

$$
\begin{array}{ll}
\multicolumn{2}{l}{CALC-SILHUETTE} \\
1 & \textbf{for } each\ edge\ L \\
2 & \quad \boldsymbol{e_1} \leftarrow \dfrac{E-L.p_1}{|E-L.p_1|} \\
3 & \quad \boldsymbol{e_2} \leftarrow \dfrac{E-L.p_2}{|E-L.p_2|} \\
4 & \quad d_1 \leftarrow \boldsymbol{e_1} \cdot L.n_1 \\
5 & \quad d_2 \leftarrow \boldsymbol{e_2} \cdot L.n_2 \\
6 & \quad \textbf{if } (d_1 < 0 \ \ \textbf{xor} \ \ d_2 < 0) \quad \#\ silhuette\ skærer\ kant \\
7 & \quad\quad CALC-SILHUETTE-POINT (L, d_1, d_2)
\end{array}
$$

formel 6.6

$CALC-SILHUETTE-POINT(L, d_1, d_2)$

1 $\boldsymbol{b} \leftarrow CALC-SUBD-EDGE-POINTS(L)$
2 $\boldsymbol{n} \leftarrow CALC-SUBD-EDGE-NORMALS(L)$
3 $i1 \leftarrow 0,\ i2 \leftarrow k+1$
4 ***repeat*** m ***times***
5 $\quad im \leftarrow (i_1+i_2)/2$
6 $\quad \boldsymbol{e}_m \leftarrow \dfrac{E-\boldsymbol{b}_{im}}{|E-\boldsymbol{b}_{im}|}$
7 $\quad d_m \leftarrow \boldsymbol{e}_m \cdot \boldsymbol{n}_{im}$
8 $\quad$ ***if*** $(d_1 < 0$ ***xor*** $d_m < 0)$
9 $\quad\quad d_2 \leftarrow d_m$
10 $\quad\quad i2 \leftarrow im$
11 $\quad$ ***else***
12 $\quad\quad d_1 \leftarrow d_m$
13 $\quad\quad i1 \leftarrow im$
14 *end of repeat*
15 $w \leftarrow \dfrac{d_1}{d_2-d_1};$
16 $\boldsymbol{p}_{silhuet} \leftarrow lerp(w, \boldsymbol{b}_{i1}, \boldsymbol{b}_{i2})$
17 $\boldsymbol{n}_{silhuet} \leftarrow lerp(w, \boldsymbol{n}_{i1}, \boldsymbol{n}_{i2})$
18 $\boldsymbol{n}_{silhuet} \leftarrow \dfrac{\boldsymbol{n}_{silhuet}}{|\boldsymbol{n}_{silhuet}|}$

formel 6.7

Som beskrevet i afsnit 5.2, holder antagelsen om et enkelt skæringspunkt dog ikke trivielt. En silhuet kan løbe igennem en kant nul, en, to eller flere gange. Hvis kanten skæres et lige antal gange, vil ovenstående søgning slet ikke finde en skæring, idet fortegnet på det to normaler vil være ens. En måde at mindske dette problem på, der dog er meget omkostningsfuld, ville være at foretage søgningen på alle kanter. Det svarer dog i praksis til at foretage en eller flere indledende subdivison-skridt på alle kanter i modellen. Yderligere vil dette ikke løse problemet, men blot mindske det. En anden fremgangsmåde kunne være at danne en omsluttende kegle for normalerne på en kurve *[AZUMA, HOPPE2]*. Det ville gøre det muligt også at søge i kanter, hvor endepunkterne ellers ikke pegede i hver sin retning. Denne fremgangsmåde ville ikke være anvendelig på dynamiske modeller, hvor disse kegler vil ændre sig med modellen. I praksis er tredobbelte skæringer ikke observeret trods mange test. Det vil derfor i det følgende antages, at alle kanter skæres enten en gang af silhuetten, eller slet ikke. Det følger heraf, at alle trekanter har enten to eller ingen kanter der skæres af stilhuetten.

Når den her beskrevne søgning er afsluttet, haves et punkt der ligger på grænsefladen til subdivision-fladen, samt normalen til fladen i dette punkt. Idet vi ønsker at anvende normalmapping på de silhuetkorrigerede flader, er det yderligere nødvendigt at danne teksturkoordinater i de fundne silhuetpunkter. Disse findes ved, for hvert silhuetpunkt, at finde det nærmeste punkt på kanten i den oprindelige model, se illustration 6.15. Teksturkoordinaten i dette punkt findes herefter, som under sædvanlig rasterisering, ved lineær interpolation.

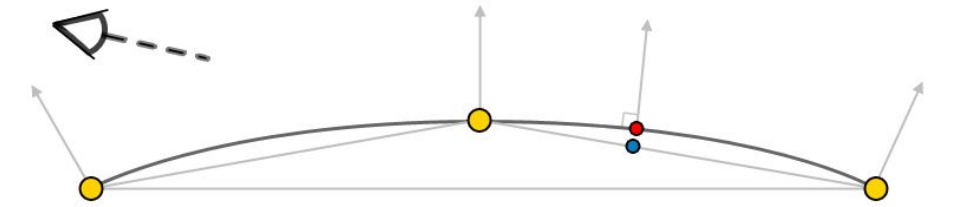


Illustration 6.15: Teksturkoordinaten til et silhuetpunkt gives ved det nærmeste punkt på kanten i lavpolygon-modellen.

6.5.1 Søgning over oprindelige kanter

Det er en væsentlig begrænsning at modellen underinddeles først. Dette gøres som nævnt for at reducere antallet af tabeller der skal gemmes. Ved at foretage en indledende underinddeling sikres at der kun er en irregulær vertex per kant, da Loop-underinddeling indsætter et regulært punkt på alle kanter. Efter denne pre-processering, er det tilstrækkeligt at gemme en tabel for hver forekommende valens i en model, frem for alle kombinationer.

En sådan indledende underinddeling har dog en række ulemper, hovedsageligt fordi det grundlæggende objekt vil indeholde fire gange så mange trekanter. Det betyder at langt flere kanter skal undersøges for om de ligger på en silhuetkant, hvilket forlænger søgetiden. Yderligere gør det, at de områder der ikke identificeres som tilhørende silhuetten, vil indeholde flere trekanter, hvilket gør selve renderingen langsommere. Løsningen på dette problem, er at huske på sammenhængen imellem kanterne i den oprindelige model, og i den hvor en indledende underinddeling er foretaget. Det giver mulighed for at undersøge om en oprindelig kant er en del af silhuetten, og herefter foretage søgningen efter det specifikke punkt på den underinddelte model.

I den specifikke implementering, er dette gjort blot ved at gemme den oprindelige model. I den givne implementering af loop-subdivision, ændrer en enkelt iteration ikke på de oprindelige laths' indeks. Dette betyder, at en lath fra basismodellen let kan identificeres efter en subdivision-iteration. Dette gør det trivielt at gå fra en silhuetsøgning på den grove model, til en søgning på den underinddelte model: Hvis en kant identificeres som en silhuet-kant, foretages første skridt i søgningen manuelt – nemlig sammenligningen imellem det første subdivision-punkt på kanten, og endepunkterne. Dette giver en kant i den underinddelte model, og søgningen kan herefter foretages som beskrevet i afsnit 6.5. Denne fremgangsmåde minder om silhuette-clippping *[HOPPE1]*, hvor et hierarki af kanter dannes, hvilket gør det muligt at foretage en søgning efter den korrekte silhuet.

6.5.2 Kurver hen over flader ud fra fundne silhuet-punkter

Vi har nu for alle kanter som silhuetten skærer, givet koordinaterne for skæringspunktet, normalen til fladen i punktet, samt teksturkoordinaterne. Opgaven er nu, for hver trekant som silhuetten skærer, at finde en kurve på subdivision-fladen, imellem de to silhuetpunkter. Kurven skal

interpolere endepunkterne, og normalen til fladen skal ideelt være orthogonal på øjevektoren i alle punkter på kurven, som yderligere skal ligge på grænsefladen.

For at gøre situationen så enkel som muligt, ønsker vi at danne en kurve med den lavest mulige grad, der giver en rimelig lighed med subdivision flader. Idet de fleste algoritmer til subdivision danner flader der er identiske med B-splines, er et første bud på en approksimation til silhuetten, en B-spline kurve. Normalerne til endepunkterne af de to kanter, kan som nævnt i afsnit 5.2 være vindskæve. Yderligere kan silhuetpunkterne være forskudt i forhold til hinanden, fordi subdivision-fladen kan "svinge" uens på de to kanter. Dette betyder, at det ikke er nok at anvende en kvadratisk kurve, da denne kun vil kunne svinge en enkelt gang, se illustration 6.16.

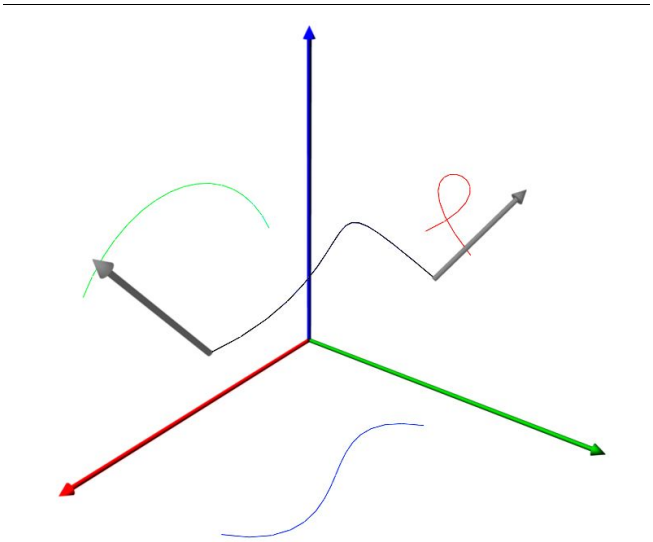


Illustration 6.16:
Det ses, at projektionerne ind på de forskellige koordinatplaner ikke giver kurver der kan beskrives med en kvadratisk kurve.

I *[THÖLE]* vises det, at kurver på en Loop subdivision-flade får samme kontinuitet som fladen selv. Dette vises dog kun for kurver med kontrolpunkter placeret i nabotrekanter, og dannet i parameter-rum[26]. Denne situation er ækvivalent til at tegne silhuet-kurver hen over en enkelt trekantflade, idet kontrolpunkterne kan antages at ligge præcis på kanten imellem de to trekanter. Approksimation med en kubisk B-spline burde således være et rimeligt valg.

26 I *[ZORIN2]* beskrives en naturlig parametrisering givet ved de barycentriske koordinater til hver trekant, og et indeks til trekanten.. Den i *[THÖLE]* beskrevne kurvedannelse, danner b-spline kurver i denne parametrisering, og beregninger efterfølgende kurver på grænsefladen, langs denne kurve.

For at danne kontrolpunkter for silhuetkurven, benyttes samme approksimation som anvendt til PN-triangles, der danner en kubisk beziér-kurve, se afsnit 4.1.4. B-splines er stykkevist bezier-kurver, og vi ønsker kun at finde et kurvestykke svarende til et enkelt segment af kurven. Det gør således ingen forskel om vi repræsenterer kurven som en Beziér-kurve, eller en non-uniform B-spline.

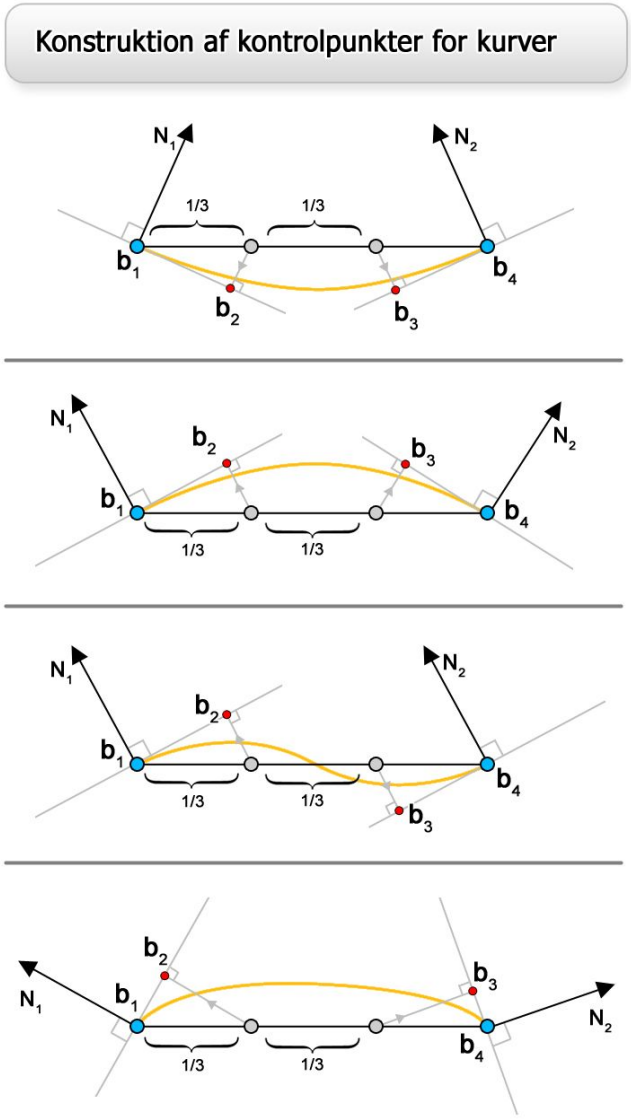


Illustration 6.17:
Med gul ses beziérkurven dannet ud fra kontrolpolygonen $\boldsymbol{b}_1 \ldots \boldsymbol{b}_4$*. Konstruktionen af kontrolpolygonen foregår som for PN-triangles, ved at projicere to ækvidistante punkter mellem* $\boldsymbol{b}_1$ *og* $\boldsymbol{b}_4$ *ind på de tilhørende normalplaner. Nederst ses, at for ekstreme normalvariationer, giver metoden meget flade kurver.*

En kubisk kurve er ikke unikt bestemt ved to punkter, og normalerne i disse to punkter, hvorfor det er nødvendigt at anvende en heuristik til at danne kontrolpolygonen. Silhuetkurven tilnærmes med en kubisk kurve med fire kontrolpunkter, der dannes ved projektion af ækvidistante punkter langs en kant, på normalplanerne til endepunkterne. I PN-triangles dannes kontrolpunkter til et trekantet beziér-patch, og denne heuristik anvendes til at danne punkterne på hver af de tre kanter. Der tilføjes to punkter til hver kant, så kanten deles op i tre lige store stykker. Første og sidste punkt i kontrolpolygonen til silhuetkurven er givet ved endepunkterne til kanten. De to resterende kontrolpunkter er givet som følger. For hvert af de to punkter på kanten, dannes et plan ved det nærmeste endepunkt, samt normalen til fladen i endepunktet. Kontrolpunkterne er herefter givet ved det nærmeste punkt på dette plan, til punktet på kanten. Se illustration 6.17.

Denne heuristik har nogle gode egenskaber – hovedsageligt at den giver en kubisk kurve der overordnet svinger ”rigtigt”, samtidig med at den interpolerer endepunkterne. Til gengæld afviger de dannede kurver i mere ekstreme tilfælde væsentligt fra en Loop subdivision flade. Når to normaler således peger i meget forskellige retninger, bliver punkterne projiceret ind ganske tæt på normalplanet – og giver således en lav bue, se illustration 6.17. Dette er modsat subdivision surfaces, eller B-splines generelt, der vil give en høj bue hvis der optræder høj krumning lokalt hen over kanten.

Det er således kun garanteret, at kurven er vinkelret på øjevektoren i endepunkterne, ikke hen over kurven. Yderligere er der ingen garanti for, at kurven ligger på grænsefladen, bortset fra i endepunkterne.

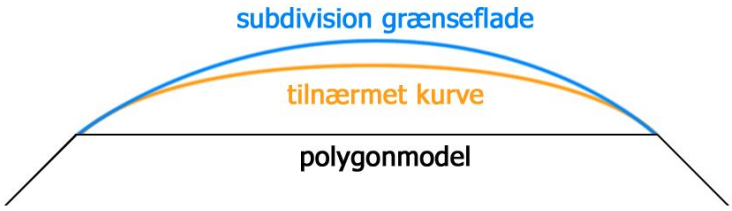


Illustration 6.18:
Den kubiske kurve dannet ved PN-triangle tilnærmelse, er ikke garanteret at ligge på grænsefladen.
(polygon-modellen ses her, ligeledes skubbet til grænsefladen)

Når vi senere tegner modellen, skal der for punkterne på silhuetkurven, bruges teksturkoordinater til opslag i normalmappet. Disse interpoleres blot lineært imellem kant-silhuetpunkterne. I den givne implementering er det også muligt at tegne en model med normaler, og disse implementeres ligeledes lineært hen over kanten, efterfulgt af en normalisering.

6.5.3 Triangulering af flader

Når alle trekanter er undersøgt, og silhuetkurver hen over de enkelte trekanter er dannet, er næste skridt at danne de trekanter der skal tegnes som approksimation for subdivision overfladen. Idet det antages at alle kanter skæres af silhuetten maksimalt en gang, og alle trekanter som silhuetten går igennem har præcis 2 silhuet-kanter, vil de to silhuet-kanter altid støde op til hinanden i hjørnet af en trekant. Der er således kun to tilfælde der skal betragtes, nemlig når enten to eller ingen kanter i en trekant skæres.

Det er valgt at foretage en triangulering af silhuettrekanter, der består af en triangle-fan, og et triangle-strip, se til højre på illustration 6.19. Denne triangulering giver en i nogen grad uniform opdeling af trekanten, er hurtig at tegne, og giver kun lille risiko for fejl. En udfordring ved denne type underinddeling er dog, at det er nødvendigt at indsætte punkter på kanten modsat silhuetkurven. Idet vi ikke underinddeler nabo-trekanten, får vi såkaldte ”T-vertices”.

T-vertices er uønskede af to grunde: For det første giver det samplingproblemer hvis lysberegninger foretages per vertex *[GOURAUD]*. For det andet er der en risiko for, at der opstår sprækker i modellen pga. afrundingsfejl, idet beregningen af punkterne på kanten foretages på to forskellige måder. I den givne anvendelse beregner vi dog belysningen per pixel, og har således ingen problemer med samplingen. De fleste moderne grafikkort foretager i dag alle beregninger i IEEE-754-1985 32bit floating point præcision. Dette er samme repræsentation som CPUer anvender. Afrundingsfejl skulle derfor opstå grundet forskelle i måden interpolationen af punkterne blev foretaget. Efter en række test, er der fundet ganske få eksempler på pixel-udfald ved anvendelsen af T-vertices. Disse forsvinder dog ved brug af, selv et meget lavt niveau af, normalmapping. Skulle dette i andre sammenhænge vise sig meget upraktisk, kan i stedet bruges den mindre praktiske triangulering vist til venstre på illustration 6.19.

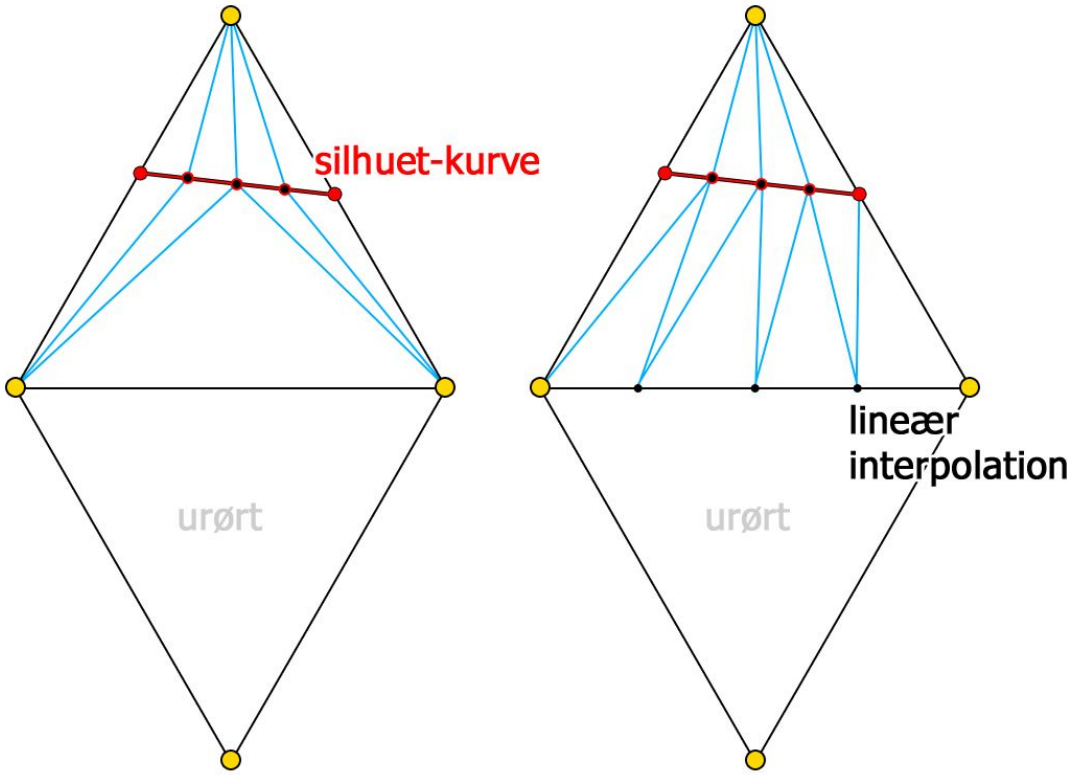


Illustration 6.19:
Det ses at trianguleringen til venstre giver meget lange trekanter. Den til højre giver til gengæld mange såkaldte T-vertices.

En anden mulighed ville være at lade trianguleringen propagere ud fra de underinddelte trekanter. Dette ville fjerne problematikken omkring T-vertices, men tilgengæld komplicere trianguleringen, idet trekanter ikke længere ville kunne beregnes enkeltvist. Yderligere vil det betyde at væsentligt flere trekanter tegnes. Dette kunne dog bruges til metodens fordel, idet den korrekte kant da ville kunne beregnes, hvilket ville betyde en mere korrekt visualisering.

Når fladen er konstrueret, tegnes hver trekant med OpenGL ”immediate-mode” kald – det vil sige at alle teksturkoordinater, normaler og positioner kræver et funktionskald hver. Dette er langt fra en optimal løsning af flere grunde. For det første tager de mange funktionskald ganske meget tid, hvilket tilføjer yderligere arbejde til CPUen. For det andet udnytter denne metode ikke de eksisterende metoder til hurtig overførsel af data til grafikkortet – primært AGP-burst og DMA-transfers. En bedre løsning ville således være, at anvende såkaldte ”Vertex-Buffer-Objects”, der gør det muligt at angive data på en, for hardwaren, mere hensigtsmæssig måde. Den nuværende metode er meget lig den der anvendes i *[BRICKHILL]*, hvor geometri tegnes umiddelbart efter konstruktion, og herefter smides bort.

6.6 Generering af normalmap ud fra subdivision fladen

En måde at gemme normalerne for en flade, uafhængigt af geometrien, er som tidligere nævnt, at gemme dem i en separat tekstur. For at kunne gøre dette, er det nødvendigt at angive, hvilke flader

der skal tildeles hvilke dele af teksturen. Opgaven er altså, at finde en parametrisering over den oprindelige flade.

I litteraturen er der flere måder at håndtere teksturkoordinater på for subdivision surfaces. En af de simpleste er også at underinddele teksturkoordinaterne via subdivision-regler *[PIXAR2]*. Denne fremgangsmåde vil dog ikke give gode resultater når det ønskes at anvende teksturer dannet ud fra højdetaljerede modeller, på de lavere detaljegrader. Interpolationen af teksturkoordinater sker sædvanligvis lineært – også når den udføres på grafikkort. Det betyder, at kanterne af de lavdetaljerede modeller, vil bruge tekstur-samples et stykke fra den subdivions-interpolerede kant. Dette gælder uanset at grænseposition-koordinater anvendes.

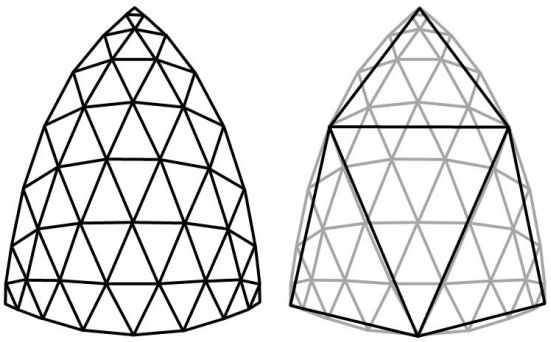


Illustration 6.20:
Ved brug af subdivision-regler til underinddeling af teksturkoordinater, kan et givent sæt koordinater kun anvendes på et subdivision-niveau.

I stedet vælges således at interpolere teksturkoordinater lineært. Dette resulterer i en ikke-uniform sampling af modellen, idet trekanter resulterende fra subdivision af en trekant, generelt ikke vil have ens størrelse. En ulempe ved denne metode er således, at præcisionen afhænger af positionen på trekanten.

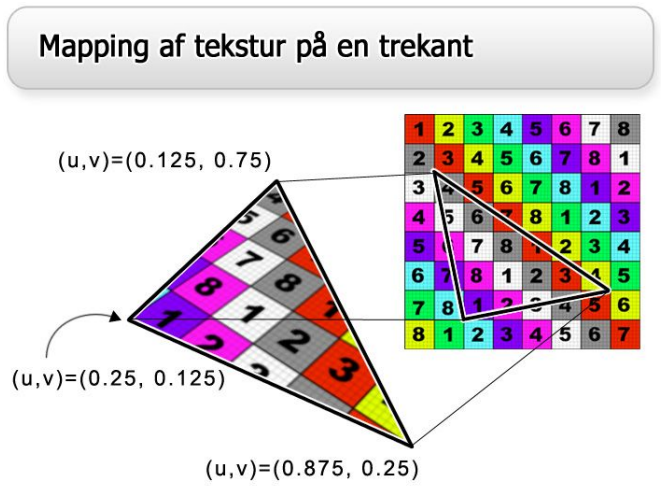


Illustration 6.21

6.6.1 uv-generering

Når der genereres teksturkoordinater, er der en række kriterier der er væsentlige for hvor godt resultatet er.

- Formen af modellen skal forvrænges mindst muligt.
- Der skal være en fornuftig sammenhæng mellem størrelsen af en polygon i mappet, og den oprindelige model.
- Der skal være mindst mulig spildplads.

- Delte teksturkoordinater i så vidt omfang som muligt, så modellen ikke skal splittes op.

Man kunne godt vælge at antage, at modellen har teksturkoordinater i forvejen. Men der er en række problemer forbundet med dette. For det første, vil pre-teksturerede modeller ofte anvende en række tricks – som f.eks. spejling. Endelig anvendes ofte mønstre der gentager sig ved brug af teksturkoordinater udenfor intervallet [0;1]. Et andet problem er, at der arbejdes med grænseflader. Når en model skubbes til grænsefladen, vil den blive forvrænget i større eller mindre grad – og således forvrænget i forhold til sine teksturkoordinater, der ikke forandres. Dette problem kan mindskes når vi selv genererer teksturkoordinaterne, ved at skubbe fladen til grænsefladen inden der genereres teksturkoordinater. Hvis uv-genereringen foretages på mængder af trekanter, kan det godt betale sig, at foretage en enkelt subdivision-iteration, for yderligere at tage hånd om forvrængningen inden for lappen.

For automatisk at finde passende teksturkoordinater, kan generelt anvendes en række forskellige teknikker. Den mest anvendte er, at foretage en række plane projektioner af modellen.

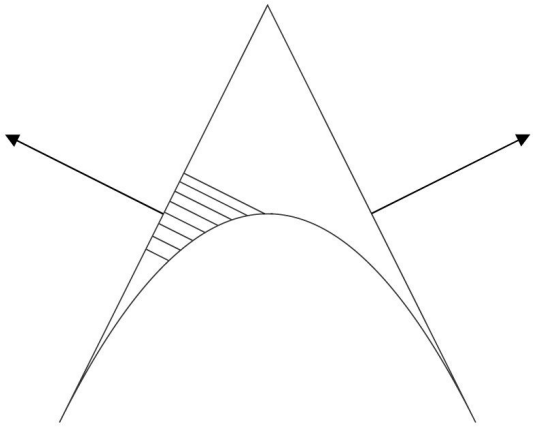


Illustration 6.22
Fejl ved projektion af krum flade på plan.

Denne metode vil dog give forvrængning i projektionen. Der er mange måder at løse dette problem på. Den simpleste er, at anvende "mange" projektionsplaner, og projicere en given flade ind på det plan der ligger mest parallelt med planet. Dette giver dog en ganske stor opdeling af modellen, og således flere diskontinuiteter langs kanterne. Der er udviklet en mængde generelle løsninger, hvoraf de fleste af ganske komplicerede *[LEVY1, SHEFFER1, SCHRÖDER4, HOPPE4]*. En måde at sikre forvrængningsfri projektion, er, at lave en projektion per trekant.

Der skal altså findes en projektion, fra objekt-koordinater til et lokalt koordinatsystem for en trekant. Denne type transformation er kendt fra lineær algebra, og kaldes en basis-skifte matrice *[EISING]*. En sådan matrice består ganske simpelt, af en basis for trekanten, angivet i objekt-koordinater. En sådan basis kan let findes, idet vi kan finde normalen til trekanten, og vælge en af siderne som anden akse. Vi har dermed, for en trekant bestående af punkterne v_1, v_2, v_3, at

$$
\begin{aligned}
\boldsymbol{T} &= \frac{v_1 - v_2}{\left\| v_1 - v_2 \right\|} \\
\boldsymbol{N} &= \boldsymbol{T} \times \frac{(v_3 - v_2)}{\left\| v_3 - v_2 \right\|} \\
\boldsymbol{B} &= \boldsymbol{N} \times \boldsymbol{T}
\end{aligned}
$$

$$
\boldsymbol{M} = \begin{bmatrix} t_0 & t_1 & t_2 \\ b_0 & b_1 & b_2 \\ n_0 & n_1 & n_2 \end{bmatrix}
$$

Formel 6.8

Det bemærkes at matricen ***M*** er præcis den samme, der anvendes når vi skal transformere en vektor til tangentrum – se afsnit 6. Et billede er to-dimensionelt – og teksturkoordinaterne er således givet ved en vektor med to elementer. Når vi transformerer en trekant med denne matrice, får vi tre koordinater. Den sidste koordinat vi dog altid være meget tæt på nul, da normalen per definition står vinkelret på trekanten. Teksturkoordinaterne til en vertex er altså givet ved transformation med ***M***, som følger

$$
(u, v) = \boldsymbol{M}\,\boldsymbol{v} = \begin{bmatrix} t_0 & t_1 & t_2 \\ b_0 & b_1 & b_2 \end{bmatrix} \begin{bmatrix} v_0 \\ v_1 \\ v_2 \end{bmatrix}
$$

Formel 6.9

Det bemærkes, at den her anvendte parameterisering, kan betragtes som analog til den naturlige parameterisering angivet i *[ZORIN2]*. Alle trekanter parameteriseres individuelt, men her vrider vi trekanterne for at matche deres oprindelige dimensioner, og herved undgå forvrængning.

6.6.2 Pakning af tekstur atlas

Det tager tid for grafikkort at skifte mellem forskellige teksturer. Det er derfor fordelagtigt, at lægge alle trekanter i et stort billede. Problemet er således, at pakke en mængde trekanter, så tæt som muligt, i en firkant. Dette problem, der også kaldes atlas-generering, er grundigt undersøgt inden for området ”computational geometry”, og det er bevist, at det er NP hårdt problem. Der findes en række gode løsningsmodeller, der dog alle er forholdsvist komplicerede. For simplicitetens skyld, er her valgt en meget enkel model. Alle trekanter approksimeres af deres boundingbox, rejses op, hvorefter de ”pakkes” *[COSGROVE1]*. Ved i formel 6.8 at lade ***T*** pege i samme retning som den længste side i trekanten, behøver vi ikke at rejse trekanten op.

Pakningen af trekanterne foregår ved, at de først sorteres efter højde, og herefter placeres i teksturen startende fra nederste venstre hjørne. Dette kan gøres ved alene at holde styr på øverste højre hjørne af bounding-boxen for kasser i nuværende- og sidste linie af trekanter. Inden indsættelsen skaleres

alle trekanter med en konstant faktor, beregnet ud fra summen af arealet af bounding-boksene samt antallet af trekanter. Når der ikke er mere plads på en linie, påbegyndes en ny linie oven på den tidligere række. Der sørges hele tiden for, at der er en vis afstand mellem trekanterne. Grunden til dette er, at nedsampling som f.eks. ved mip-mapping, ofte vil sløre billedet. Trekanter der støder op til hinanden vil derfor medføre fejl i lavere opløsninger af teksturen. Det bemærkes, at det er uundgåeligt at der vil opstå afvigelser og fejl, hvis opløsningen er uforholdsmæssigt lav.

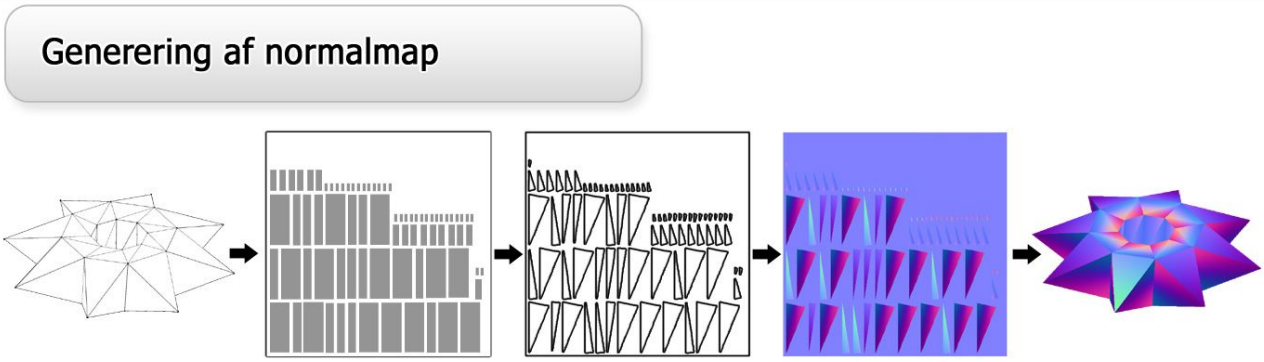


Illustration 6.23
Fra wireframe-model til påføring af normler givet tangentrum. Det sidste billede af modellen er en visualisering af normalerne inden de anvendes til lysberegning.

6.7 Evaluering af normaler

Evalueringen af grænsenormalerne til en subdivision flade er implementeret via underinddeling og en simpel pixelshader. For at sikre korrekte normaler i hver pixel, skal fladen underinddeles så meget, at alle pixels indeholder mindst en vertex. Anvendelse af non-uniform underinddeling ville være gavnligt i denne sammenhæng, da trekanterne har forskellig størrelse i teksturen. Pixelshaderen normaliserer blot normalen til fladen, og ændrer intervallet fra [-1..1] til [0..1], så normalen kan angives som en farve[27]. Normaliseringen er nødvendig, idet grafikkortet foretager lineær interpolation af alle vertex-parametre hen over polygonerne. En lineær interpolation ændrer generelt længden af normalen, hvorfor en efterfølgende normalisering er nødvendig. Underinddeles dog trekanten til subpixel niveau, er dette ikke nødvendigt.

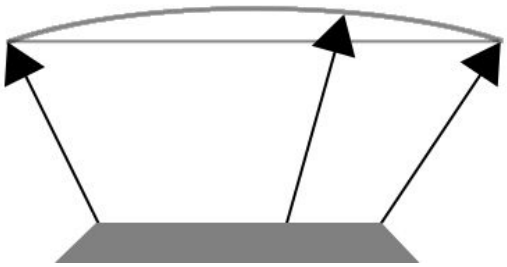


Illustration 6.10

Det er i den implementerede prototype valgt at gemme normalerne til objektet i i forhold til

27 For floating-point teksturer er dette ikke nødvendigt, idet det er muligt at repræsentere negative tal.

modellens lokale koordinatsystem – dette betegnes sædvanligvis ”objektrum”. Som vist i forprojektet, er det stadig muligt at foretage sædvanlig objekt-animation, såsom morphing og skinning. Dette kræver blot at basisen for objektrummet gemmes for hver vertex. I den foreliggende implementering er dette dog ikke gjort, men det er en triviel udvidelse der kan foretages når det viser sig nødvendigt.

6.8 Løsning af sampling problem omkring grænser i normalmap

Når en tekstur under rendering samples med andet end ”nærmeste nabo”-filtrering, foretages der en vægtning af flere pixels i teksturen. Det betyder, at der langs med kanten af en uv-mapping kan opstå problemer med, at ikke-samplede pixels tages med i vægtningen. En mulighed er at foretage denne sampling manuelt, og kun tage ”korrekte” samples med i vægtningen. Dog er den slags betingede operationer forholdsvist dyre på grafikkort, og metoden vil betyde at problemet rettes for hver frame.

En anden tilgangsvinkel til problemet kunne være at udfylde den ikke-samplede del af teksturen, med værdier fra samples der ligger tæt på. I forprojektet til dette projekt, tegnedes en linie langs alle kanter, og hjørnepunkter tegnedes dobbelt. En anden mulighed kunne være, at tegne ”vinger” langs alle uvmapping-kanter med samme tekstur som kanten. Hvis tekstur-atlasset er tæt pakket, er det dog ikke trivielt at undgå overlap imellem sådanne vinger.

Et alternativ til at rendere vinger, kunne være for alle ikke-samplede pixels, at beregne den nærmeste samplede pixel. Dette er et kendt og løst problem i traditionel billedbehandling. En billed-transformation der således finder netop nærmeste objekt, er en Euklidisk Distance Transformation (EDT). Algoritmen beskrives traditionelt i form af en scanline-algoritme, der gennemløber billedet to gange, og ser på tidligere pixels fra samme gennemløb *[CARSTENSEN]* . Der findes dog beskrivelser af algoritmen, der tillader at alle pixels i et billede evalueres parallelt *[YAMADA]*. Dette passer os godt, idet det gør det muligt at foretage en implementering direkte på grafikkortet. Den parallelle algoritme kan ikke måle sig med den tilsvarende scanline-algoritme i en direkte optælling af udførte operationer. Vi har dog ikke brug for at udregne et fuldstændig euklidisk vektorkort, men ønsker blot at tilføje en ”kant” til alle objekter i vores normalmap. Ved at beholde data på grafikkortet, undgås overheadet ved at kopiere data frem og tilbage imellem grafikkort og systemhukommelse. Det giver ikke direkte mening at sammenligne antallet af operationer, da nyere grafikkort netop i høj grad er parallelle i deres design[28], og i øvrigt generelt har væsentligt hurtigere hukommelse.
Som en sidebemærkning skal det nævnes, at algoritmen i høj grad baserer sig på heltalsoperationer. Disse foretages i floatingpoint på et grafikkort, der således vil foretage mere komplicerede beregninger end egentlig nødvendigt.

6.8.1 Parallel Euklidisk afstandstransformation

Givet et binært 2D-billede der angiver lokationen af objekter i planen, er opgaven, for alle pixels i billedet, at finde den korteste afstand til en objekt-pixel, samt en vektor til denne nærmeste pixel. Med ”afstand” menes den euklidiske afstand imellem pixelcentre

28 De nyeste Nvidia-grafikkort har op til 16 parallelle fragment-processorer, mens CPU'er stadig generelt kun har en enkelt kerne.

$$d_e(a,b)=\sqrt{(b_x-a_x)^2+(b_y-a_y)^2}\quad .$$

Formel 6.11:

I det følgende gennemgås algoritmen til parallel beregning af en EDT. Gennemgangen vil løst følge den givet i *[YAMADA]*.

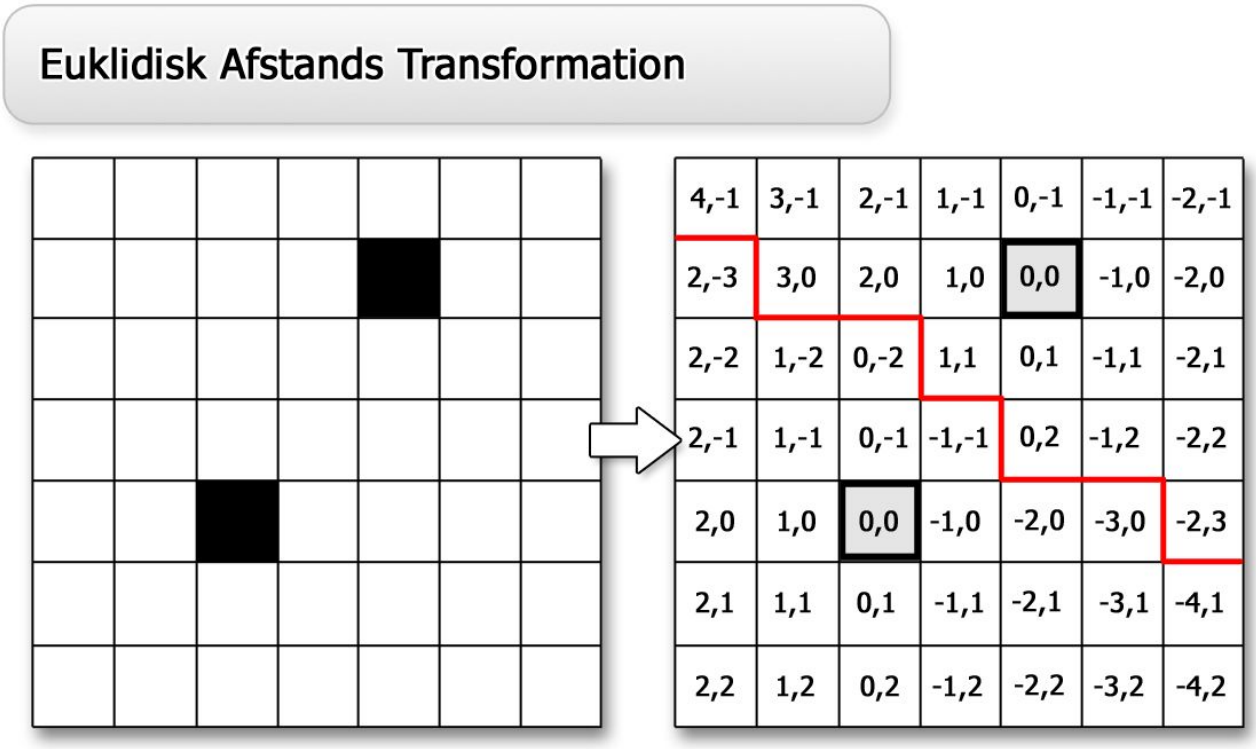


Illustration 6.24:
Det ses at transformationen deler billedet op i to regioner, svarende til pixels der ligger nærmest hvert objekt. Denne segmentering betegnes ofte et Voronoi-diagram.

Formelt kan situationen beskrives som følger. Givet et binært billede ***B*** bestående af to disjunkte mængder ***S*** og ***S'***. ***S*** er mængden af objekt-pixels, med værdien 1, og ***S'*** er mængden af ikke-objekt-pixels, med værdien 0. Et afstandstranformeret billede *L(**B**)*, er et billede hvor der til enhver pixel $p'=(x',y')\in \boldsymbol{S'}$ er knyttet en vektor *v* pegende på en pixel $p=(x,y)\in \boldsymbol{S}$, hvor $d_e(p,p')$ er minimal, for alle $p\in \boldsymbol{S}$.

Naboerne til en pixel defineres som på illustration 6.25, som mængden $\boldsymbol{u}_n\in U$. Algoritmen giver mulighed for at betragte enten 4 eller 8 naboer til en pixel. De to varianter definerer to forskellige mængder af naboer til en pixel *b*

$$U_4 = \{ b + \boldsymbol{u}_n \mid n = 1,3,5,7 \}$$
$$U_8 = \{ b + \boldsymbol{u}_n \mid n = 1..8 \}$$
formel 6.12

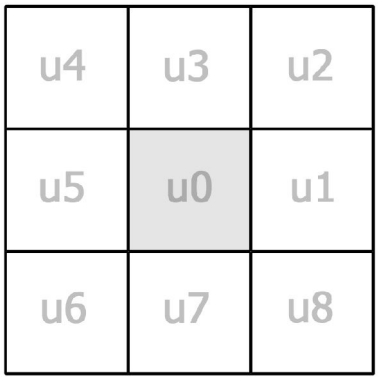


Illustration 6.25: definition af naboer. u0 er nulvektoren, og peger således til pixelen selv.

I *[YAMADA]* gives et bevis for, at 8-nabo varianten af algoritmen finder en fuldstændig og korrekt EDT. Benyttes 4-nabo udgaven kan der opstå små fejl i særlige tilfælde.

Algoritmen til at finde *L(**B**)* er herefter givet som følger. Først dannes et initielt vektorkort, $L^0(\mathbf{B})$, således at:

$$L^0(b) = \begin{cases} (0,0), & hvis\ b \in \boldsymbol{S} \\ (\infty,\infty), & hvis\ b \in \boldsymbol{S}' \end{cases}$$
formel 6.13

For en given iteration, $t = 0..N$, i algoritmen, er næste iteration $L^{t+1}(\boldsymbol{B})$ givet ved

$$L^{t+1}(b) = L^t(b + \boldsymbol{u}_{min}) + \boldsymbol{u}_{min},$$
$$- hvor\ \boldsymbol{u}_{min}\ er\ den\ \boldsymbol{u}_n \in U\ for\ hvilken\ det\ gælder\ at$$
$$d_e(L^t(b + \boldsymbol{u}_n) + \boldsymbol{u}_n)\ \ er\ minimal$$
formel 6.14

Hvis flere $\boldsymbol{u}_n$ giver samme afstand, vælges den nabo med det laveste indeks n.

Det endelige vektorkort, *L(**B**)* er fundet, når en iteration ikke ændrer nogen pixels i billedet, hvilket sker seneste efter samme antal iterationer som *max(højde, bredde)* i pixels. Modsat scanline algoritmen der giver en fuldstændig EDT efter blot to iterationer, kræver den parallelle variant lige så mange iterationer som bredden af den ønskede kant af valide afstandsvektorer, omkring de enkelte objekter – hver iteration tilføjer så at sige et ”lag” af pixels til det eksisterende vektorkort.

Algoritmen giver ikke nogen løsningsmodel for håndtering af kant-pixels. Dette er ligeledes en klassisk udfordring i billedanalyse, hvor algoritmer der kigger på nabo-pixels ikke er definerede for de yderste pixels. Løsningen kan være enten at give pixels der ligger udenfor billedet en ”neutral” værdi, eller simpelthen ikke at beregne de yderste pixels. Ser vi på EDT specifikt, giver den første løsning mulighed for at beregne en EDT for alle punkter i billedet, men kræver et validitets-check per sample, mens den anden blot skal checke per pixel. Den ”neutrale” værdi vil for EDT være en ”uendelig stor” afstand.

6.8.2 Konkret implementering

Udførelsen af den parallelle EDT-algoritme, foretages i praksis i tre skridt. Først dannes et nyt vektorkort ud fra alpha-kanalen i det oprindelige billede. Et vektorkort indeholder en vektor til nærmeste ”objekt”-pixel, men også afstanden til denne. Alle pixels med en alpha-værdi ”tilstrækkeligt”[29] tæt på 1 antages at tilhøre et objekt. Alle andre afstande sættes til en værdi større end den størst mulige tekstur-størrelse. Dette skridt foretages ved brug af en shader, der ser på alpha-værdien af en tekstur og skriver vektorer ud som floatingpoint-værdier i en såkaldt ”pixel-buffer” (pbuffer). Pixel buffere er OpenGLs måde at foretage ”render-to-texture”, og kan således anvendes både som framebuffer, og bindes som tekstur. Efter dannelsen af det initielle vektorkort, køres *n* iterationer af den parallelle algoritme. Dette sker ved, via en shader der sampler 4/8-nabopixels, at tegne fra et vektorkort over i en ny pbuffer. Der itereres altså frem og tilbage imellem to teksturer, og hver rendering svarer til en iteration i algoritmen.

Afslutningsvist opdateres den oprindelige tekstur ud fra den endelige vektorkort. Dette foretages ligeledes med en shader, der i to skridt finder den nye værdi. Først findes den ønskede forskydning, ved ud fra vektorkortet, hvorefter værdien af den nærmeste pixel findes ved et ”afhængigt” (dependant) teksturopslag i den oprindelige tekstur.

29 I den givne implementering er valgt, at en pixel er samplet når dens alpha-værdi er over 0.98. Dette er en problematik der kun er relevant hvis pixels kan være delsvist samplede, hvilket sker ved brug af f.eks. antialiasing.

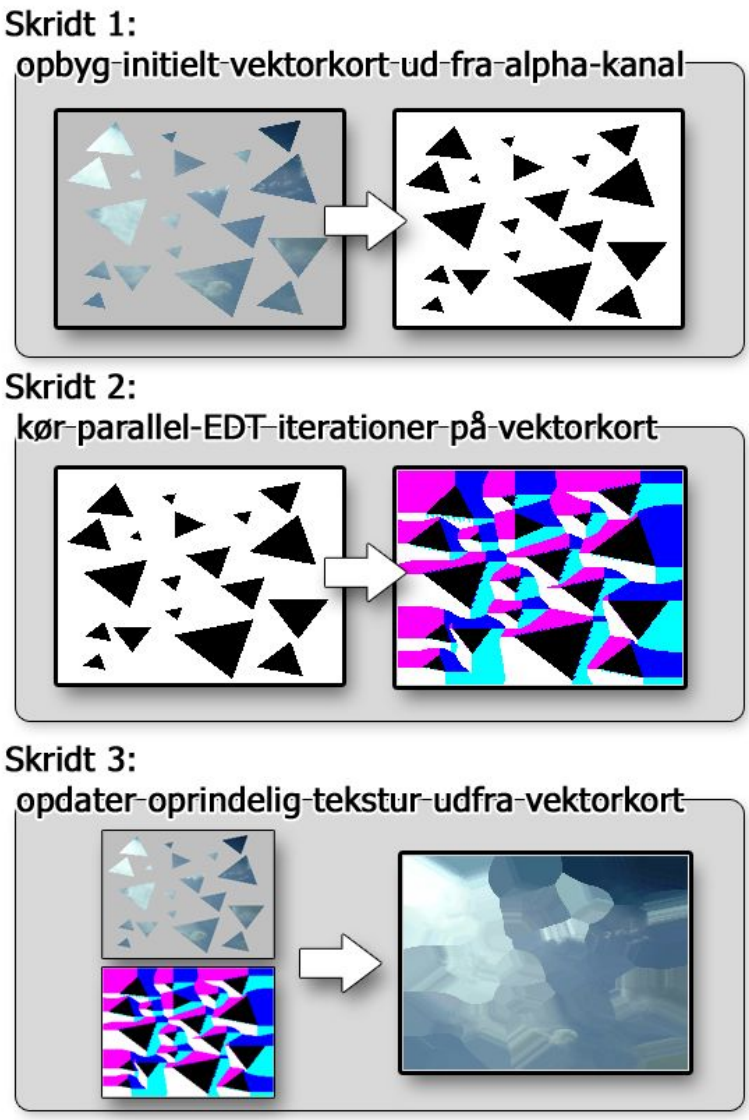


Illustration 6.26:
Renderingen foregår i tre skridt, der hver har en separat shader. Afstandene i afstandskortet er her tegnet direkte, hvilket giver de sære farver der ses på billedet. Her ses algoritmen kørt til ende.

I den givne implementering anvendes afstanden i vektorkortet ikke, men kun informationen om hvilken pixel der er nærmest. Det ekstra arbejde forbundet med tillige at gemme afstanden er dog minimalt, og er her gjort for generaliseringens skyld. Som beskrevet kan der vælges imellem to varianter af den parallelle EDT – en der ser på fire naboer til en pixel og en der ser på otte. Normalt anses teksturopslag på grafikkort for relativt dyre. Forskellen på hastigheden af en enkelt iteration af de to EDT-varianter er dog ikke ret stor idet teksturen vil blive cachet efter første opslag. Derimod konvergerer 8-nabo varianten hurtigere end 4-nabo udgaven, hvilket er væsentligt for hvor mange iterationer vi ønsker at foretage. Da der er et vist overhead ved udførelse af hver iteration, bliver forskellen imellem de to varianter yderligere mindsket.

I den foreliggende implementering er vektorkortet gemt som 4x16bit floatingpoint. Kun de tre af værdierne benyttes dog, og af dem er den ene, afstanden, ikke reelt anvendt. Det er dog ikke muligt

at allokere teksturer af 2x16bit i OpenGL, hvorfor for meget hukommelse allokeres. Anvendelsen af floatingpoint giver mulighed for at foretage en fuld EDT på de størst mulige teksturer i OpenGL (på nuværende tidspunkt 4096x4096). Er målet dog at lave under 256 iterationer (f.eks. hvis ingen objekter er længere væk fra et andet objekt end $\sqrt{256^2+256^2}$), kan det helt undgås at bruge floatingpoint-teksturer. I stedet kan afstande gemmes i en almindelige farvebuffer. For ikke-floatingpoint buffere er mulighederne for konfigurationer flere, og det ville være muligt at allokere f.eks. en 2x8bit buffer – hvilket er minimum for at den givne opgave kan udføres. Samlet set betyder det, at en implementering med almindelige farvebuffere vil spare hukommelse såvel som give en væsentlig hastighedsforbedring, idet mindre data skal hentes for hver nabosampling i algoritmen. Der er her en ganske betragtelig potentiel hastighedsforbedring.

Det har længe været et problem at lave algoritmer i OpenGL, der itererer imellem to teksturer, da den anvendte løsning til "render-to-texture" har været meget langsom. Normalt anvender man en pbuffer der både kan fungere som framebuffer, og bindes som tekstur. Problemet er, at hver pbuffer har sin egen OpenGL-kontekst – og at skift imellem disse tager lang tid. I implementeringen af denne algoritme er dog anvendt et "workaround" der fjerner denne flaskehals, ved at anvende en double-buffer pbuffer, og iterere imellem front- og back- bufferen. Dette er teknisk set ulovligt men er bredt accepteret blandt grafikkort-producenter – og det virker *[GPGPU04]*. Et alternativ til denne fremgangsmåde, der dog kun er tilgængeligt i beta-drivere, er at anvende OpenGL-udvidelsen "Frame-Buffer-Objects" (FBO). FBO'er er opbygget på en måde der i højere grad giver driveren mulighed for at foretage optimeringer i overførsel af data til og fra grafikkortet. Da resten af prototypen dog anvender pbuffers, og problemerne med disse alligevel omgås, er det valgt at beholde anvendelsen af disse. Fremtidige implementeringer bør dog undersøge denne mulighed.

6.9 Ikke benyttede optimeringer

Der er i det ovenstående truffet en række valg, der har betydning for metodens hastighed og præcision. Idet det er en ny metode, og et af målene er at forbedre kvaliteten af silhuetten, er det her søgt at finde den bedst mulige approksimation. Præcision er derfor gået forud for hastighed, men flere steder er det muligt at dreje metoden mod at være hurtigere, på bekostning af præcision. Nogle af optimeringerne er generelle forbedringer der blot kan anvendes på denne metode også: Andre er meget metodenære, og giver mulighed for at justere metoden imellem hastighed og præcision. Disse optimeringer vil dog generelt lede til unøjagtigheder. Overvejelserne vil dog blive præsenteret her, i henhold til fremtidige anvendelser af metoden, hvor hastigheden kan være mere kritisk. I dette afsnit beskrives en række af de valg der er truffet, samt diverse muligheder for at gøre metoden hurtigere, og deres konsekvenser.

Under søgningen over en kant, beregnes for hvert punkt en ny view-vektor, givet ved en vektor til øjepunktet. For kanter i en vis afstand, vil disse vektorer være næsten identiske, hvorfor det giver mening at approksimere med samme vektor ved alle beregningerne. Dette ville være en betydelig besparelse da beregningen foretages ofte, og inkluderer en normalisering. Ved tilpas store afstande kan beregningen af vektoren helt udelades, og vertex->øjepunkt kan erstattes med øjets view-vektor. Dette svarer fuldstændig til OpenGLs mulighed for at skifte imellem lokale og ikke-lokale beregninger af lyset på en overflade.

Idet det observeres, at normalerne "næsten" varierer lineært hen over en kant, ville det være en

mulighed erstatte den tunge beregning af grænsetangenterne med en lineær interpolation, samt en normalisering per kant-punkt. Fremgangsmåden ville være en lidt dårligere approksimation til subdivision fladen, men være op til tre gange hurtigere, idet kun positionen udregnes, mens tangent-beregningerne kan springes over. Fordi normalerne i endepunkterne af kanten angives i forhold til grænsefladen, vil en lineær interpolation give mening når der ikke er stor forskel på normalerne – dette vil være tilfældet for de fleste modeller med få polygoner. Det er altså en optimering der bør overvejes afhængigt af naturen af de flader der visualiseres.

Idet vi anvender normalmapping, er det unødvendigt også at finde og tegne normalerne til kanten. Normalerne skal anvendes i søgningen, men der er ingen grund til at gemme dem til senere brug. I den foreliggende implementering tegnes de dog sammen med modellen, for at kunne foretage en sammenligning imellem vertex, per-pixel- og normalmap-belysning.

Som prototypen er implementeret nu, underinddeles alle gennemsøgte kanter lige meget. Det ville give god mening at ændre denne fremgangsmåde, og anvende et variabelt antal iterationer, afhængigt af den ønskede præcision. Et mål for den nødvendige nøjagtighed kunne være forskellen på normalerne i endepunkterne af kanten. Et andet rimeligt estimat ville være den projicerede størrelse af f.eks. kanten eller objektet som helhed

Ved søgningen over en kant, anvendes kun halvdelen af punkterne. Der kunne derfor spares en del beregninger ved kun at regne på de punkter der blev benyttet. Algoritmen til beregning via kant-opslag er dog i forvejen ganske hurtig. Yderligere baserer den sig på sekventielt gennemløb af arrays, hvilket er hurtigt på moderne processorer, da de ofte vil cache de næste par elementer i hvert array. Den tid der potentielt spares risikerer således at tabes igen pga. mindre orden i beregningerne.

En mulighed der kunne anvendes på systemer hvor hukommelsesforbrug ikke er en flaskehals, ville være at beregne alle underinddelte kanter på en gang. På nuværende tidspunkt beregnes punkterne til en kant umiddelbart inden der søges efter silhuetpunkter på pågældende kant. En implementering der beregnede alle nødvendige kanter på en gang, og herefter søgte efter silhuetpunkter, kunne potentielt give bedre cache-performance. Som beskrevet i *[SCRHÖDER3]*, genbruges tabellerne med vægte til hver beregning, hvorfor der skulle være god mulighed for at cache opslaget. Dette er indtil videre et postulat, men bør undersøges nærmere.

En generel optimering, som vil give god mening at anvende i sammenhæng med denne metode, er at begrænse søgningen til kanter der er i viewfrustum. En følgevirkning af dette ville dog være, at antagelsen om, at enhver trekant har enten ingen eller to silhuetpunkter ville blive brudt. Dette kunne dog trivielt løses, ved tilsvarende kun at triangulere og tegne trekanter, der var helt eller delvist i frustum.

Der anvendes ingen tidslige optimeringer i metoden. Der er flere grunde til dette. Vi ønsker ikke at cache kant-subd data, idet det vil gøre ressourceforbruget større. I silhuet-søgningen kunne anvendes data fra sidste billede til at finde den nye silhuet. Algoritmen præsenteret i *[HALL]* er dog væsentligt mere kompleks end brute-force algoritmen anvendt nu, og der er tvivl om hvorvidt den altid vil finde nye silhuetter. Idet den yderligere har en worst-case køretid større end den nuværende brute-force algoritme, anses det ufordelagtigt at foretage et skift.

6.10 Detaljer omkring den implementerede prototype

I dette afsnit præsenteres de væsentlige dele af programmet der ikke er beskrevet i de foregående afsnit. Prototypen er implementeret i C++, og der er anvendt en række funktioner, specifikke for Microsoft Windows. Yderligere er der gjort brug af en række kode-biblioteker til forskellige formål.

- SDL, til håndtering af vinduer og diverse input
- OpenGL, til hardware-accelereret tilgang til 3D-tegne funktioner
- GLU, for diverse udvidelser til OpenGL
- Nvidia CG, C for Graphics, som wrapper for OpenGL's shader-assembler sprog. Dette muliggører udvikling af højniveau shaderkode. Fordelen frem for OpenGL's eget GLSL er alene, at CG er ældre, og af den grund mere stabilt.

Det er i implementeringen forsøgt at anvende tankegangen fra *[GAMMA]*, og danne generaliserede interface-klasser for det meste funktionalitet. Dette giver let mulighed for at ændre den specifikke implementering, uden at skulle foretage ændringer i de funktioner der anvender klasserne. Set i retrospektiv kunne et par designvalg dog med fordel ændres. Primært kunne typesikkerheden forbedres, eksempelvis ved brug af metoder i stil med ”policies” *[ALEXANDRESCU]*.

6.10.1 Silhuetsøgning

Silhuetsøgning er overordnet set blot en måde at visualisere en model. Metoden er derfor implementeret som en renderingsklasse, med sædvanlig interface til *render()* og *upload()* kald, som beskrevet i afsnit 6.3. Det eneste der skal gøres for at tegne en model via silhuetsubdivision, er således at anvende denne renderklasse, i stedet for rendering via f.eks. Immediate-Mode eller Display Lists. I pre-render-fasen beregnes silhuetpunkterne til fladen. Under rendering dannes silhuetkurver, og fladen trianguleres. Al beregning associeret til silhuetberegning er isoleret i klassen ”cSilhuetteCalc”, der også gemmer resultaterne af disse beregninger. Selve underinddelingen af en kant er implementeret i klassen ”cEdgeSubdivider”, der tillige beregner og gemmer de nødvendige tabeller. Udover de nødvendige tegnefunktioner i cSilhuetteRenderer, er tilføjet et par funktioner der er anvendelig til at se hvordan metoden fungerer. Specifikt er der funktioner til tegning af normalmappet, silhuetkurven, samt de kant der søges på – disse er dog kun tilgængelige internt i klassen.

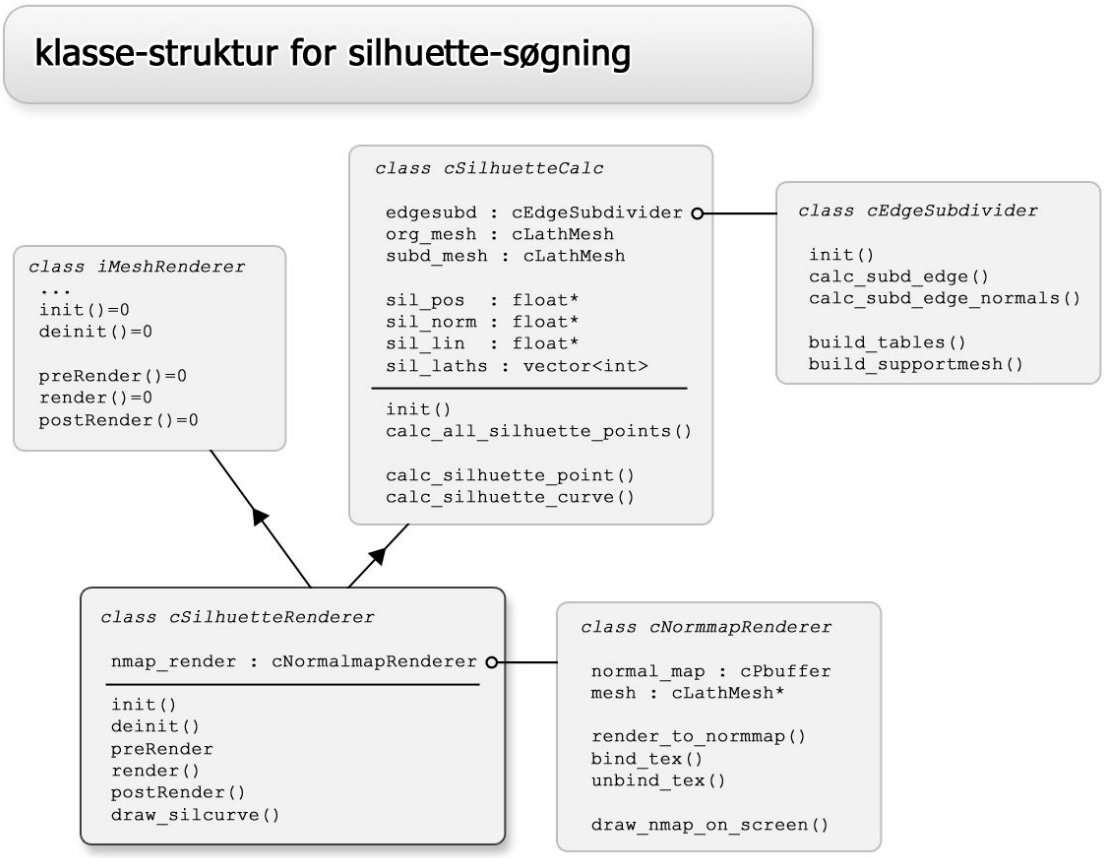


Illustration 6.27:
Den egentlige beregning af silhuetpunkter sker i cSilhuetteCalc::calc_silhuette_point(..). CsilhuetteRenderer anvender disse punkter, og danner og renderer den endelige trekant-model. Underinddeling af kanten til brug i silhuetsøgningen sker i cEdgeSubdivider, der tillige gemmer de nødvendige tabeller.

6.10.2 Shadere

Normalmapping og per pixel belysning er implementeret via shadere, der baserer sig på Nvidia's ”C for Graphics”. Til at håndtere shadere, er der konstrueret en klasse, ”cShaderContainer”, der håndterer alle implementeringsspecifikke funktioner – herunder skift i states og parametre til shaderne. Idet det antages at der kun anvendes en enkelt OpenGL-kontekst, eller, hvis der der er flere, at de anvender samme ressourcelager, er klassen implementeret som en singleton *[GAMMA, ALEXANDRESCU]*. Shadere implementeres som klasser. En shader kender filnavne på vertex- og pixel-shadere, samt navnene på de parametre der kan sættes i shaderen. Yderligere er den i stand til at sætte disse parametere.

6.10.3 Subdivision

En ”Subdivider” er en bekvemmelighedsklasse, der samler funktioner i et let ”lav en subdivision-iteration på model”-kald. En subdivider kan foretage en subdivision-iteration, og beregne grænse-positioner og -normaler.

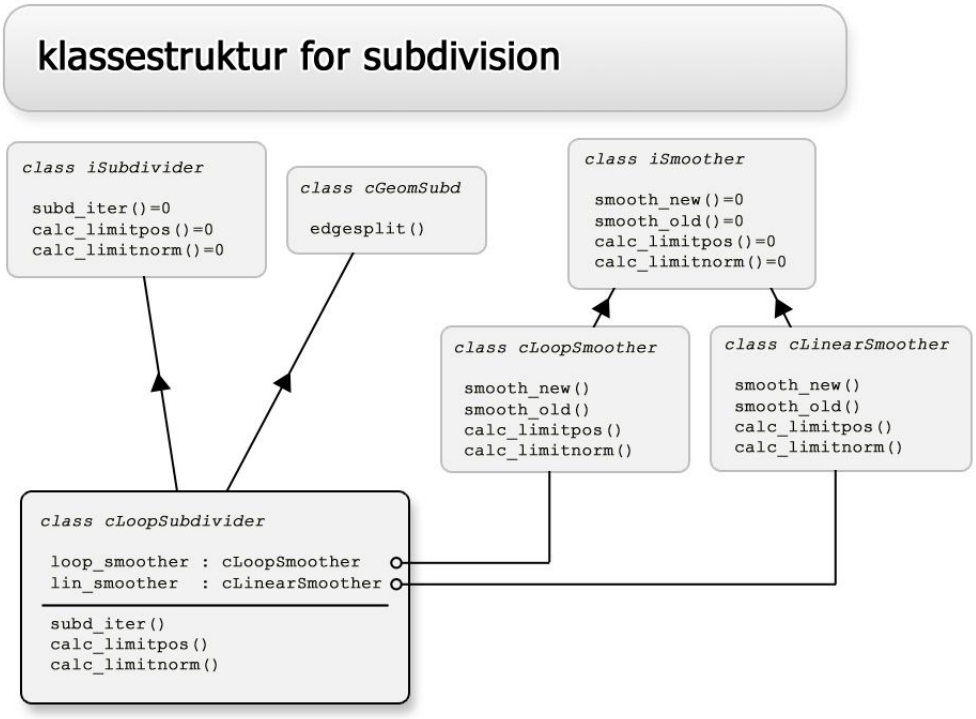


Illustration 6.28:

For ”smoothere” gælder det tilsvarende, at de skal kunne udglatte nye punkter og gamle punkter. Den egentlige beregning af grænsepunkter og normaler ligger i denne klasse. Beregningen af grænsepunkter ligner så meget den for udglatning af gamle punkter, at kun vægtene skal ændres.

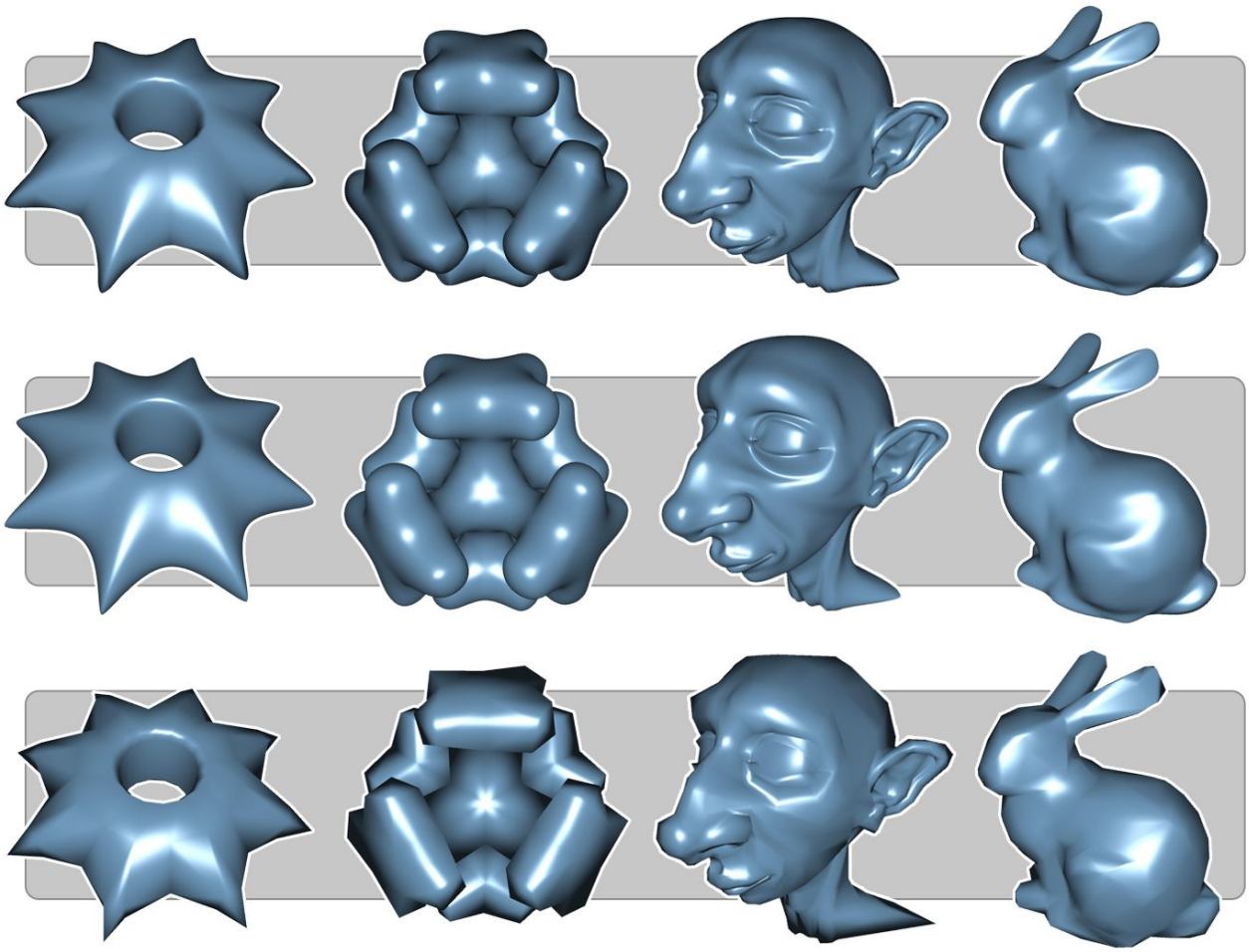
Al geometrisk underinddeling af modeller er isoleret i cGeomSubd. Dette er ud fra en betragtning om, at mange subdivision-metoder underinddeler modeller ensartet: Først underinddeles modellen, så udglattes henholdvist nye, og dernæst gamle punkter. Det er derfor let at samle funktioner til at underinddele en model, og markere hvilke punkter der er nye, og hvilke er gamle.

7 Resultater og diskussion

I dette afsnit vil blive vist billeder lavet med den implementerede prototype på subdivision silhuetsøgning. Resultaterne vil yderligere blive vurderet kvalitativt såvel som kvantitativt, hvorefter artefakter ved metoden diskuteres, og mulige løsningsforslag gives.

I dette afsnit vises billeder også af modeller der aldrig har været tiltænkt anvendt til subdivision[30]. En sammenligning af udseende skal således ske med modellernes grænseflader – ikke i forhold til hvordan de "rigtigt" skal se ud. I mangel på bedre materiale, er modellerne alligevel vist her.

Alle test er afviklet under Microsoft Windows 2000, på en Intel P4 3.2GHz med et Nvidia Geforce 6600GT der anvender forceware drivere version 71.89.



Figur 7.1:
Sammenligning imellem forskellige typer visualisering. Øverst ses silhuetsøgning med normalmapping, i midten grænsefladen, og nederst den grove model med perpixel belysning.

30 Idet Loop-subdivsion anvendes, og denne er en approksimerende metode, "skrumper" modellerne efter hver iteration. Det betyder, at subdivision anvendt på polygon-reducerede modeller, som f.eks. stanford kaninen, giver flader der er mindre end de oprindelige modeller.

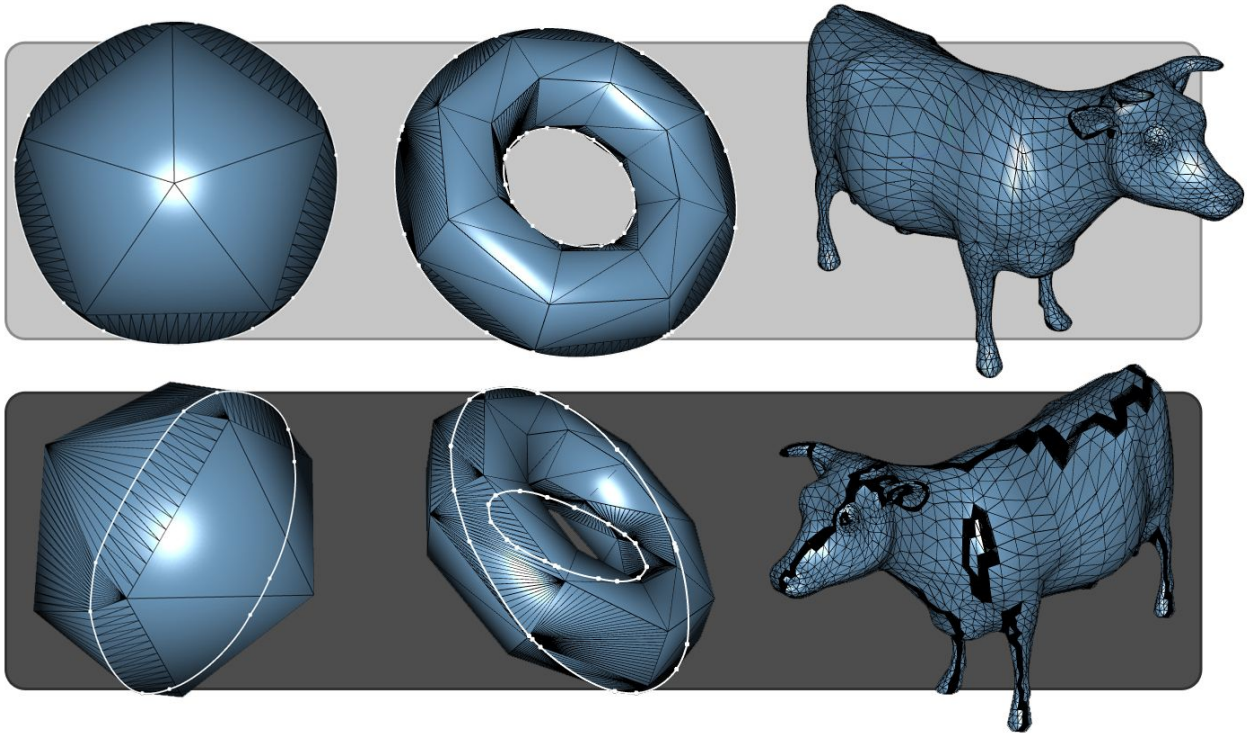
7.1 Sammenligning af visualiseringer

I dette afsnit vil der blive foretaget sammenligning imellem traditionelle måder at visualisere subdivision flader, og visualisering via silhuet-søgning kombineret med normalmapping.

I det følgende vil betegnelsen ”per pixel” belysning blive brugt. Med betegnelsen menes, at lysberegningen foretages for hver pixel der tegnes. Til forskel fra standard OpenGL-belysning hvor lyset udregnes per vertex, og interpoleres lineært henover en flade, interpoleres i stedet vertex-normalerne samt retnings- og lys-vektorer.

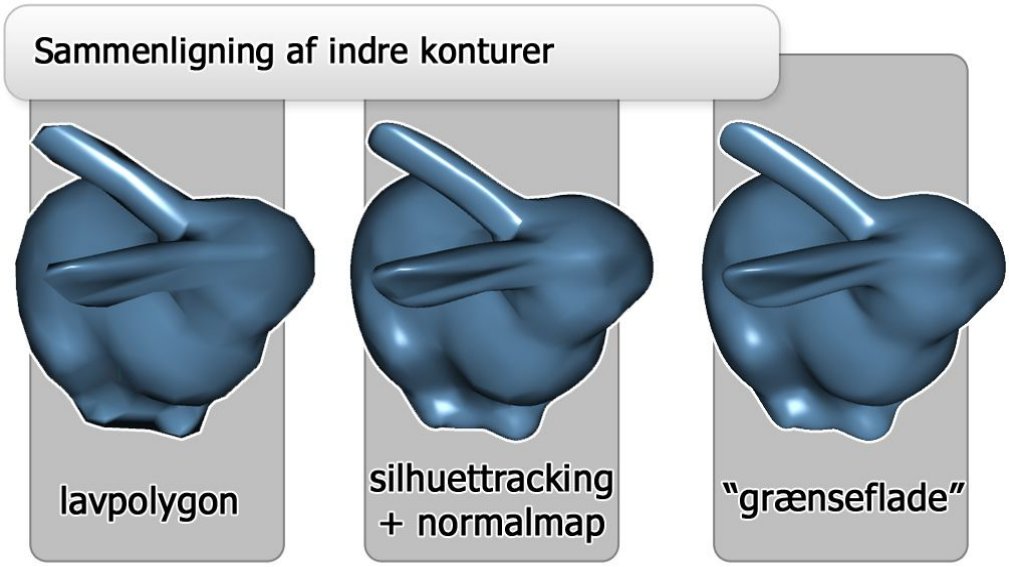
Modeller der i det følgende betegnes ”grænseflade”, er modeller underinddelt til subpixelniveau for billedet – yderligere underinddeling vil således ikke gøre en visuel forskel. Medmindre andet er angivet, er lavpolygon-flader skubbet til grænsefladen. Der anvendes ligeledes grænsenormaler.

Figur 7.1 viser øverst en række modeller visualiseret via silhuet-søgning. Det ses, at silhuetterne fremstår væsentligt blødere end den tilsvarende visualisering af den grove model alene. Det observeres også, at anvendelsen af normalmapping forbedrer shadingen af fladerne, og at især højlyset fremtræder mere korrekt. Der er yderligere umiddelbart ingen fejl at se på grund af den meget lokale triangulering. Overordnet set er ligheden med grænsefladen meget stor, og kun lokale afvigelser i højlyset afslører fladen som en approksimation. Sammenlignes med lavpolygon-udgaven ses det, at alle skarpe kanter, langs såvel modellens silhuet samt indre konturer, er væk. På figur 7.2 ses wireframe-visualisering af af en række modeller. Det ses, at trianguleringen foretages meget lokalt omkring silhuetten, mens resten af modellen tegnes med sin oprindelige detaljeringsgrad. Selvom den tabelbaserede kant-evaluering beskrevet i afsnit 3.2 kræver at kun det ene endepunkt er irregulært, hvilket normalt kræver underinddeling af modellen, tegnes den oprindelige model. Idet punkterne sendes til grafikkortet per frame, giver dette en væsentligt hastighedsforbedring sammenlignet med at tegne modellen efter en subdivision-iteration.



Figur 7.2:
På begge billeder ses trianguleringen af silhuet-underinddelingen. Øverst vises objektet fra det øjepunkt silhuetten er dannet ud fra – nederst fra en anden vinkel, så underinddelingen bedre kan ses.

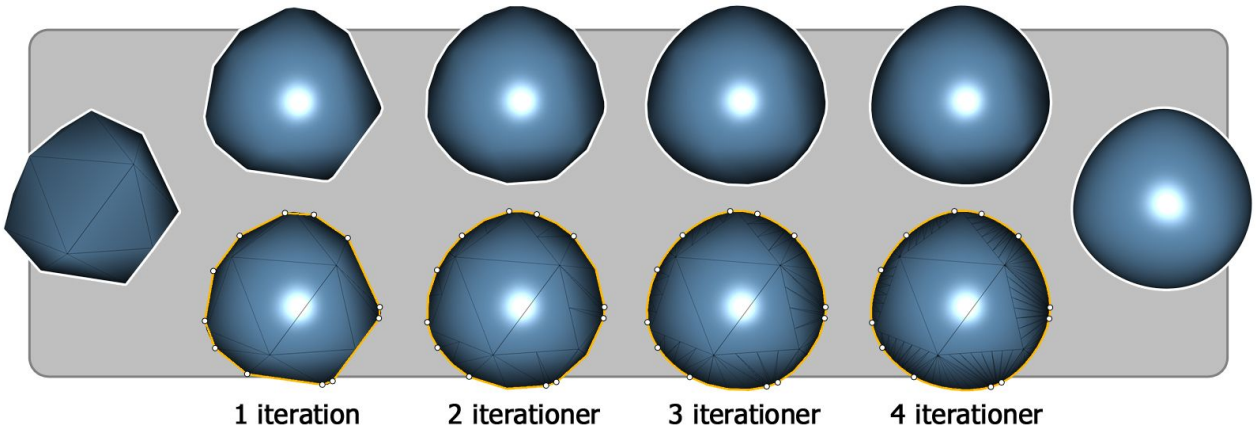
Ses igen på figur 7.2, bemærkes det, at silhuetkurven for hver trekant dannes med samme antal evalueringer. Dette betyder at antallet af punkter varierer meget i forhold til kurvelængden. Sædvanligvis løses denne type problemer ved at buelængde-parameterisere kurven *[GRAVESEN]*. I dette tilfælde vil en simpel heuristik for antallet af evalueringer, givet ved afstanden imellem endepunkterne dog med stor sandsynlighed være tilstrækkelig.



Figur 7.3:
Det ses her, at metoden også håndterer "indre" konturer, hvor en del af objektet overlapper en anden del af objektet.

På figur 7.3 ses en model, hvor en del af geometrien overlapper resten. Den overlappende geometri er dels ørerne på kaninen, men også den del af kroppen forrest, der skygger lidt for poterne. Det ses, at metoden korrekt håndterer denne situation, hvilket ellers har været et problem for lignende metoder, se afsnit 4.1.6.

Antal evalueringer af silhuetkurverne, samt antal skridt i søgningen efter silhuetkanter, er de to væsentligste måder at styre præcision og hastighed af metoden. På figur 7.4 ses metoden evalueret med forskellige grader af præcision. De viste modeller er visualiseret ved at ændre præcisionen i såvel kantsøgningen som den efterfølgende beziér-kurveevaluering. En fordel ved metoden er, at disse to parametre ikke er koblede, hvorfor det altid er muligt at få silhuetten til altid at fremstå blød, selvom den er dannet fra upræcise punkter på kanten. Her ændres dog de to parametre sammen, da den resulterende visualisering hermed svarer til en given subdivision iteration af fladen. Det ses på figuren, at de fundne silhuetpunkter ikke ændrer sig ret meget, hvilket indikerer at der uden større tab, kan anvendes kun få iterationer i kantsøgningen.



Figur 7.4:
Konvergens for metoden. Ved søgning kan antal underinddelinger styres, og ved tegning kan antal underinddelinger på silhuetten ændres. Her vises tilfældet hvor der anvendes lige mange underinddelinger for alle kanter. Begge rækker viser silhuet-søgning, nederst vises blot yderligere trianguleringen.

7.2 Hastighed og effektivitet

I dette afsnit præsenteres kvantitative resultater for den implementerede prototype, og disse vurderes i forhold til andre måder at visualisere modellerne. Til de følgende test er der taget udgangspunkt i "rigtige" modeller, med forskellig detaljegrad (antal trekanter i basismodellen). Resultaterne er målt fra et enkelt observationspunkt, som gennemsnit over 5-10 sekunder.

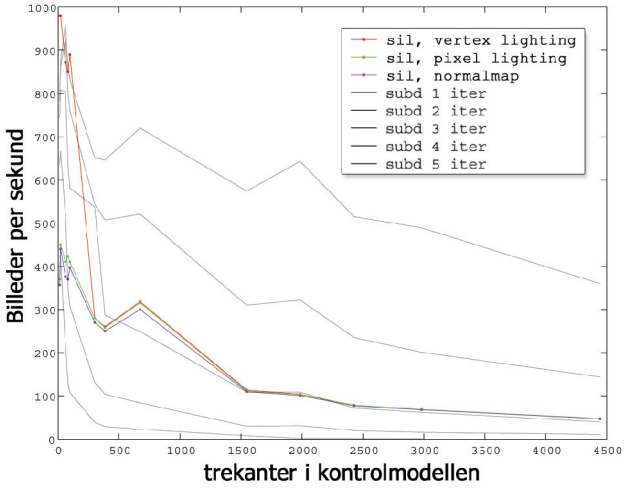


Illustration 7.5:
Her vises hastigheden hvormed en række modeller med varierende detaljegrad, kan visualiseres med silhuet-søgning og almindelige subdivision. Silhuetsøgningen foretager 5 iterationer.

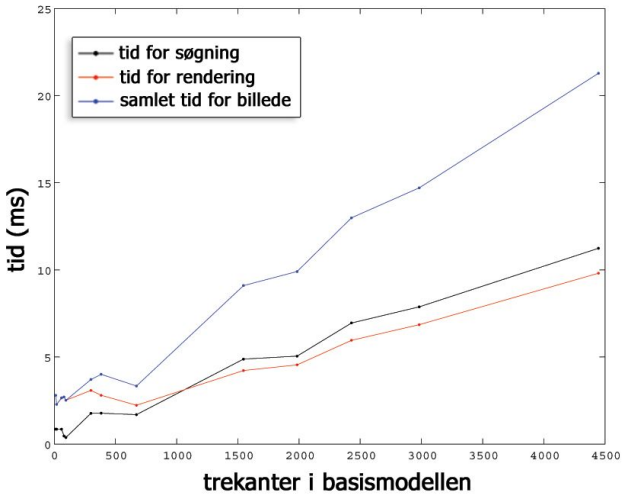


Illustration 7.6:
Her vises en graf over tidsforbruget under tegning af en model med silhuetsøgning. Det ses at søgningen tager ca. lige så lang tid som at tegne modellen. Silhuetsøgningen foretager 5 iterationer.

På illustration 7.5 sammenlignes visualisering ved silhuet-søgning med sædvanlig visualisering af samme model efter et antal subdivision-iterationer. For modeller der som udgangspunkt består af mange trekanter, er det kun nødvendigt med få iterationer, før yderligere underinddeling ikke vil gøre en visuel forskel. Sammenligner vi silhuetsøgningen med den subdivision-iteration den selv søger på, her 5. iteration, ses det at metoden for selv ret simple modeller er væsentligt hurtigere. En anden betragtning der kan gøres er, at hastigheden for visualisering med silhuetsøgning, med den anvendte computerkonfiguration, omtrent svarer til visualisering af modellen med 3 subdivision iterationer.

Det ses at ydeevnen for metoden ikke direkte afhænger af en models detaljegrad. Dette hænger sammen med, at en fuld søgning på en kant kun foretages når endepunkterne peger i hver sin retning. Effektiviteten af søgning på en model er således afhængig både af modellens geometri, og observationspunktet. Et gennemsnit af silhuetsøgning over mange synsvinkler på en model, burde således give en mere direkte sammenhæng imellem antal trekanter og søgetid.

Af illustration 7.6 fremgår det, at selve kant-søgningen tager omkring halvdelen af den tid det tager at visualisere en model, mens tegnefunktionen står for resten. Yderligere ses det, at søgningen er tilnærmelsesvist lineær i forhold til antallet af trekanter i basismodellen. Det samme gælder for tegning af de trekanter der genereres af metoden. Tegnefunktionen er som nævnt i afsnit 6.5.3 ikke optimeret, og skønnes at kunne optimeres omkring 50% - hvilket ud fra disse observationer vil gøre metoden 25% hurtigere.

På illustration 7.7 sammenlignes effektiviteten af kantsøgning med varieret præcision. Som forventet er der en eksponentiel sammenhæng imellem antal underinddelinger på en kant, og søgetiden. Umiddelbart kunne godt undre, eftersom søgningen er lineær i forhold til antallet af iterationer: Idet søgetiden i et segment tager konstant tid, vil søgetiden være direkte proportional med antallet af søgninger. Forklaringen skal findes i, at vi beregner alle punkter på en kant, uden hensyntagen til om de bruges eller ej. Dette valg blev truffet ud fra en betragtning om cache-venlighed af beregningen, se afsnit 6.4. Da antallet af punkter på en kant stiger eksponentielt med antallet af iterationer, gør også tiden det tager at beregne dem.

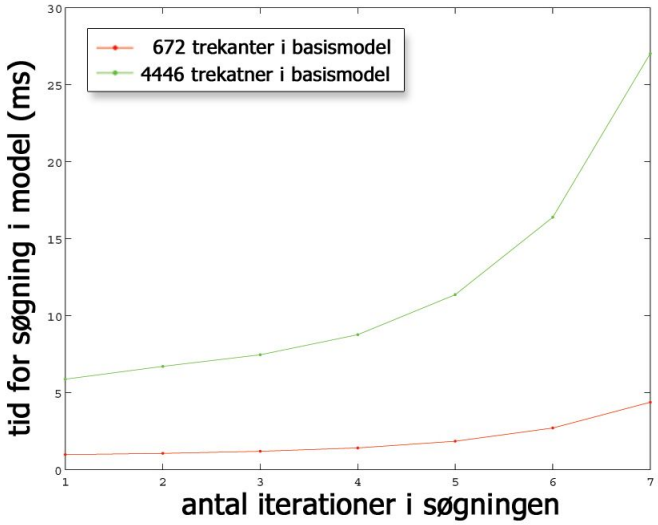


Illustration 7.7:

7.2.1 Ydeevne af parallel euklidisk afstandstransformation

Der ses her meget kort på hastigheden ved afvikling af den parallelle Euklidiske afstandstransformation, der anvendes til at afhjælpe samplingproblemer i normalmappet. Den her testede EDT-variant, er 8-nabo udgaven.

Ser vi på illustration 7.8, observeres det som forventet, at den parallelle EDT er næsten lineær i henhold til antallet af iterationer der køres. Tilsvarende ses det på illustration 7.9, at EDT-beregningen er meget tæt på lineær i forhold til antallet af pixels der beregnes – og således med teksturstørrelsen kvadreret.

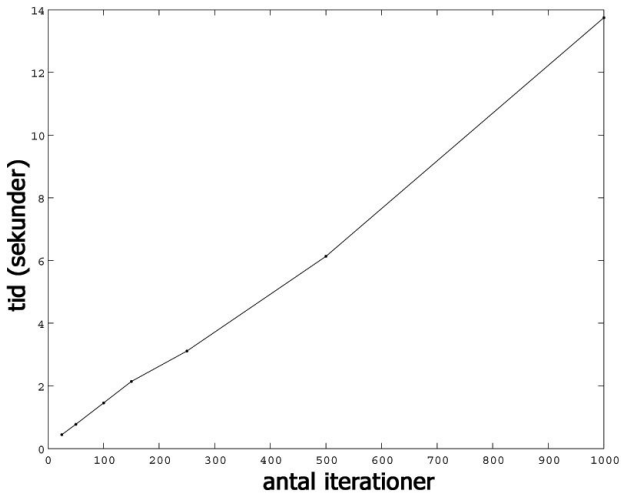


Illustration 7.8:
Variabelt antal iterationer vises her for en tekstur af størrelsen 1024x1024.

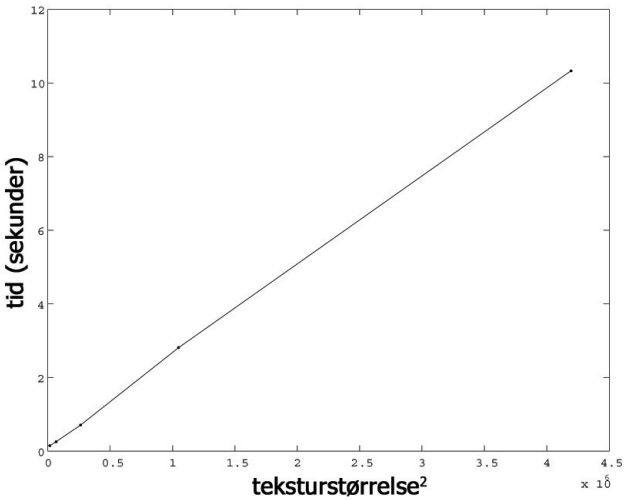


Illustration 7.9:
Variabel teksturstørrelse vises her for 200 EDT-iterationer

I de følgende afsnit beskrives afvigelser der opstår som følge af visualisering af subdivision flader med silhuetsøgning.

7.3 Popping-artefakter

Betegnelsen ”popping” bruges til beskrivelse af tidslige diskontinuiteter. Dette har eksempelvis været et problem ved klassiske Level-of-Detail metoder, hvor en model af en given detaljegrad udskiftes med en anden – gøres dette ikke meget omhyggeligt, ses dette skift som et spring i visualiseringen af modellen.

Der er i den implementerede prototype ikke gjort noget for at undgå popping – hovedsageligt fordi metoden ikke udviser dette problem i nogen væsentlig grad. Som det vil blive forklaret i dette afsnit, sker de fleste ændringer af overfladen glidende. Idet silhuetten hovedsageligt ændrer sig ved rotation af et objekt, opfattes denne ændring kun vanskeligt. Der er dog en række situationer der kan resultere i popping, og disse vil her blive beskrevet. I relation til silhuet-søgning, er der to typer popping der vil blive fremhævet som væsentlige:

- ”silhuette-popping” situationer hvor silhuetten fra det ene billedet til det næste ændrer sig markant.
- Flade-popping. Dette vil ske hvis selve fladen hopper fra en position til en anden. Dette vil ofte give anledning til ændring i den observerede lysberegning, men er et separat fænomen.

7.3.1 Silhuette-popping

Ideelt set skulle der ikke forekomme popping når silhuetkurver bevæger sig imellem trekanter. Tilstandsskiftet fra når en trekant indeholder en silhuet og når den ikke gør, sker nemlig når silhuetkurven ligger helt op til et hjørnepunkt. Dette betyder, at trianguleringen giver en helt flad trekant, der er visuelt identisk med trekanten uden triangulering, pga. den lineære interpolation af vertex-attributter. Idet vi yderligere anvender per-pixel evaluering til beregning af lys, gør antallet af vertices ingen forskel. Vertex-attributter interpoleres tilsvarende lineært henover kanterne, hvorfor data anvendt per pixel vil være uforandret uanset tilføjelsen af lineære vertices[31].

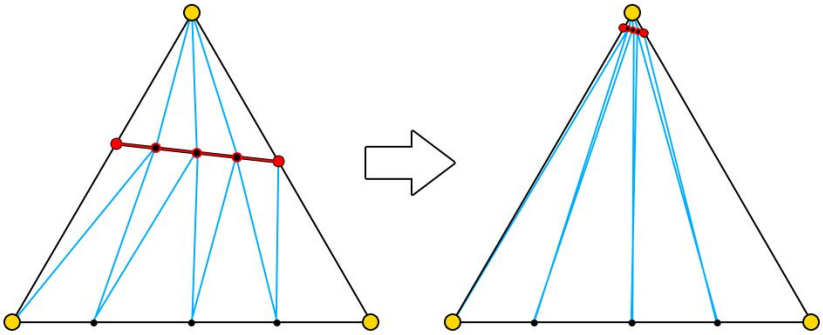
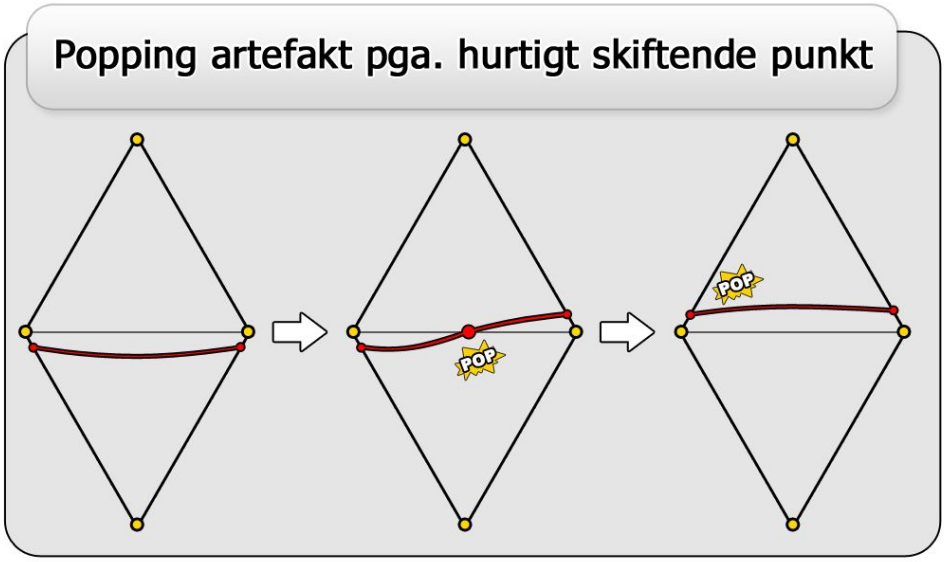


Illustration 7.10:
Trianguleringen af silhuet-trekanter vil normalt gøre, at skiftet imellem silhuet-trekant og almindelig trekant er usynligt.

Der er dog en række fejlkilder der kan resultere i silhuette-popping. Det største problem i henhold til silhuette-popping er, at silhuet-kurverne ikke dannes korrekt. Silhuet-punkter ligger på grænsefladen og kan gøres vilkårligt nøjagtige, ved at øge antallet af iterationer i søgningen. Silhuetkurver imellem silhuetpunkterne er derimod ikke garanteret at ligge på grænsefladen, eller at have normaler der er vinkelrette på øjevektoren. Valget af PN-triangle metoden til at danne kontrolpolygonen for den kubiske kurve, giver yderligere ikke en silhuet-kurve der ligger på subdivision-fladen. I de fleste tilfælde betyder dette, at silhuetten fremstår blød men en smule fladtrykt – meget i stil med PN-triangles-metoden, se figur 7.13 s. 128. Idet silhuetkurverne ikke ligger på grænsefladen, kan der opstå fejl når de punkter der danner kurverne ændres. Dette sker eksempelvis når silhuetten bevæger sig ”på tværs” henover en kant. I langt de fleste tilfælde vil silhuetten i denne bevægelse, skære kanten en enkelt gang, se figur 7.11. Reelt betyder dette, at et silhuetpunkt vil bevæge sig hurtigt henover kanten. Sker ændringen hurtigt nok observeres den som et ”pop”. Ideelt set skulle denne situation ikke give problemer, idet silhuetkurven skulle dannes korrekt uanset hvilke punkter der blev valgt. Pga. unøjagtighederne ved PN-triangle approksimationen, kan silhuetkurven dog have en anden form, hvilket resulterer i popping.

31 Under implementeringen har det været vigtigt at lave robuste beregninger for at undgå afrundingsfejl i interpolationen af vektorer. Det væsentligste middel er at undgå vektor-normaliseringer i vertex-shaderen, hvorved større værdier interpoleres henover fladen – med mindre relative fejl i de resulterende vektorer til følge.



Figur 7.11:
Silhuetkurven er markeret med rødt. Dette er i virkeligheden ikke et pop, men et silhuetpunkt der bevæger sig meget hurtigt henover kanten. Silhuetterne kan dog se meget forskellige ud, med det ekstra punkt indsat (fordi pn-triangle approksimationen ikke giver kurver på fladerne).

Et andet problem der er direkte relateret til PN-triangle approksimationen er, når et identificeret silhuet-kurvestykke ikke reelt danner den tiltænkte silhuet. Fordi silhuetkurverne henover hver flade ikke ligger på grænsefladen, kan der opstå silhuetfejl såvel som "rigtig" popping. På illustration 7.12 ses et eksempel på dette. Silhuetpunkterne findes her korrekt, men fordi den dannede kurve ligger inden i fladen, i stedet for på fladen, kan et stykke af fladen ses, der ikke burde være en del af objektets silhuet.

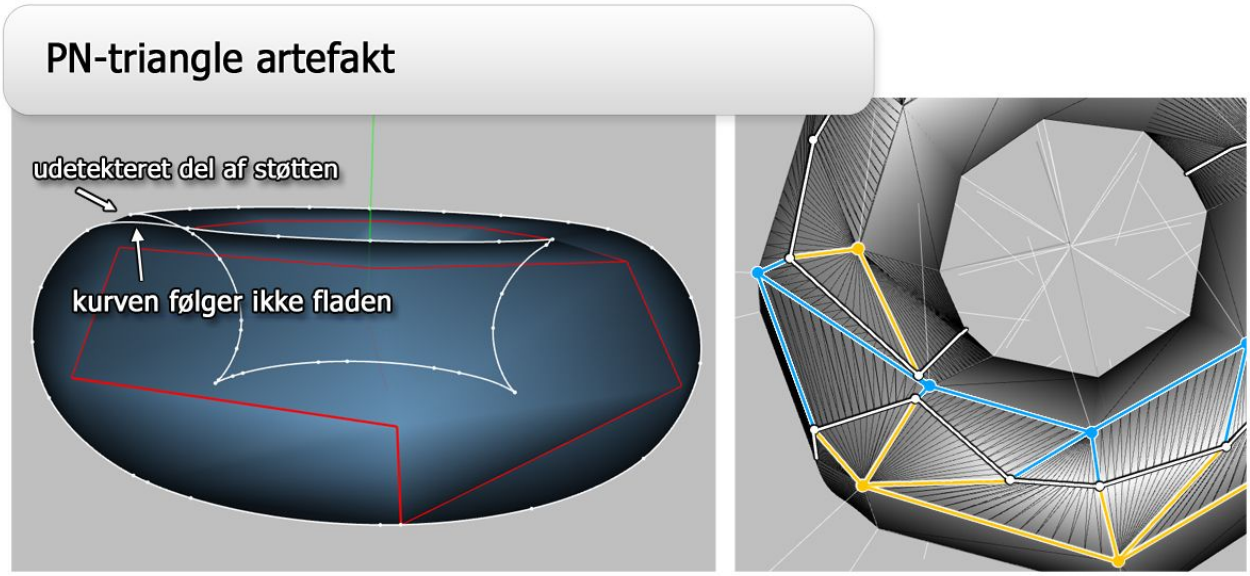
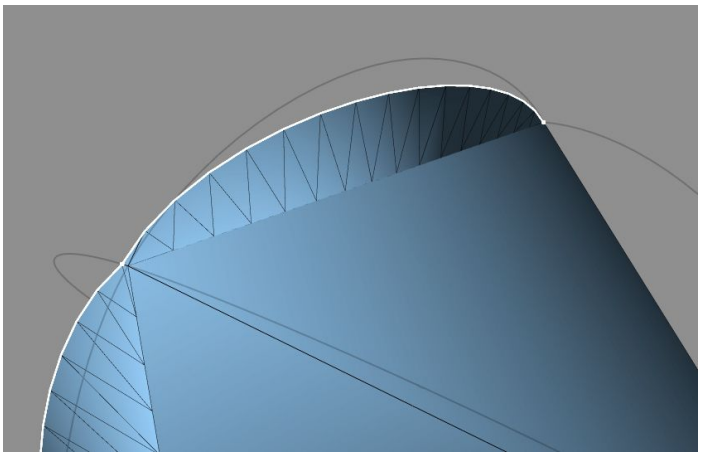


Illustration 7.12:
Til venstre ses et eksempel på en silhuetfejl forårsaget af at pn-triangle approksimationen ikke giver kurver der ligger på subdivsion-fladen. Til højre ses de gennemsøgte kanter i området – gul markerer dele af kanterne der vender imod øjepunktet, blå markere de der vender væk. Silhuetten er på begge billeder givet ved den hvide kurve.

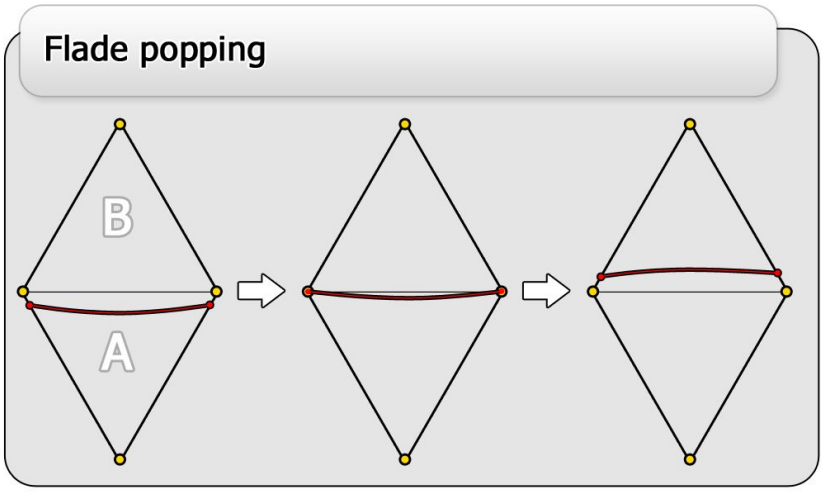
7.3.2 Flade popping

I dette afsnit behandles popping af fladen, der ikke giver anledning til ændring af silhuetten. Dette er relevant, fordi lyset på fladen vil blive beregnet anderledes, ved et skift i overfladens geometri. Idet normalerne er givet separat fra geometrien, i et normalmap, vil der for et givet punkt på fladen ikke ske nogen ændring af parametrene i lysberegningen. Ved et pludselig skift i geometrien, vil et punkt på fladen, dog kunne blive tegnet en smule forskudt på skærmen. Det vil i det følgende blive gennemgået hvornår sådanne skift kan forekomme.



Figur 7.13:
På billedet ses en flade hvor silhuetten er lige ved at skære en kant på modellen. De subdividede kanter til modellen ses som grå wireframe-kurver over modellen. Det ses at silhuetkurven, tegnet med hvid, afviger fra den subdividede kant.

En situation hvor fladepopping vil forekomme, er vist på figur 7.14. Fladen ændrer sig her, fordi trianguleringen af trekanterne skifter brat når silhuetkurven forlader trekant A. Medmindre silhuetkanten er helt flad, vil den øverste del af trekant A ændre højde når den falder tilbage til ikke at blive underinddelt af silhuetkurven. Idet der ikke korrigeres for højdeforskellen ved normalmappingen, f.eks. via reliefmapping, vil det give et lille spring i lysvektorerne, samt hvilke dele af trekanten der tegnes. Dette giver et decideret pop. På figur 7.13 kan ses hvor skarp en kant trianguleringen kan give, imellem en silhuetkurve og en kant i modellen. Yderligere er det tydeligt, at afstanden imellem den grove flade, og silhuetkurven er stor nok til at give ændringer i de vektorer der indgår i lysberegningen – primært øjevektoren og lokale lysvektorer. En indledende underinddeling af modellen kan mindske disse afvigelser, se afsnit 7.6. Alternativt kunne det antages at såvel øjepunkt som lys var uendeligt fjerne. Denne approksimation anvendes ofte for at øge hastigheden af shading. Det vil dog kun løse problemet for vektorerne, ikke fejlsamplingen af normalmappet.



Figur 7.14:
Medmindre subdivision fladen imellem trekant A og B er helt flad,vil overfladen springe når silhuetkurvern passerer kanten direkte på tværs.

I langt de fleste tilfælde vil silhuetkurven dog bevæge sig ”skævt” henover kanten imellem to trekanter, hvorved problemet forsvinder. Trekanten vil stadig ændre form, men det sker nu i en glidende bevægelse. Situationen vist her er grænsetilfældet for figur 7.11, hvor det midterste punkt slet ikke opstår. Problemet optræder dog også når punktet springer hurtigt henover kanten, men bliver værre jo tættere på denne situation vi kommer.

En løsning på begge de her beskrevne problemer, er at lave en rigtig underinddeling af kanten imellem de to trekanter – og således alle kanter der støder op til en trekant som silhuetten løber igennem. Dette kræver at trianguleringen af trekanterne propagerer et niveau ud. Dette vil dels resultere i at flere trekanter tegnes, men også komplicere trianguleringen idet hver enkelt trekant ikke kan betragtes separat. En ”nabo”-trekant skal således trianguleres efter hvor mange af dens naboer er silhuettrekanter.

Den måde kanten underinddeles er væsentlig. Idet vi har udviklet en måde til hurtigt at subdivide en kant (afsnit 6.4), ville det være nærliggende at ”skubbe” kanten til grænsefladen. Så længe silhuetkurven tilnærmes med PN-triangle metoden, risikerer vi dog, at rigtig subdivision af kanten giver yderligere artefakter. Grunden er, at kanten imellem de to trekanter vil være mere korrekt end silhuetkurven. Ses således igen på figur 7.14, og antages øjepunktet at være i bunden af billedet, risikerer man, at kanten imellem A og B dækker for silhuetkurven på det sidste af de tre billeder. En bedre fremgangsmåde til den nuværende implementering, vil således være at bruge samme PN-triangle approksimation til beregning af kanten. Fordi de to kurver vil udvise samme ”forkerte” form, vil der ikke være problemer ihht. præcisionen af silhuetkurven.

En sidste situation der kan give anledning til pludselige ændringer i overfladen, ses på illustration 7.15. Her roterer silhuetten, og trianguleringen af trekanten ændrer sig så snart silhuetten passerer den nederste venstre vertex. Modsat ovenstående situationer, er der ingen umiddelbar løsningsmodel for denne situation. Til gengæld er problemet ikke så slemt som de andre, idet overfladen ændrer sig mindre markant. Problemet er dog lettere synligt for meget aflange trekanter.

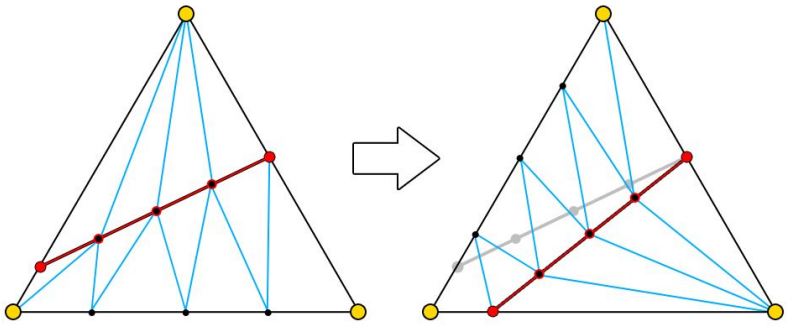
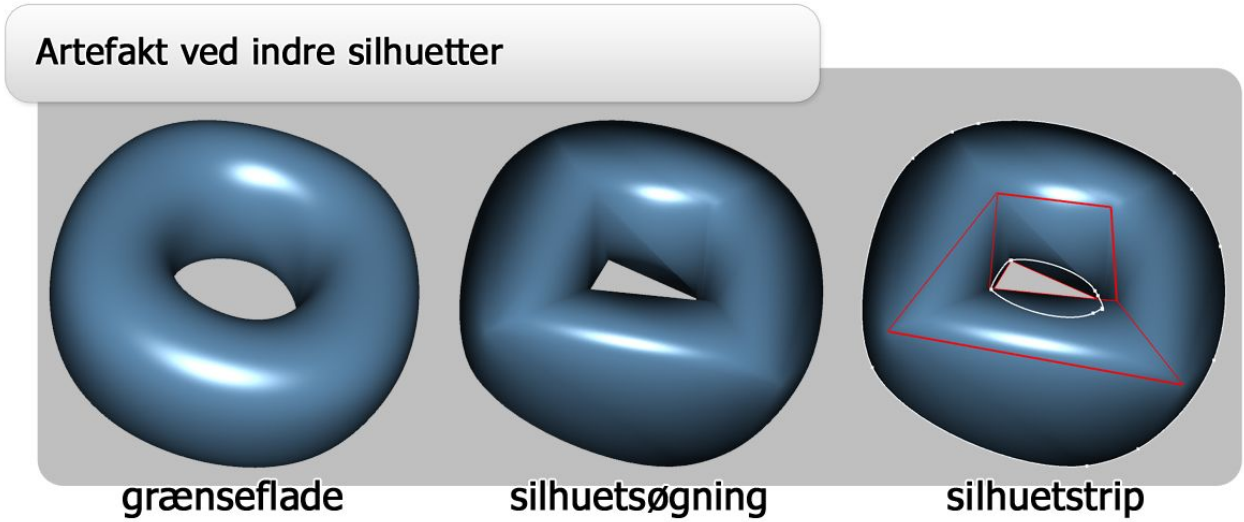


Illustration 7.15:
Når en silhuet roterer imellem to kanter, skifter trianguleringen af kanten. Dette vil i de fleste tilfælde give et mindre spring i overfladen.

7.4 Artefakter i forbindelse med konkave konturer

Med betegnelsen ”konkav” kontur, menes her et område af fladen hvor en eller begge af fladens principale krumninger er positive *[GRAVESEN]*. Konkave konturer udgør et problem fordi de forskyder fladen indad (i forhold til normalen). Problemet opstår når en naboflade til en silhuettrekant ikke forskydes indad, og derfor kan skygge for den korrigerede kontur. Dette kan forekomme på så simple modeller som en bøjet cylinder. Resultatet er, at den bløde silhuet skjules, og den kantede nabokant ses i stedet.



Figur 7.16:
Det ses at konkave silhuetter ikke dannes korrekt. Til højre ses naboer til silhuettrekanter markeret med rød. Underinddeling af disse kanter vil i de fleste tilfælde løse problemet med trekanter fra udetaljerede model, der kommer til at danne den indre silhuet.

I de fleste tilfælde kan problemet løses ved at underinddele silhuet-strip kanterne, markeret med rødt på figur 7.16, på samme måde som forklaret i afsnit 7.3.2 **<om fladepopping>**. Dette er en forholdsvist triviel tilføjelse til metoden, der dog vil få underinddelingen til at propagere et enkelt niveau ud. Der er dog ingen garanti for, at det altid vil være silhuet-strip kanten der giver problemet, men det er tilfældet i langt de fleste situationer.

7.5 Artefakter ved triangulering

Trianguleringen af silhuetfladerne kan i særlige tilfælde være årsag til småfejl i visualiseringen. Som netop beskrevet, kan der for lokale områder af en flade være problemer med triangulering af nabotrekanterne. I visse situationer er der risiko for, at selve trianguleringen fra subdivision-kurven bliver synlig udenfor silhuetten. Det betyder igen at den silhuet-korrigerede kant ikke indgår som en del af objektets silhuet. Når det sker, vil trekanten dog være meget lille, idet den ligger på silhuetten. Fejlen er således svag, men resulterer alligevel i en højfrekvent ændring i den ellers bløde silhuet.

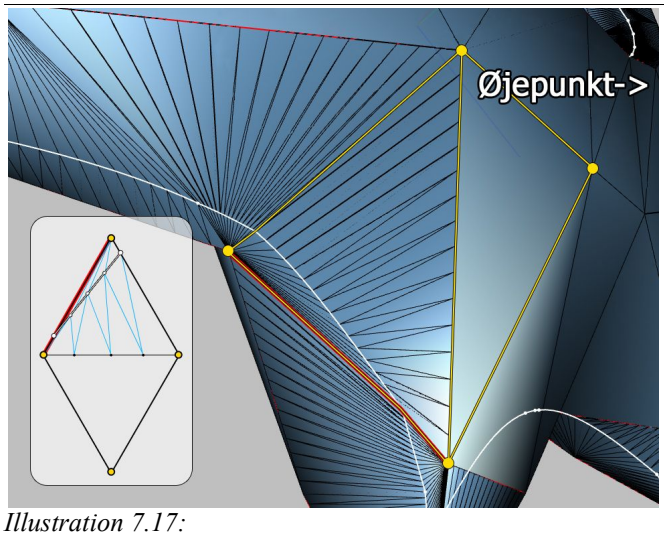


Illustration 7.17:

For meget tynde modeller med høj krumning, er det yderligere muligt, at silhuetkurven kommer til at skære en fremadrettet polygon, således at fladen skærer sig selv. Et eksempel er en meget tynd torus, hvor den indre silhuet let kan gå i gennem den ydre, hvis denne ikke er en del af silhuetten, og således tegnes i sin ”grove” form. Dette vil dog som regel ikke ses hvis backface-culling anvendes, men kan være et problem i ekstreme tilfælde.

Trianguleringen af silhuettrekanter dannes som beskrevet i afsnit 6.5.3 på en måde der giver risiko for pixel-udfald. Dette er kun observeret i ganske få tilfælde, og fjernes næsten helt ved anvendelse af fullscreen-antialiasing.

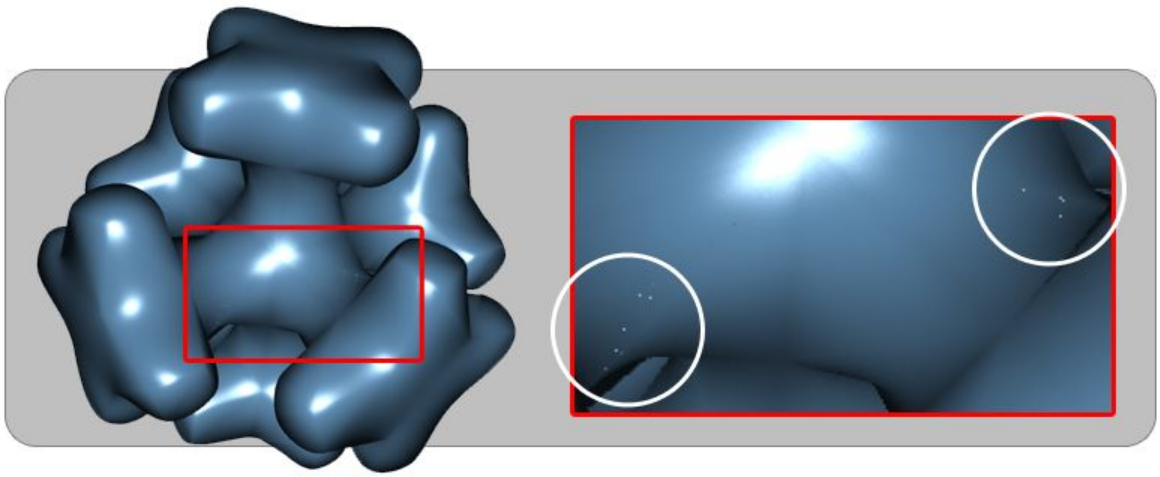


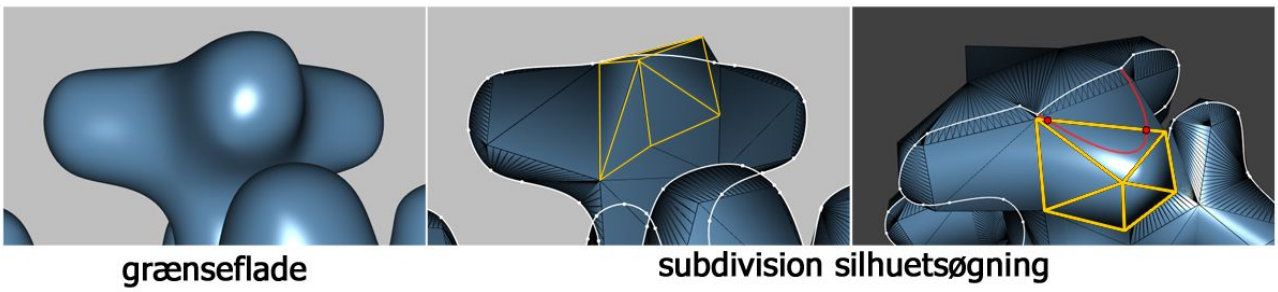
Illustration 7.18:
Pixeludfald, hvor enkelte pixels ikke bliver tegnet, optræder til tider pga. trianguleringen.

7.6 Fejl i søgningen

Søgningen efter silhuetpunkter på en kant er ikke perfekt. Hvis der er givet et og kun et silhuetpunkt, er metoden garanteret at konvergere imod dette punkt. Der er dog situationer hvor silhuetpunkter ikke bliver fundet.

Et eksempel på dette er, når to silhuetter går igennem den samme kant. I dette tilfælde vil normalerne til endepunkterne ikke pege i hver sin retning, hvorfor kanten slet ikke bliver undersøgt. Situationer som det ikke er lykkedes at konstruere eksempler på er, at tre eller flere silhuet-punkter ligger på samme kant. Det er ikke muligt korrekt at danne silhuetkurver ved kun at se på kanter, hvis silhuetten kan skære hver kant flere gange. Grunden er, at silhuetkurverne kan skære hinanden inde i en trekant, hvilket gør det ikke-entydigt at forbinde silhuetpunkterne korrekt. En efterfølgende triangulering vil således ikke have nok information til på korrekt vis at forbinde silhuettens kantpunkter.

Idet søgningen baserer sig på, at normalerne til den dannede subdivision flade skal vende i hver sin retning i forhold til øjepunktet, findes der tilfælde den ikke fanger. Et eksempel ses på figur 7.19, hvor kanten markeret med rødt, ikke opdages. Til højre på figuren ses det, at silhuetten ikke skærer kanten der ellers burde være en del af silhuetten. Årsagen er, at normalerne til fladen langs kanten peger væk fra kameraet et sted inde på kanten, men imod kameraet i begge endepunkter. Dette sker fordi subdivision-fladen så at sige er samplet for få gange, hvorfor vi ikke finder en række silhuetpunkter.



Figur 7.19:
Der ses her en situation hvor metoden fejler fordi normalen svinger to gange.

Et workaround er således, at foretage en enkelt indledende subdivision på modellen, se illustration 7.20. Dette bringer samtidig hele modellens udseende tættere på grænsefladen, hvilket forbedrer blandt andet kvaliteten af normalmappingen, samt præcisionen af silhuetkurverne. At foretage en indledende subdivision iteration er ikke en decideret løsning, og vil ikke ændre situationer hvor de to skæringspunkter ligger på samme halvdel af kanten. I de fleste tilfælde vil det dog være en tilstrækkelig forbedring til at rette op på de observerede problemer.

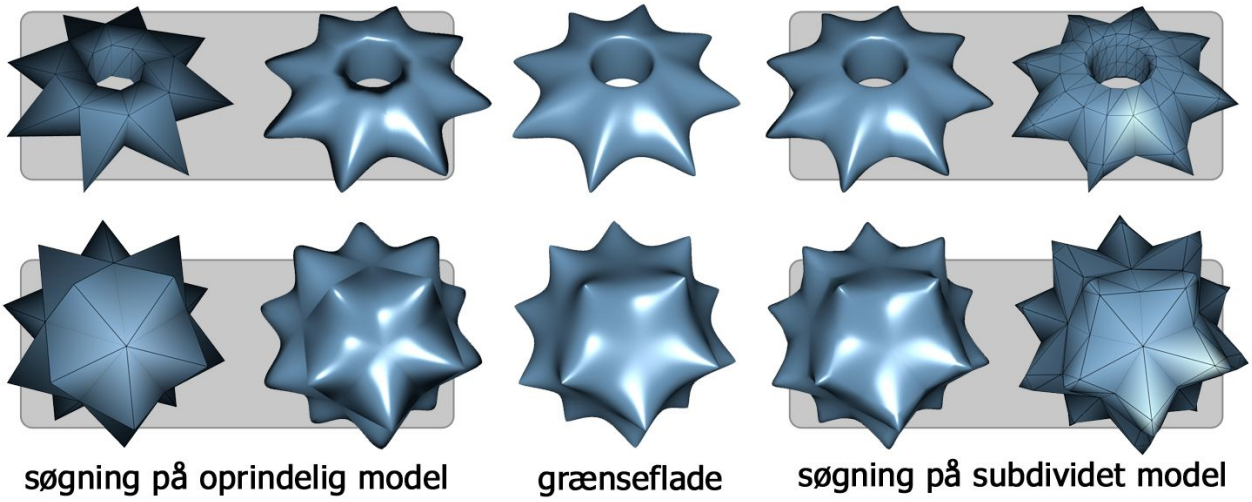


Illustration 7.20:
For modeller med få polygoner, kan det være nødvendigt at underinddele modellen en enkelt gang, inden den kan anvendes med silhuetsøgning.

7.7 Belysning

I dette afsnit gives en kort oversigt over afvigelser i belysningen i forhold til subdivision grænsefladen. Idet den primære korrektion af belysningen på fladerne anvender normalmapping, er problemerne også hovedsageligt relateret til denne teknik. I forprojektet til dette projekt blev problematikken i henhold til brug af normalmapping gennemgået i detaljer, og der henvises dertil for en nøjere gennemgang.

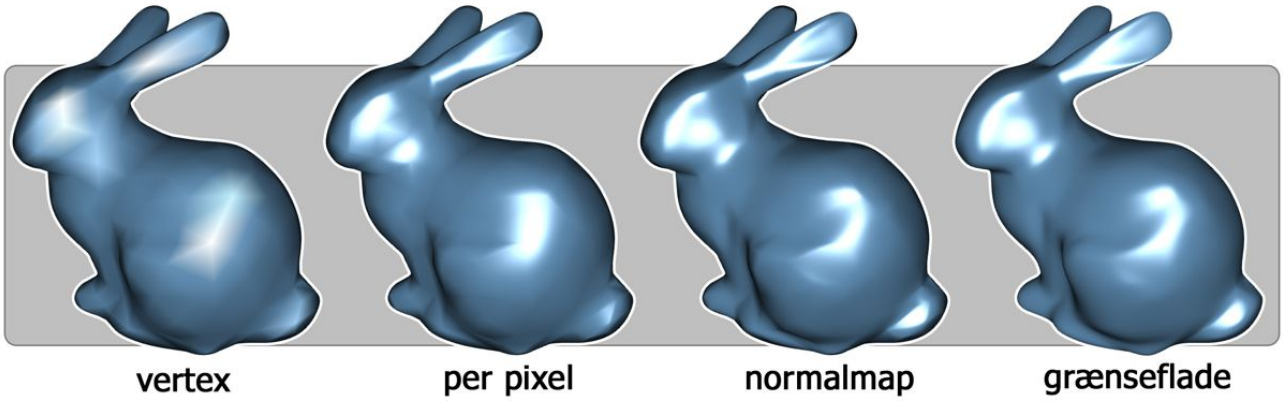
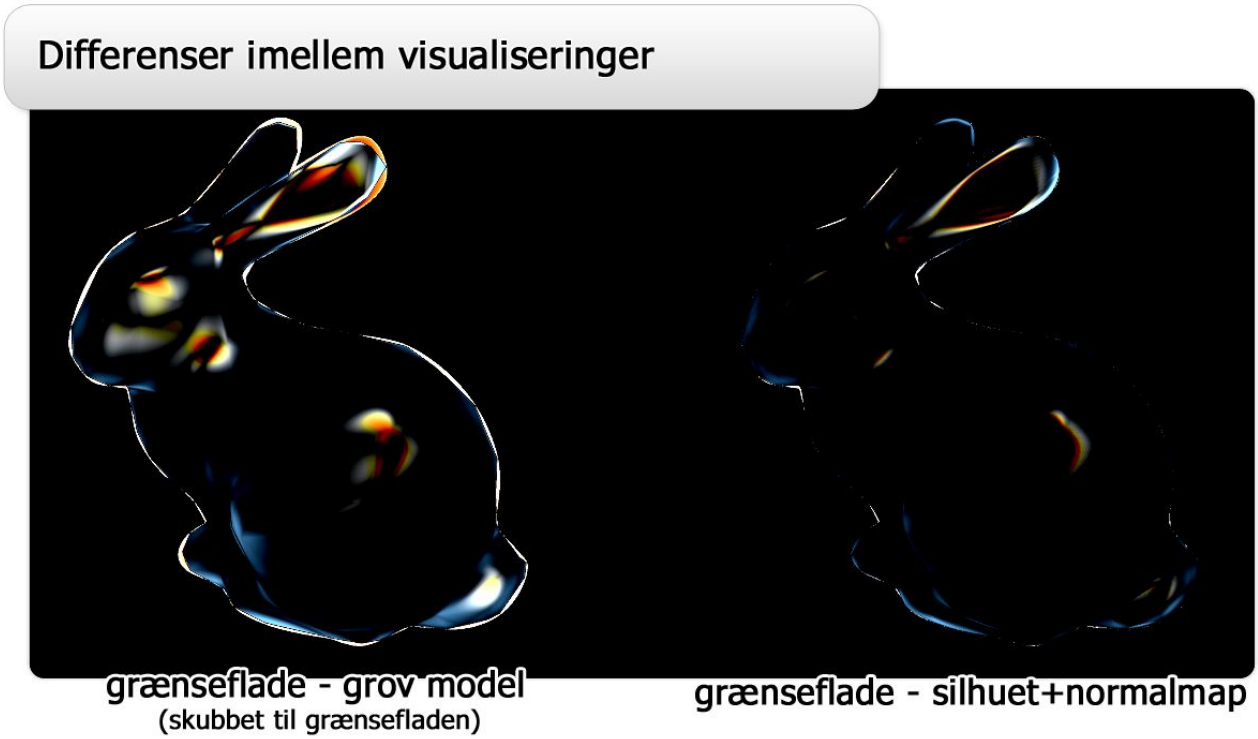


Illustration 7.21: Sammenligning af lysberegninger på den silhuet-korrigerede model.

Det væsentligste problem ved brugen af normalmapping er, at der ikke tages højde for, at den approksimerede flade er forskudt i forhold til den renderede flade. Teoretisk set betyder det at lyset bliver beregnet for skæringspunktet imellem øjevektoren og den grove model, i stedet for øjevektoren og grænsefladen. Hvis forskydelsen er stor, bliver forskellen i denne beregning det tilsvarende, idet ikke alene fladeparametrene bliver forkerte, men også de vektorer der indgår i lysberegningen. I praksis betyder afvigelsen, at der sker et hurtigt skift i belysningen, fordi der henover en kant opstår en diskontinuitet i den første-ordens afledede til de øje- og lys-vektorer der indgår i lysberegningen. På visualiseringen ses dette mest tydeligt i højlyset, som et knæk hvor det passerer en kant.

I områder nær silhuetten hvor silhuetsøgningen bringer fladen nærmere grænsefladen, bliver afstanden, og dermed fejlen, tilsvarende mindre. På figur 7.22, ses det, at belysningen nær silhuetten er mere korrekt, end for den grove model.



Figur 7.22:
Her ses to differenser imellem visualisering af en Stanford-kanin. Det ses at fejlene ved såvel silhuetten som det indre af modellen mindskes betydeligt ved brug af den præsenterede metode.
(kontrasten i billedet er forøget, og de mørkeste farver er fjernet)

7.7.1 Normalmapping-artefakter

Brugen af normalmapping gør, at metoden arver en række skavanker herfra. Problemet omkring kanterne af samplingen i normalmappet er løst på en bedre og mere elegant måde, end i forprojektet. De øvrige problemer der opstår pga. stor forskydning imellem kontrolpolygon og grænseflade, er ikke behandlet. Problematikken omkring afstand imellem den renderede og den approksimerede flade, er netop det som parallax-mapping, relief-mapping og tilsvarende metoder forsøger at rette, se afsnit 4. Idet silhuet-søgningen gør at fladen hele tiden ændrer sig, er det dog ikke trivielt at anvende disse metoder, da de antager statiske modeller. Et workaround ville som nævnt i afsnit 5.6.1 være at anvende en korrigerende algoritme i det indre af modellen, og udfase denne i nærheden af silhuetten. Dette vil dog efterlade silhuetten ukorrigeret, hvorfor en løsning der i stedet korrigerer for silhuet-forskydelsen bør undersøges.

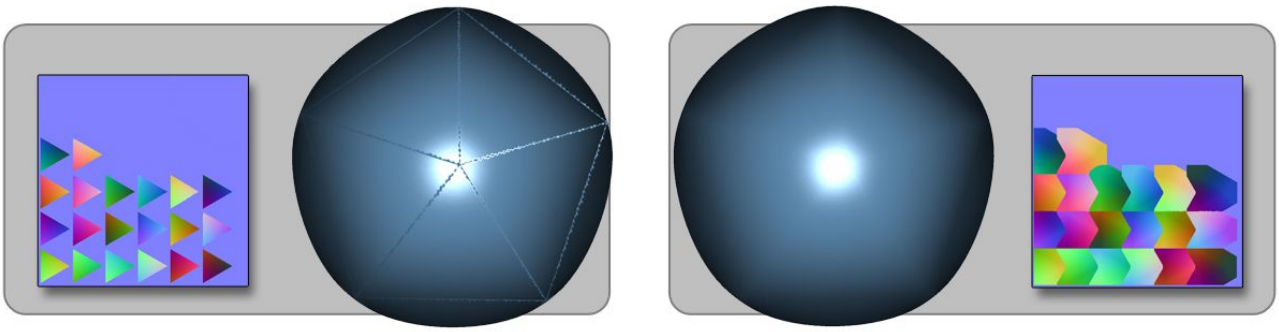


Illustration 7.23: Løsningen via EDT fjerner effektivt samplingfejlene langs teksturgrænser.

Algoritmen for den euklidiske afstandstransformation (EDT), stiller ikke store krav til hardware, men er i den viste implementering ikke helt hurtig nok til at kunne anvendes per billede. Størrelsen af det bånd der er nødvendigt omkring hver trekant er ret begrænset, hvilket gør det hurtigt at beregne. Det giver således mening at foretage beregningen på grafikkortet frem for at hente teksturen fra systemhukommelse og køre en fuld EDT, der garanterer at alle pixels bliver korrekte. Med brug af EDT er det muligt at anvende mipmapping med metoden – jo flere niveauer mip-mapping der ønskes, des bredere bånd er nødvendigt. I sidste ende sætter afstanden imellem de separate trekanter i uv-atlasset dog en begrænsning på nedsamplingen.

7.8 Skæringer imellem subdivision-flader

Metoden har ikke til hensigt at afhjælpe problemer ved skæringer imellem flader. Der vil derfor, i områder hvor to subdivision-flader skærer hinanden, stadig optræde hakkede kanter. Samme problem gør sig gældende for objekter der skærer sig selv – hvad enten det er pga. konstruktion eller deformation af modellen. Årsagen til dette er, at den indre del af modellerne ikke ændres geometrisk, men at blot lysberegningen ændres via normalmapping. Det betyder, at dybdebufferen bliver markant anderledes i forhold til grænsefladen, hvilket igen resulterer i ”forkerte” skæringer. Der er vist metoder svarende til ”depth-sprites” *[RTR2]*, til udvidelse af parallax-mapping der løser dette problem.

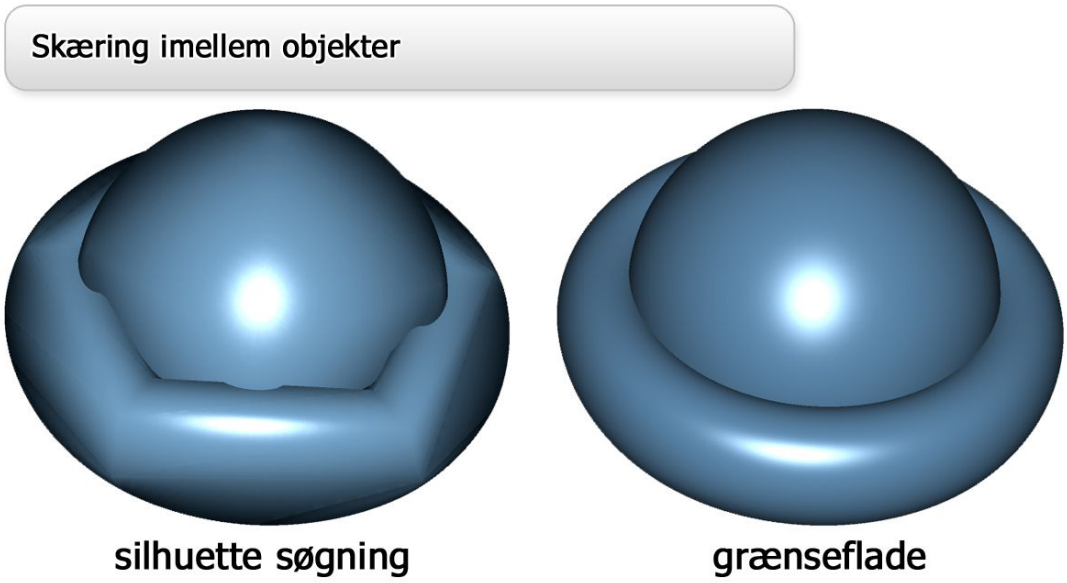


Illustration 7.24:
Metoden giver ikke mulighed for at skære objekter med hinanden. Områder af en model der ikke ligger på en kontur, tegnes som sin lavpolygon-flade, hvofor skæringer ligeledes bliver grove.

7.9 Opsummering

Der er i dette afsnit vist eksempler på billeder frembragt ved brug af subdivision silhuet-søgning. Det er vist at metoden er i stand til effektivt at forbedre silhuetten, og at ligheden med grænsefladen øges væsentligt herved. Der er desuden gennemgået en række forhold der kan give anledning til artefakter, og forslag til forbedringer der kan begrænse disse er givet.

8 Overordnede betragtninger omkring metoden

I dette afsnit vil der ud fra de opnåede resultater, blive foretaget overordnede betragtninger omkring anvendelsen af metoden. Yderligere vil det blive diskuteret, hvilken retning videre arbejde med metoden bør have.

Det er umiddelbart muligt at ændre metoden til, at fungere på flere andre "primale" subdivision-metoder – dette inkluderer metoder der giver mulighed for anvendelsen af punkter og skarpe kanter *[SCHRÖDER3]*. Det er således ikke en direkte udvidelse af metoden, men nærmere en specifikation til en given anvendelse. En udvidelse til åbne modeller vil resultere i yderligere preberegning og større tabeller, men vil ikke tilføje arbejde runtime. Spidse punkter og skarpe kanter kræver desuden mulighed for at afmærke punkter og kanter som værende skarpe, men dette kan umiddelbart tilføjes til lath-datastrukturen og er således let muligt. Det vil altid være relevant at overveje hvilken subdivision metode der bedst passer til ens behov, og anvende denne. Den i dette projekt benyttede subdivisionmetode bør således ikke blot overtages tankeløst – men vælges med tanker på den specifikke anvendelse. Eksempelvis anvendes Catmull-Clark flader i mange modelleringspakker, og vil derfor være et bedre valg i mange situationer.

Den her viste implementering bygger på Loop-subdivision, der er baseret på trekanter. Dette er især en fordel ved dannelsen af silhuetkurver, idet en række forhold kan antages, som gælder for trekantmodeller men ikke generelle polygonmodeller. Dette er dog ikke en væsentlig begrænsning, men betyder at anvendelse på ikke-trekant baserede metoder kræver omstrukturering af disse opgaver.

For silhuetkanter med kun et skæringspunkt, kan præcisionen af søgningen let styres ved at variere antallet af iterationer i søgningen. Den efterfølgende dannelse af konturkurver er sværere at gøre præcis, idet PN-triangle approksimationen ikke til fulde opfylder de nødvendige krav. Det resulterende antal trekanter kan dog kontrolleres direkte ved, at variere antallet af evalueringer for silhuetkurven. En interessant egenskab ved metoden er, at der altid kan dannes bløde kurver uanset præcisionen af de fundne konturpunkter. Idet søgning efter silhuetpunkter og tegning af de dannede trekanter tager omtrent lige lang tid, giver det dog mening at beholde et fornuftigt forhold imellem præcisionen af søgning og den efterfølgende triangulering.

Metoden kan også anvendes til tracking af silhuetkanter til skyggeberegninger via shadow-volumes. Dette er en vigtig pointe, idet det tidligere er blevet nævnt som et argument for ikke at anvende PN-triangles *[CARMACK]*. De beregnede shadow-volumes vil dog ikke kunne anvendes til at kaste skygge på modellen selv, idet dybde-bufferen svarer til den lavdetaljerede model.

Metoden til kontur-korrektion er mere fleksibel end at lægge al geometri statisk på grafikkortet, idet kontrolmodellen frit kan manipuleres inden tegning. Den eneste udfordring i denne sammenhæng er, at der skal foretages en enkelt subdivision-iteration på modellen, enten en gang for alle, eller hver gang modellen ændres. Generelt har hardware bevæget sig i retning af statisk geometri, der processeres per frame. Den her viste fremgangsmåde går imod dette, og foretager de beregninger der skal til på CPUen, for herved at kunne tegne så få polygoner, at overheadet ved at overføre data til grafikkortet ikke bliver et problem. For større modeller kan denne overførsel stadig være en begrænsning, eftersom modellen selv uden underinddeling kan være tung. Generelt vil modeller målrettet imod subdivision dog være af moderat størrelse, hvorfor dette ikke vurderes til at være et problem i praksis.

8.1 Implementering på GPU

I dette afsnit foretages en række overordnede overvejelser omkring muligheden for at afvikle metoden på grafikkortet. Der er en række grunde til at dette er ønskværdigt. Primært undgås kopieringen af silhuettrekanterne fra systemhukommelse til grafikkortet for hvert billede der tegnes. Yderligere er det en generel fordel at flytte egnede algoritmer til grafikkortet, idet CPUen typisk håndterer mange opgaver, og således lettere bliver en flaskehals for en given applikation. Et andet aspekt er, at grafikkortets processor, GPUen, er specialiseret til netop floatingpoint vektoroperationer, som metoden i vid udstrækning baserer sig på. Endelig ser udviklingen af GPUer ud til at forblive hurtigere end for CPUer, hvorfor hastighedsforskellen imellem de to processorer kun vil øges.

Den i dette projekt udviklede metode bygger på tabelbaseret subdivision, der tidligere er vist implementeret på en GPU *[SCHRÖDER5]*. Yderligere er den her viste metode simplere, idet kun kantpunkter betragtes, hvorfor denne del af metoden umiddelbart kan flyttes til grafikkortet.

I den foreliggende anvendelse giver dette dog ikke mening at gøre alene, da kantinddeling kun anvendes i forbindelse med søgninger efter silhuetpunkter. Denne søgning burde dog også kunne implementeres som et GPU-program – analoge løsninger er vist til tracking af silhuetter til brug i skyggekast *[BRABEC]*. Fremgangsmåden skulle være at beregne de relevante silhuet-punkter på GPU'en, og gemme dem ordnet i en vertexbuffer[32]. Det vil herefter være muligt at generere trekant-indices på CPU'en, ved blot at identificere silhuet-kanter, men ikke de eksakte punkter.

For at migrationen til grafikkortet skal give mening, er der to opgaver der skal flyttes samlet. Det handler om beregningen af silhuetpunkterne, og beregningen af silhuetkurverne henover hver flade. Silhuetpunkterne findes ved en binær søgning i den subdividede kurve, hvilket forekommer trivielt når først punkterne er beregnet. Beregningen af silhuetkurven er formodentlig mere udfordrende, idet udgangspunktet vil være data, ligeledes dannet på GPUen.

Samlet set er det ikke en triviel opgave at flytte hele metoden til grafikkortet, og det bør overvejes nøje hvordan det skal gøres i praksis. Opgaven kan med stor sandsynlighed udføres på GPUen, men på nuværende tidspunkt er det uklart i hvilken grad det vil øge evalueringshastigheden.

8.2 Indledende underinddeling

For at kunne foretage tabelbaseret subdivision af en kant, er det nødvendigt at kun det ene endepunkt er irregulært. Der foretages derfor en indledende underinddeling på modellen, men kun den oprindelige udgave af modellen tegnes i områder af modellen væk fra silhuetten, hvilket betyder en væsentlig hastighedsforbedring.

Det er dog værd at bemærke, at hvis modellen i stedet tegnes efter den indledende underinddeling, opnås naturligt et bedre udgangspunkt for tilnærmelse af grænsefladen, da en subdivision-iteration i sagens natur bringer fladen tættere på grænsefladen. For modeller med meget få polygoner vil dette således som regel være en fordel, idet de i højere grad rammes af de observerede fejl. Underinddelingen mindsker risikoen for at silhuetten skærer en kant flere gange, og bringer modellen som helhed tættere på grænsefladen, hvilket igen mindsker fejlen ved brug af

32 Se afsnit 5.5 om pixelbuffer objekts

normalmapping. Dette er et rimeligt skridt, idet der kan være meget stor forskel på en lavpolygon-model og dens grænseflade, og underinddelingen kun vil tilføje et moderat antal polygoner.

8.3 Direkte evaluering af silhuetkurver

I resultatafsnittet blev silhuetkurverne identificeret som årsag til popping på silhuetten. Problemet relaterer sig til den måde kontrolpolygonen for silhuetkurverne dannes, nemlig på samme måde som for PN-triangles. I stedet for at evaluere en kubisk kurve som tilnærmelse til silhuetten, ville det være muligt at anvende direkte evaluering af subdivision-fladerne *[STAM2, ZORIN2]* til at finde eksakte punkter på silhuetfladen. Denne metode er hurtig nok til at det virker fornuftigt at overveje den til dette formål.

I *[ZORIN2]* tales om en "naturlig parameterisering" for en Loop subdivision-flade. Denne består af de barycentriske koordinater for hver trekant, samt en fjerde, diskret koordinat, givet ved et indeks til trekanten. En måde at forskyde silhuetkurverne, ville således være enten at projicere den kubiske beziér-kurve til de flade trekanter, eller at danne den direkte i dette parameterrum.

Det andet problem med kurverne er, at normalen til det nærmeste punkt på fladen, ikke er garanteret at være ortogonal på øjevektoren, ud over i endepunkterne. Den bedste tilgang til dette problem er formodentlig at forsøge at fitte kontrolpunkterne på kurven ud fra de evaluerede normaler. For at denne fremgangsmåde skal være anvendelig, skal fittingen være meget robust; der er stor risiko for blot at tilføre støj til silhuetten. Yderligere er det næppe en operation der hurtigt nok kan udføres for alle dannede silhuetkurver. Problemet vurderes dog at være af mindre betydning, og bør ikke behandles yderligere før effekten af at forskyde kurven til grænsefladen er afklaret. Samlet set virker det som en fornuftig tilgang, at anskue problemet givet i rummet for den naturlige parameterisering. Den i dette projekt viste metode, ville svare til en søgning langs kanterne i hver trekant. Denne anskuelse ville formodentlig lette udvidelser til mere udførlige søgninger efter silhuetpunkter.

9 Konklusion

Der er i dette projekt udviklet en ny metode til forbedring af konturer, til brug ved visualisering af subdivision flader. Metoden tager udgangspunkt i en søgning efter konturpunkter, og danner herudfra en tilnærmelse til det renderede objekts konturer. Det er således muligt at bevare bløde konturer på en ellers lavdetaljeret flade, ved lokalt at foretage en forskydning til modellens subdivision-grænseflade. Dette løser et væsentligt problem omkring visualisering af subdivision-flader, hvor kantede silhuetter ofte afslører fladernes lavdetaljerede natur.

Den præsenterede metode til kontur-korrektion er kombineret med den i forprojektet udviklede metode, "subdivision-surface normalmapping". Herved er præsenteret en samlet løsning for visualisering af subdivision flader i realtid. Det er vist, at denne kombination giver en overbevisende approksimation til subdivision-grænsefladen, såvel i det indre af fladen, som nær konturer på objektet. Hastigheden for denne samlede løsning er vurderet i forhold til direkte subdivision af modellen. Approksimeres grænsefladen med tre eller flere subdivision-iterationer, er visualisering med normalmapping og kontur-korrektion hurtigere end blot at tegne trekanterne genereret ved sædvanlig underinddeling af fladen. Yderligere er det vist, at metoden skalerer bedre til højere iterationer end bruteforce-tegning. En række operationer der udføres per vertex, så som skinning og morphing, er nu blevet standard at udføre på grafikkortet. Med brug af kontur-korrektion, kan disse operationer i stedet udføres på kontrolmodellen, inden visualisering. Dette vil være en betydelig lettelse i forhold til beregning på den underinddelte model.

Den udviklede metode har et meget lavt forbrug af lager-ressourcer, idet der under søgningen alene tages udgangspunkt i kontrolmodellen for subdivision fladen. Til gengæld er den ganske beregningstung, og afvikles hovedsageligt på computerens CPU. I praksis er dette en ulempe, idet alle beregnede trekanter skal sendes til grafikkortet. Metoden har dog potentiale for at blive GPU-accelereret, og bygger på en metode der tidligere er vist implementeret på grafikkortet.

En af grundene til at metoden kan flyttes til GPU er, at den er trivielt parallellisérbar – hver trekant kan uproblematisk evalueres isoleret. Denne egenskab er yderligere afgørende, idet CPU'er indenfor en overskuelig fremtid vil få flere kerner, svarende til maskiner med flere processorer. Algoritmer der kan afvikles parallelt vil derfor kunne opnå væsentligt bedre resultater, da de formår at udnytte alle ressourcerne.

Igennem dette projekt er identificeret en række problematikker ved korrektursøgning, der ikke trivielt kunne angives ved projektets start. Den væsentligste mangel ved selve den præsenterede korrektion af konturer er, at den ikke umiddelbart egner sig til brug i konkave områder af fladen. Der er foreslået en udvidelse af metoden der i de fleste tilfælde vil afhjælpe dette problem ved, at lade underinddelingen af fladen propagere ud fra konturerne. Yderligere opmærksomhed bør dog rettes imod identifikation af de problematiske kanter, for at kunne foretage korrekt visualisering i alle tilfælde.

Et andet væsentligt problem ved metoden, er at de dannede kontur-kurver ikke ligger præcis på grænsefladen til modellen. Dette giver anledning til forkert renderede konturer, og kan resultere i visuel "popping". Problemet opstår på grund af den valgte måde at danne kontur-kurverne, baseret på heuristikken fra PN-triangles. En ideel løsning vil formodentlig basere sig på direkte evaluering

af fladen, men det bør undersøges, om der i stedet kan udvikles bedre heuristikker for konstruktion af kontrolpolygonen for konturkurverne.

Den udviklede metode eliminerer i praksis kantede silhuetter, hvilket betyder at andre skavanker træder i forgrunden, primært problemet omkring afvigelsen imellem flader ved brug af normalmapping. Kombinationen med normalmapping i det hele taget, begrænser anvendelsesområderne for den resulterende metode. Idet vertex-animation nødvendiggør genberegning af normalmappet, er der et væsentligt overhead ved at foretage denne type operationer i realtid. Alternativer til denne løsning bør derfor overvejes, f.eks. ved brug af en kvadratiske funktion. Som en del af implementeringen af normalmapping er det vist, at brugen af en Euklidisk afstandstransformation effektivt fjerner artefakter forbundet med bi-lineær sampling af grænseområder. Det er yderligere vist, hvordan en parallel variant af en EDT kan hardware-accelereres, ved implementering i pixelshadere.

10 Vedlagt Software

På den vedlagte CD findes følgende kataloger:

`/docs` – indeholder de vigtigste artikler anvendt under dette projekt. Sorteret efter område.

`/prototype/bin` – indeholder binær filer og nødvendige dll-filer.

`/prototype/data` – indeholder testscener i obj-format

`/prototype/libs` – de kode-biblioteker der anvendes i oversættelsen af prototypen.

`/prototype/src` – C++ kildekode til den implementerede prototype.

`/prototype/vc2003_proj` – indeholder projektfiler til Microsoft Visual Studio .NET 2003.

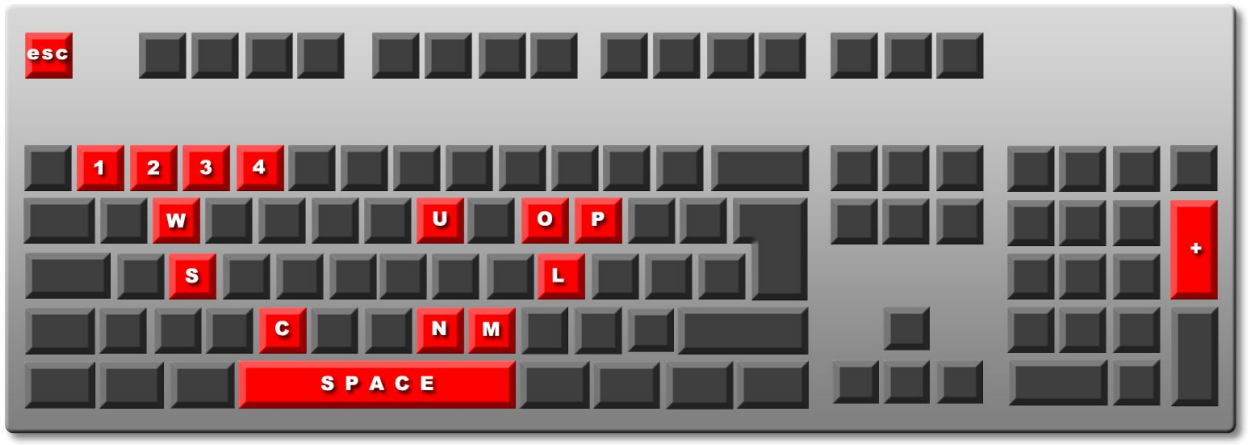
Systemkravene til det implementerede program, der findes under `/prototype/bin`, er følgende:

- Windows2000 / Windows XP
- Et Nvidia GeforceFX-kort eller bedre.

Progammet kan køres direkte fra windows - der er ingen kommandolinie-parametre. Når programmet åbnes vises en fil-dialog hvorfra skal vælges en fil indeholdende det objekt der ønskes visualiseret. I prototypen der er leveret med, er en hård grænse på 500 polygoner sat, under hvilken en model underinddeles.

Der er implementeret en standard trackball til at navigere i programmet.

- Venstre museknap roterer
- Højre museknap bevæger kameraet i den retning man ser (”dolly”)
- Midterste museknap bevæger kameraet i skærmens plan (”strafe”)



c	Centrer omkring objekt
p	Skift mellem perspektivisk og ortografisk projektion.
Escape	Forlad programmet
w	Skift imellem at tegne modellen som wireframe/solid/solid+wireframe.
m	Skift imellem materialer
o	Skift imellem at vise origo for verdenskoordinater.
1	Anvend per-vertex belysning
2	Anvend per-pixel belysning
3	Anvend normalmapping (kun for silhuette-tracking)
4	Visualiser objektets normaler med farver
space	Skift imellem normal model og silhuetsøgning
Når modellen tegnes via silhuetsøgning:	
s	Toggle imellem at tegne silhuetwireframe. Silhuetkurven tegnes i hvid, silhuet-grænser tegnes i rød, og de oprindelige kanter tegnes med gul.
u	Toggle opdatering.
Når modellen tegnes uden silhuet-søgning:	
+	Underinddel model.
L	Skub model til grænseflade.

11 Litteraturfortegnelse

Primær-litteratur i forbindelse med projektet er markeret med **fed**.

[AKENINE], Tomas Möller, Carlo H. Séquin, ”Heuristic Backface Culling of Animated Subdivision Surfaces”, technical report.

[ALEXANDRESCU] Andrei Alexandrescu, ”Modern C++ Design”, ISBN 0201704315

[ALLIEZ], Pierre Alliez, Nathalie Laurent, Henri Sanson, and Francis Schmitt. Efficient View-dependent Refinement of 3D Meshes using Sqrt(3)-Subdivision. The Visual Computer, 19(4):205-221, 2003.

[AZUMA] Daniel I. Azuma, Daniel N. Wood, Brian Curless, Tom Duchamp, David H. Salesin, Werner Stuetzle, ”View-dependant refinement of multiresolution meshes with subdivision connectivity”. Proceedings of the 2nd international conference on Computer graphics, virtual Reality, visualisation and interaction in Africa

[BIEDERMAN] Irving Biederman, ”Recognition-by-Components: A Theory of Human Image Understanding”, Psychological Review vol. 97 nr. 2, 1987

[BIERMANN] Henning Biermann, Adi Levin, Denis Zorin, ”Piecewise smooth Subdivision Surfaces with Normal Control”, SIGGRAPH 2000

[BLINN1] James F. Blinn, Martin E. Newell, ”Texture and reflection in computer generated images”, Communications of the ACM archive, vol.19, Issue 10, (october 1976) pp 542–547.

[BLINN2] James F. Blinn, ”Simulation of wrinkled surfaces”, Proceedings of the 5th annual conference on Computer graphics and interactive techniques, 1978 , pp.286-292.

[BOEHM] W. Boehm, ”Generating the Bezier points of triangular splines”, Surfaces in Computer Aided Geometric Design, Barnhill, R. E, 1983

[BRABEC] Stefan Brabec, Hans-Peter Seidel, ”Shadow Volumes on Programmable Graphics Hardware”, Eurographics 2003, vol.22, 2003.

[BRAWLEY], Z. Brawley, N. Tatarchuk, ”Parallax Occlusion Mapping: Self-Shadowing, Perspective-Correct Bump Mapping Using Reverse Height Map Tracing”, ShaderX^3, 2004.

[BRICKHILL] – David Brickhill, "Practical Implementation Techniques for Multi-Resolution Subdivision Surfaces". GDC Conference Proceeding, 2001.

[CARMACK] Interview med John Carmack, http://www.beyond3d.com/interviews/carmackdoom3/

[CARSTENSEN] – Jens Michael Carstensen, ”Image analysis, vision and computer graphics”, 2. udgave, Informatik og Matematisk Modellering – Technical University of Denmark, 2002.

[CATMULL] Catmull E, Clark J, ”Recursively generated b-spline surfaces on arbitrary topological

meshes". Computer aided design 10, 1978.

[CHAIKIN] G. M. Chaikin, "An Algorithm for high speed curve generation", Computer Graphics and Image Processing, 1974.

[CHEN] Min Si Chen, A.M.Day, "Integrating Displacement Mapping with Subdivision Surfaces", 20th Eurographics UK Conference 2002.

[COOK] R. Cook, "Shade Trees", Computer Graphics, Proceedings Siggraph 1984, 18(3) pp.223-231.

[COHEN01] Jonathan Cohen, Marc Olano, Dinesh Manocha, "Appearance-Preserving Simplification", ACM Siggraph 1998.

[COSGROVE1] Dennis Cosgrove, Takeo Igarashi, "Adaptive Unwrapping for Interactive Texture Painting", Proceedings of the 2001 symposium on Interactive 3D graphics.

[DANA], DANA K. J., NAYAR S. K., VAN GINNEKEN B., KOENDERINK J. J.: "Reflectance and texture of realworld surfaces. ACM TOG 18, 1 (Jan. 1999), 1–34.

[DEBOOR1] C. De Boor, "On Calculating with B-splines", Journal of Approximation Theory, 1972

[DEBOOR2] C. De Boor, "A practical guide to splines", Applied Mathematical Sciences, volume 27.

[DISNEY] - Ramón Montoya-Vozmediano, " "Knot Insertion" on Subdivision Surfaces". Siggraph 2003 sketch

[DOBKIN] David P. Dobkin, "Computational Geometry and Computer Graphics"

[DOGGET] Michael Dogget, Johannes Hirche, "Adaptive view Dependent tesselation of displacement maps", Siggraph/Eurographics Workshop on Graphics Hardware, pp. 59-66, August 2000.

[DONNELLY], William Donelley", "Per-Pixel Displacement Mapping with Distance Functions", GPU Gems 2, ISBN: 0321335597. Kapitlet kan også hentes gratis fra http://developer.nvidia.com

[EISING] – Jens Eising, "Lineær Algebra", 1. udgave, 1997. ISBN 87-88764-39-7

[FOLEY] James D. Foley, Andries van Dam, Steven K. Feiner, John F. Hugues, "Computer graphics: principles and practice", 2nd edition, Addison-Wesley juli 1997, ISBN 0201848406

[GAMMA] - "Design Patterns: elements of reusable object-oriented software" / Erich Gamma et al., 1995, Addison-Wesley. ISBN 0-201-63361-2

[GL1] – link til online-diskussions-forum hvor en udvidelse af relief-mapping der tager højde for krumningen af en flade diskuteres.
http://www.opengl.org/discussion_boards/cgi_directory/ultimatebb.cgi?ubb=get_topic;f=3;t=01284

[GOURAUD] Henri Gouraud, ”Continuous Shading of Curved Surfaces”, IEEE Transactions on Computers, vol. c-20, no. 6, 1971

[GPUGPU.ORG] Hjemmesiden gpgpu.org er samlingssted for alternative anvendelser af grafikkortet. Her kan også findes en række publikationer om emnet.

[GPGPU04] – Aaron Lefohn, University of California, ”GPGPU: GPU Data Formatting and Addressing”, slides til præsentation ved ACM Siggraph 2004.

[GRAVESEN] Jens Gravesen, Institut for Matematik, DTU, ”Differential Geometry and Design of Shape and Motion”. Lecture notes for 01243, November 8, 2002.

[HALL] Tom Hall, ”Silhuette Tracking”, technical report - http://www.bytegeist.com

[HALSTEAD], M Halstead, M Kass, T DeRose , ”Efficient, fair interpolation using Catmull-Clark surfaces”, Proceedings of the 20th annual conference on Computer, 1993

[HART] John. C. Hart, ”Sphere Tracing: A Geometric Method for AntiAliased Ray Tracing of Implicit Surfaces”. The Visual Computer 12, pp. 527-545.

[HAVEMANN] Sven Havemann, ”Interactive rendering of catmull/clark surfaces with crease edges”, The Visual Computer, 2002.

[HAVEMANN2] Kerstin Müller, Sven Havemann, ”Subdivision Surface Tesselation on the Fly using a versatile Mesh Data Structure”, Computer Graphics Forum, 2000.

[HOPPE1] Hugues Hoppe, Tony DeRose, Tom Duchamp, Mark Halstead, ”Piecewise Smooth Surface Reconstruction”, Proceedings of SIGGRAPH, 1994

[HOPPE2] – Hugues Hoppe, ”View-dependent Refinement of Progressive Meshes”, ACM SIGGRAPH 1997, pp. 189-198.

[HOPPE3], Hugues Hoppe, ”Progressive meshes”, ACM SIGGRAPH 1996, 99-108.

[HOPPE4], Hugues Hoppe, T. DeRose, T. Duchamp, J. McDonald, W. Stuetzle, ”Mesh Optimization”, Siggraph 1993, Proceedings, pp 19-26.

[HOPPE5], X. Gu, S. Gortler, H. Hoppe, L. McMillan, B. Brown, A. Stone, ”Silhouette Mapping”, Technical Report TR-1-99, Department of Computer Science, Harvard University, March 1999.

[HOPPE6], Hoppe, H. ”Smooth view-dependent levelof-detail control and its application to terrain rendering”. IEEE Visualization 1998, October 1998, pp 35-42.

[KANEKO] Tomomichi Kaneko, Toshiyuki Takahei, Masahiko Inami, Naoki Kawakami, Yasuyuki Yanagida, Taro Maeda, Susumu Tachi, ”Detailed Shape Representation With Parallax Mapping”, ICAT 2001.

[KETTNER] Lutz Kettner, ”Designing a Data Structure for Polyhedral Surfaces” (openmesh).

[KOBBELT2] Leif Kobbelt, Daubert, Seidel, ”Ray Tracing of Subdivision Surfaces”, Rendering Techniques, 1998.

[KOBBELT4] Leif Kobbelt, ” $\sqrt{(3)}$ -Subdivision”, ACM SIGGRAPH 2000

[KOHLER] Markus Kohler, Heinrich Müller, ”Efficient calculation of subdivision surfaces for visualization”, Visualization and mathematics, pp 165-179, 1997.

[LAVOUE] G. Lavoue, F. Dupont, A. Baskurt, ”Toward near optimal quad/triangle surface fitting”, IEEE Int. Conf. on 3-D Digital Imaging and Modeling (3DIM'2005), Ottawa, Canada, June 2005.

[LEE] Aaron Lee, Henry Moreton, Hugues Hoppe, ”Displaced Subdivision Surfaces”, Proceedings of ACM SIGGRAPH 2000. pp. 85-94, July 2000.

[LEGAKIS] - Justin Legakis, Ron MacCracken, Kenneth I. Joy, ”Data Structures for Unstructured Meshes”, The Visual Computer, 1999.

[LEVY1] Bruno Lévy, Sylvain Petitjean, Nicolas Ray, Jérome Maillot, ”Least Squares Conformal Maps for Automatic Texture Atlas Generation”, Siggraph 2002.

[LOAN], Charles F. Van Loan, ”Introduction to Scientific Computing” 2. udgave, Prentice-Hall, ISBN 0-13-949157-0

[LOOP1] – Charles Loop, ”Smooth subdivision surfaces based on triangles”, Masters thesis, Institut for Matematik, Utah Universitet, 1987.

[MAPS] AWF Lee, W Sweldens, P Schroeder, L Cowsar, DP, MAPS: Multiresolution Adaptive Parameterization of Surfaces, Siggraph 1998

[MCGUIRE] Morgan McGuire, Max McGuire, ”Steep Parallax Mapping”, I3D 2005 Poster.

[MCGUIRE2] Morgan Mcguire, John F. Hughes., ”Hardware-Determined Feature Edges”

[MEYER] Mark Meyer, Mathiu Desbrun, Peter Schröder, Alan H. Barr, ”Discrete Differential-Geometry Operators for Triangulated 2-Manifolds”, Visualization and Mathematics III, 2003

[MOULE] Kevin Moule, ”Efficient Bounded Adaptive Tessellation of Displacement Maps”, GI2002.

[NASRI] Ahmad Nasri, Gerald Farin, ”A subdivision algorithm for generating rational curves”, Journal of Graphics Tools, vol 6 issue 1, pp. 35-47, 2001.

[NV40] Emmett Kilgariff, Randima Fernando, ”The GeForce 6 Series GPU Architecture” GPU-Gems 2 Chapter 30 , ISBN: 0321335597.

[OLIVEIRA] Manuel M. Oliverira, Gary Bishop, David McAllister, ”Relief texture mapping”, Proceedings 27th annual conference on Computer Graphics and interactive techniques 2000, pp. 359-368.

[PASCUCCI] V. Pascucci, ”Slow growing subdivision in Any Dimension, towards removing the curse of Dimensionality”, Eurographics 2002, vol 21

[PBOURKE1] .obj-format specifikation http://astronomy.swin.edu.au/~pbourke/geomformats/obj/

[PIXAR2] Tony DeRose, Michael Kass, Tien Truong, ”Subdivision Surfaces in Character Animation”, Proceedings of the 25th annual conference on Computer graphics and interactive techniques.

[PULLI] Kari Pulli, Mark Segal, ”Fast Rendering of Subdivision Surfaces”, Rendering Techniques 1996, Wien.

[RIESENFELD] R. Riesenfeld, ”On Chaikin's Algorithm”, IEEE Computer Graphics and Applications 4, 1975

[RÖTTGER] Röttger, S. and Heidrich, W. and Slusallek, P. and Seidel, H. P. ”Real-Time Generation of Continuous Levels of Detail for Height Fields”, V. Skala, editor, Proceedings of WSCG’98, pages 315-322, 1998.

[RTR2] – Tomas Möller, Eric Haines, ”Real-time rendering 2nd edition”. ISBN 1-56881-182-9

[SABIN1] M. Sabin, D. Doo, ”A Subdivision Algorithm for Smoothing down Irregularly Shaped Polyhedrons”, proceedings on Interactive Techniques in Computer Aided Design, Gologna, 1978.

[SABIN2] TW Sederberg, J Zheng, D Sewell, M Sabin, ”Non-uniform Recursive Subdivision Surfaces”, SIGGRAPH, 1998.

[SANDER] - Silhouette clipping. P. Sander, X. Gu, S. Gortler, H. Hoppe, J. Snyder. ACM SIGGRAPH 2000, pp. 327-334.

[SCHRÖDER1] Peter Oswald, Peter Schröder, ”Composite Primal/Dual $\sqrt{(3)}$ subdivision schemes”, Computer Aided Geometric Design 2003.

[SCHRÖDER3], Jeffrey Bolz, Peter Schröder, ”Rapid Evaluation of Catmull-Clark Subdivision Surfaces”, Proceeding of the seventh international conference on 3D Web technology, 2002

[SCHRÖDER4], Andrei Khodakovsky, Nathan Litke and Peter Schröder, ”Globally Smooth Parameterizations with Low Distortion”, ACM Transactions on Graphics, 2003

[SCHRÖDER5], Peter Shröder, Jeff Bolz, ”Evaluation of Subdivision Surfaces on Programmable Graphics Hardware”, Caltech, 2003

[SCHWEITZER] Jean E. Schweitzer, ”Analysis and Application of Subdivision Surfaces”, PhD dissertation, University of Washington, August 1996.

[SEDERBERG1] Thomas W. Sederberg, Jianmin Zheng, Almaz Bakenov, Ahmad Nasri, ”T-splines and T-NURCCs”, ACM Transactions on Graphics, 2003.

[SHEFFER1] AllaSheffer, John C. Hart, ”Seamster: inconspicuous low-distortion texture seam layout”, Proceedings of the 25th annual conference on Computer graphics and interactive techniques, Siggraph 1998

[SIGGRAPH] Siggraph 2000 course notes, ”Subdivision for Modelling and Animation”, Editorer: Denis Zorin og Peter Schröder.

[SLOAN] Peter-Pike Sloan, Jan Kautz, John Snyder , ”Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments”, ACM Transactions on Graphics, Proceedings of the 29th annual conference on Computer graphics and interactive techniques, volume 21 Issue 3

[STAM1] Jos Stam, ”Exact Evaluation of Catmull-Clark Subdivision Surfaces at Arbitrary Parameter Values”, Siggraph 1998

[STAM2] Jos Stam, ”Evaluation of Loop Subdivision Surfaces”, SIGGRAPH'99 Course Notes

[STAM3] Jos Stam, Charles Loop, ”Quad/Triangle Subdivision”, Computer Graphics Forum, 2003

[STOLLNITZ] Eric J. StollNitz, Tony D. DeRose, David H. Salesin, ”Wavelets for Computer Graphics”, 1996, Morgan Kauffmann Publishers, ISBN 1-55860-375-1

[TATARCHUK] N. Tatarchuk, ”Parallax Occlusion Mapping”, slides fra præsentation GDC2005, http://ati.com/developer/techpapers.html#gdc05

[THÖLE], J.L., H. Thöle, “Creating and Editing Curves on Subdivision Surfaces”, Proceedings of SIBGRAPI 2003 (São Carlos), pp. 75–80

[VELHO1] Luiz Velho, Jonas Gomes, ”variable Resolution 4-K Meshes: Concepts and Applications”, Computer Graphics Forum, 2000.

[VELHO2] Luiz Velho, Denis Zorin, ”4-8 Subdivision”, Computer Aided Geometric Design, 2001.

[VLACHOS] Alex Vlachos, Jörg Peters, Chas Boyd, Jason L. Mitchell, ”Curved PN Triangles”, Proceedings of the 2001 symposium on Interactive 3D graphics, pp. 159-166

[WEILER] Kevin Weiler, ”Edge Based Data Structures for Solid Modeling in Curved-Surface Environments”

[WANG1] Lifeng Wang, Xi Wang, Xin Tong, Stephen Lin, Shimin Hu, Baining Guo, Heung-Yeung Shum, ”View-Dependent Displacement Mapping”, Proceedings of ACM SIGGRAPH 2003 , pp. 334–339.

[WANG2] Xi Wang, Xin Tong, Stephen Lin, Shimin Hu, Baining Guo, Heung-Yeung Shum, ”Generalized Displacement Maps”, Eurographics Symposium on Rendering, 2004.

[WARREN1] Joe Warren, Henrik Weimer, ”Subdivision Methods for Geometric Design”, Academic Press 2002, ISBN: 1-55860-446-4

[WARREN2] Joe Warren, Scott Schaefer, ”A factored approach to Subdivision Surfaces”, Compututer Graphics and Applications 2003, Vol. 24, No. 3, pp 74-81.

[WARREN3] Joe Warren, Scott Schaefer, ”On C^2 Triangle/Quad Subdivision”, ACM Transactions on Graphics, 2005.

[WELSH] Terry Welsh, ”Parallax Mapping with Offset Limiting: A Per-Pixel Approximation of Uneven Surfaces”, rev. 0.3, Januar 2004. Technical Report, http://www.infiscape.com/doc/parallax_mapping.pdf

[YAMADA] - Hiromitsu Yamada, ”Complete Euclidean Distance Transformation by Parallel Operation”, International conference on pattern recognition, ICPR 7.conf. Montreal 1984.

[ZORIN1] Denis Zorin, ”Stationary Subdivision and Multiresolution Surface Representations”, PhD theis, Caltech, Pasadena, 1997.

[ZORIN2] Denis Zorin, Daniel Kristjansson, ”Evaluation of Piecewise Smooth Subdivision Surfaces”, The Visual Computer, 2002.

[ZORIN3] Denis Zorin, Peter Schröder, Wim Sweldens, ”Interpolating Subdivision for meshes with arbitrary topology”, Proceedings of the 23rd annual conference on Computer graphics and interactive techniques, 1996.