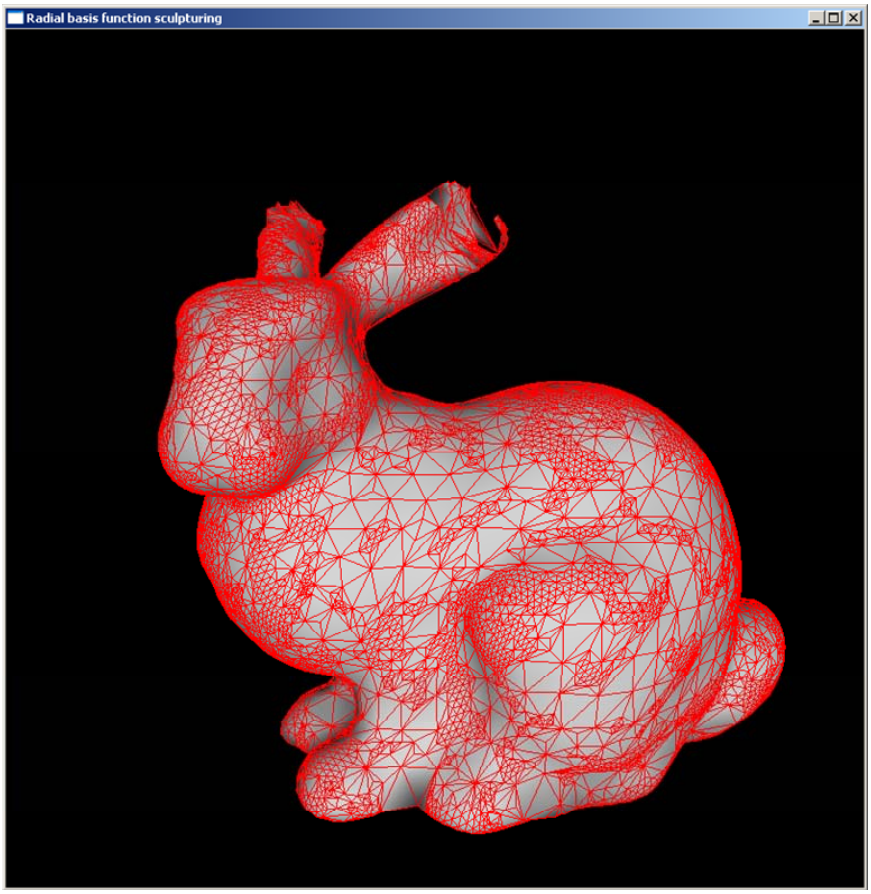
Polygonalisering af ISO-Overflader



Indholdsfortegnelse

Forord

Dette projekt er mit afsluttende eksamensprojekt, skrevet ved IMM på DTU i efteråret 2004. Projektet er evalueret til 15 ECTS point.

Jeg vil gerne benytte dette forord til også at bibringe Institut for Informatik og Matematisk Modellering (IMM) og i særdeleshed mine vejledere professor Niels Jørgen Christensen og adjunkt J. Andreas Bærentzen en stor tak for at give mig mulighed for at skrive mit eksamensprojekt hos dem. Jeg har gennem det sidste år arbejdet med specialkursus, lidt uortodokst være i praktik og endelig skrevet dette eksamensprojekt hos jer, og det har givet mig stor indsigt i computer grafik og i særdeleshed modellering i 3d. Jeg har gennem dette forløb fået bekræftet, at det er inden for computergrafikken, at jeg er i mit es, og jeg formoder at en stor del af mit liv vil forløbe inden for dette område.

Med hensyn til syntaksen i nærværende rapport, vil jeg kort nævne følgende generelle ting. Når der i teksten henvises til en ”Figur”, forefindes figuren som regel på samme side eller evt. forrige eller næste side. Hvis der derimod refereres til et ”Billede” refereres der til screenshots taget fra min polygoniser. Disse er placeret som bilag i afsnit 7.3. Af og til er der ord hvortil der er vedhæftet note, disse noter er placeret i en ordforklaring der er at finde helt bagerst i rapporten i afsnit 7.5

På vedlagte cd, findes al kildekoden, i et større framework, der er udviklet i Visual Studie .net, samt en kopi af denne rapport. Kildekoden for denne implementering findes under mapper med navnet ”Jakobsen”, og kildekoden for Bloomenthals findes under et tilsvarende mappenavn.

1 Indledning

I dette projekt fremlægges en løsning til at visualisere et 3d objekt defineret ud fra en implicit funktion $f(\bar{x}) = 0$ hvor $\bar{x}$ er et punkt i det 3 dimensionale rum. Der findes flere metoder til at gøre dette. En populær og traditionel måde er ved ray-casting, hvor rødderne til funktionen findes ved skæringer mellem funktionen og linier (rays) kastet fra pixels på skærmen til kamerapositionen. Denne metode har den ulempe, at hele forløbet skal køres for hver position kameraet antager i scenen, og dette er yderst ressourcekrævende. En anden metode er at danne et mesh[1], som approksimerer overfladen. Fordelen i forhold til ray-casting er, at forløbet her kun køres en gang, og da et mesh er en samling af trekanter i rummet, er hele figuren derfor repræsenteret, uanset hvor kameraet placeres i scenen.

1.1 Målsætning

Dette projekt har til formål at udvikle et stykke programmel, som kan visualisere et tredimensionelt volumen givet ved en implicit funktion; en Polygoniser[2]. En polygoniser danner en mængde sammenhængende trekanter, et mesh, der senere visualiseres af f.eks. OpenGL[3].

1.1.1 Problemstilling

De simpleste former af polygonisere *[Lorensen, Bloomenthal]* er ikke i stand til adaptivt at tilpasse meshet i forhold til stor variation i lokale områder at et volumen, de er derfor nødsaget til uniformt at neddele et net, indtil en tilpas cellestørrelse er opnået, se evt. figurer på side 10.

Derudover danner flere af disse polygonisere også grimme meshes, der indeholder mange såkaldte slivers[4], se evt. Billede 5, Billede 7 og Billede 9.

1.1.2 Hypotese

I stedet for at lave en større og meget mere kompliceret implementering, der kan løse ovenstående problemer, formodes det at problemerne kan løses ved at tilføje et par simple metoder til en simpel udgave af en polygoniser.

Det antages, at der ved brug af adaptiv √3 neddeling af et dualt net, kan opnås en stor detaljegrad (mange små trekanter) i områder med stor variation og lille detaljegrad (få store trekanter) i områder, der kun varierer lidt.

Dertil tilføjes en simpel algoritme der skubber de punkter, der danner trekanterne, væk fra hinanden, således at man opnår den størst mulige spredning på disse punkter, og derved danner pæne trekanter.

1.2 Oversigt over valg af metoder

Der er allerede påvist mange forskellige metoder til hvorledes man danner dette mesh, og i dette afsnit beskrives hvilke valg der er taget i forhold til dette projekt.

Grundlæggende for at generere et mesh, skal der bruges nogle vertices[5] til at danne de polygoner, der danner overfladen. Disse kan genereres på flere måder. Først skal rummet neddeles i et antal mindre bokse/celler, kaldet polygonaliseringsceller eller voxels, for at kunne bestemme overfladens omtrentlige placering. Nogle af metoderne af beskrevet i afsnit 2.2 og den metode, der er valgt, er at gøre brug af uniforme duale net.

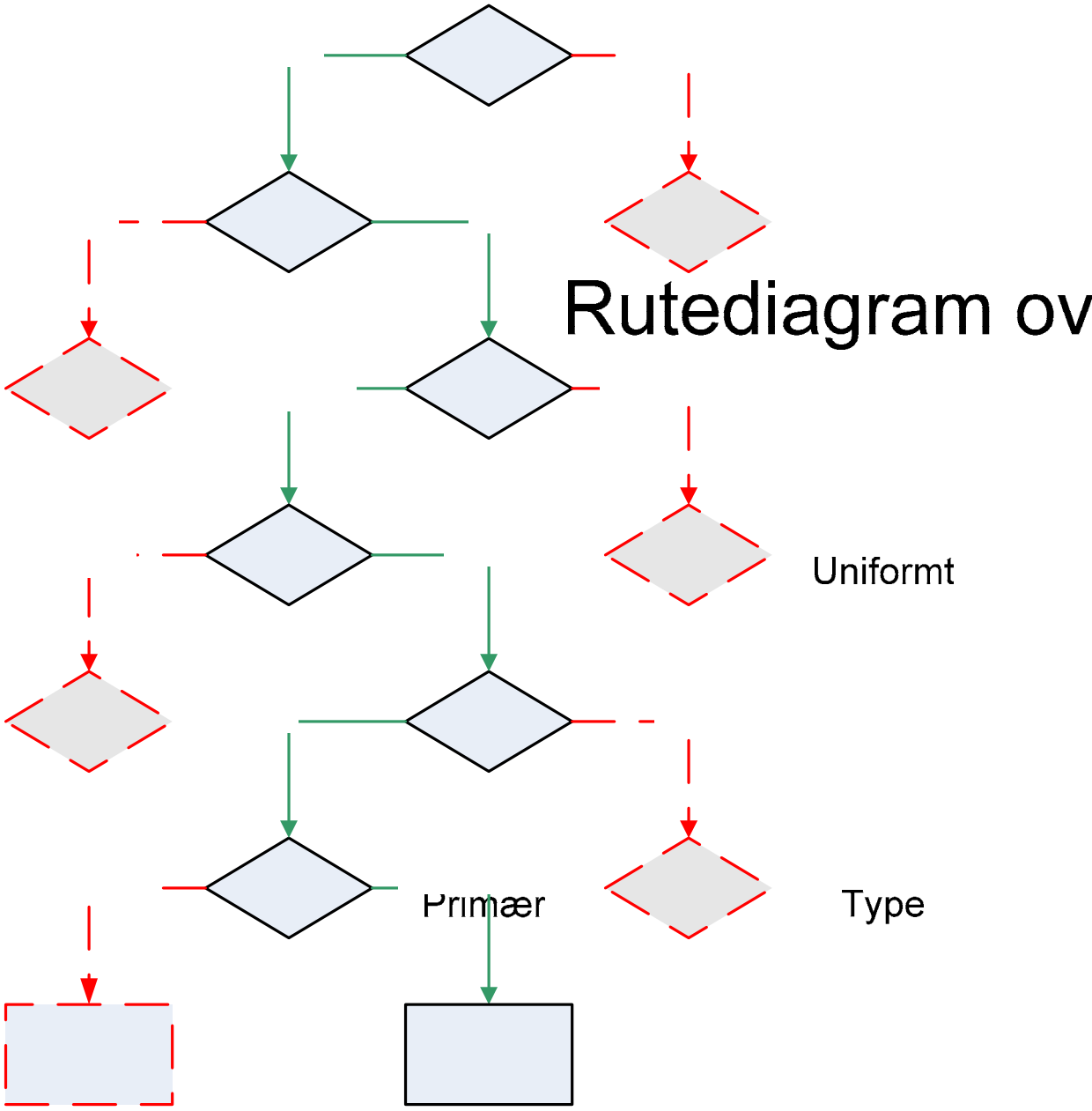
Valget af et uniformt net, kræver endnu et valg, om man vil undersøge alle celler, eller om man vil starte med en celle der skærer overfladen og så herefter følge de celler, der hænger sammen med denne første celle og tilsammen dækker hele overfladen. Det er her er valgt at bruge samtlige celler og metoderne er beskrevet i afsnit 2.3.

Det skal herefter vælges, hvorvidt man ønsker at interpolere sig frem til den eksakte position på et fastlagt net, hvor overfladen skærer nettet, og placere vertexet der, eller man kan beregne det sted i cellen, hvor overfladen mest sandsynligt befinder sig, og placere vertexet der, eller man kan placere vertexet et tilfældigt sted i cellen, f.eks. midten, og så herfra projicere den ned eller op på overfladen, ved at flytte det i modsat retning af gradienten for det punkt. Det er i dette projekt valgt at benytte projicering af vertices placeret i midten af cellerne. Dette er beskrevet i dybere detaljer i afsnit 2.4.

Siden, der i projektet arbejdes med et uniformt net, har man ingen mulighed for at have forskellige størrelser af polygonaliseringsceller, og hvis man ønsker at have større nøjagtighed i områder med stor variation, er man nødt til at have et finere net eller neddele polygonerne/trekanterne i mindre trekanter, der derefter tilpasses bedre til overfladen. Et par metoder er beskrevet i afsnit 2.5, og i dette projekt er det valgt at benytte adaptiv √3 neddeling.

Opsummering af de valg som polygoniseren benytter (valgene er yderligere illustreret på Figur 1)

- Et uniformt dualt net, til at danne de originale vertices
- Lineær søgning, for at finde samtlige elementer af volumenet
- Projicering til overfladen, til at ”fitte” vertices til overfladen
- Udglatning/smoothing, for at undgå slivers
- Adaptiv √3 neddeling, til at findele meshet, der hvor der er stor lokal variation



Figur 1: Rutediagram som viser valg taget i forbindelse med projektet

2 Teori

2.1 Implicitte overflader

En implicit overflade, er en overflade som er defineret ud fra en implicit funktion $f(\bar{x}) = c$, *[Bloomenthal2, Ning]*. For et hvert punkt i vores rum, giver denne funktion os en skalar, og danner derfor et skalar felt.
Et volumen kan heraf beskrives som den iso-overflade, hvor funktionen er lig med nul: f(x,y,z) = 0. Hvis funktionen er større eller mindre, er det et punkt som ligger henholdsvis indenfor eller udenfor volumenet, altså for $f(\bar{x})$ hvor $\bar{x}$ er et punkt p(x,y,z) haves:

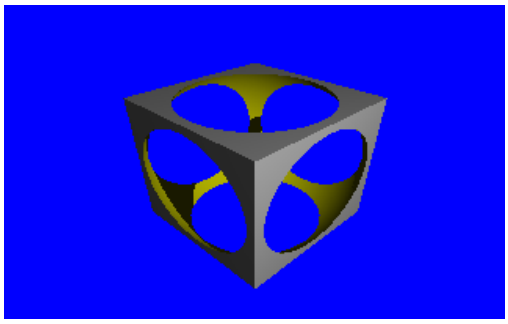
- $f(\bar{x}) = 0$ er et punkt på overfladen af volumenet
- $f(\bar{x}) < 0$ er et punkt udenfor volumenet
- $f(\bar{x}) > 0$ er et punkt indenfor volumenet

Der findes flere forskellige måde at danne overflader af denne karakter på, den simpleste er selvfølgelig simple overflader/primitiver, som f.eks. en kugle $f(\bar{x}) = x^2 + y^2 + z^2 - r^2$, men det giver selvsagt nogle problemer, når man vil danne overflader af mere kompleks karakter, som f.eks. et skrivebord, for hvordan ser funktionen for et skrivebord ud?

2.1.1 Blobs

En blob er i praksis en kugle eller ellipsoide, der bruges som byggesten til at forme en ønsket overflade, således at man indsætter flere og flere blobs, som overlapper hinanden indtil man opnår den ønskede effekt.

2.1.2 Constructive Solid Geometry (CSG)



Figur 2: En wiffled cube konstrueret med CSG, figuren er lånt fra ukendt kilde

Constructive Solid Geometry eller bare CSG, gør i langt bredere form brug af det samme princip. For det første kan der anvendes enhver form for primitiv, der defineres af en implicit funktion, som byggesten og derefter opbygger man sit volumen ved hjælp af booleske funktioner, så man f.eks. kan visualisere forskellen mellem en kugle og en kubus, en såkaldt ”wiffled cube”, Figur 2. På denne måde kan man både trække primitiver fra hinanden og lægge dem sammen, hvilket svarer til differensmængden eller fællesmængden for primitivernes funktioner, og derved skabes et CSG-træ, der til sidst beskriver det endelige volumen.

2.1.3 Radial Basis Functions (RBF)

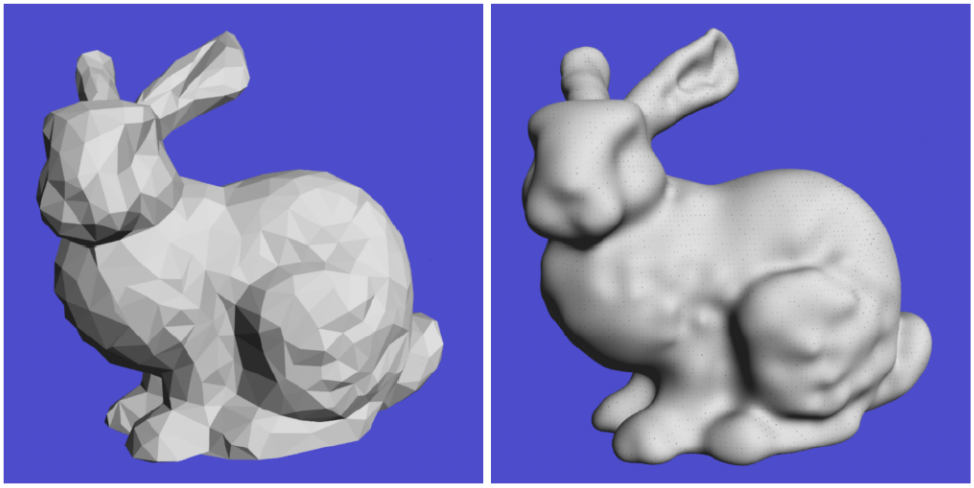
Radial Basis Functions *[Turk, Carr, Reuter]* fungerer på en helt anden måde. Her har man et antal punkter som man ved ligger på overfladen, f.eks. et undersæt af punkter fra en punktsky fra en laserscanning. Ved brug af RBFs, som har den egenskab at de er radiale om deres centre, kan man danne en approksimeret overflade, som er beregnet som den overflade der har den mindste energi-fluktuering. Dette gøres i praksis ved at beregne en vægt, der angiver, hvor stor en indflydelse RBF'en skal have på den endelige overflade, for hvert af disse sample-punkter. Den implicitte funktion/overflade er således nu approksimeret som en vægtet sum af alle disse rbf'er og det er dette, der beskriver vores volumen:

$$f(\vec{x}) = p(\vec{x}) + \sum_{j=0}^{k} w_j \phi(|\vec{x} - \vec{c}_j|)$$

Hvor $p(\vec{x})$ er et polynomium af lav grad, ω er vægten for det punkt, φ er den specifikke radial basis function og |.| angiver den Euklidiske længde mellem det relevante punkt og de andre punkter i mængden.

Det anses for værende irrelevant at gå i dybere detaljer med matematiske formuleringer og definitioner for denne metode selv om det er denne form for overfladedannelse der er brugt i projektet, da det er en naturlig udvidelse af tidligere arbejde indenfor samme felt. Dog skal det bemærkes at ulempen ved denne metode i forhold til f.eks. CSG er at den ikke er i stand til at reproducere skarpe kanter, hvilket dog på den anden side giver den fordel, at der derfor ikke er behov for at tage højde for dette, når der skal polygonaliseres, hvilket har stor relevans for projektet.

På nedenstående Figur 3 ses hvorledes en RBF overflade dannes ud fra sample punkter svarende til de originale vertices i en eksisterende polygonmodel. Man ser tydeligt hvorledes metoden udglatter overfladen, men samtidig bevarer diverse karakteristika.



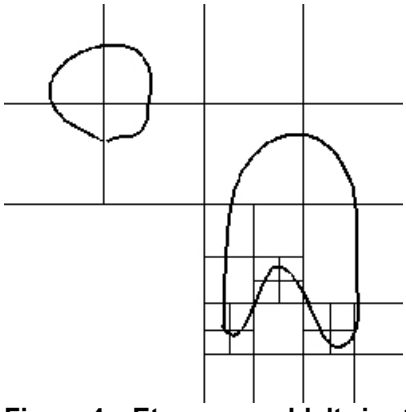
Figur 3: Stanford Bunny konverteret fra eksisterende polygon model til rbf-model, figuren er lånt fra [Turk]

2.2 Net typer

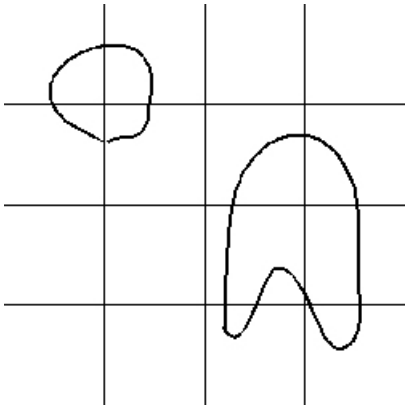
Problemet ved at skulle visualisere en implicit overflade er at det netop kun er funktionen eller et CSG-træ af funktioner, der er givet. Man er derfor nødsaget til at generere nogle punkter/vertices som bruges til at danne de polygoner/trekanter, der visualiserer overfladen. Dette gøres ved at inddele det foruddefinerede rum i et net. Hvert underrum i dette net kaldes for en polygoniseringscelle eller en voxel. Der er forskellige måder man kan inddele sit rum på.

2.2.1 Strukturerede net

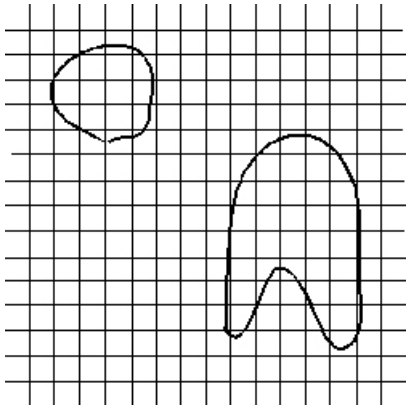
I et struktureret net benyttes ofte en træstruktur til at neddele rummet i, f.eks. et octree, som f.eks. brugt i *[Schaefer2]*. Polygoniseren starter således med en celle, som er volumenets bounding box[6], hvorefter denne celle neddeles i 8 nye celler, og således forsættes der for hver ny child[7] celle med at blive neddelt, indtil enten en celle vurderes som tom, altså ikke indeholdende elementer af volumenet, eller til et vist neddelingsniveau er opnået. Bladnoderne i træet vil nu enten være tomme, eller have den tætteste approksimation af overfladen i sig. Problemet med at bruge et struktureret net er delvis, at det er sværere at implementere, og i større grad, at man får bladceller af forskellig størrelse, hvilket giver problemer når man skal til at generere polygonerne, da man da skal have små polygoner til at hænge sammen med større. Fordelen er, at man oftest bruger færre celler til at approksimere volumenet med, og at man har en høj grad af approksimation, da cellerne automatisk bliver mindre i områder med stor overfladevariation. På Figur 4 ses et rum neddelt i et struktureret net. For overskuelighedens skyld er her brugt 4-neddeling på et 2d rum. Praktisk udvides dette bare med en dimension til 3d og dermed 8-neddeling.



Figur 4: Et rum neddelt i et struktureret net



Figur 5: Et rum uniformt neddelt, måske tilpas, men ikke lige så høj approksimation som i det strukturerede net



Figur 6: Et rum uniform neddelt for at opnå ligeså høj approksimation som ved struktureret neddeling

2.2.2 Uniforme net

Uniforme net udskiller sig fra de adaptive ved ikke at inddele rummet i tilpassede celler, men i helt ens celler *[Lorensen]*. Deraf navnet uniform. Dette har de fordele, at det er langt lettere at implementere samtidig med, at man undgår problemer med at skulle få polygoner fra celler af forskellig størrelse til at passe sammen. Det uniforme net til gengæld den ulempe, at man fastsætter en bestemt størrelse på nettet fra start af, og det gør at der skal afbalanceres mellem at

- have få og store celler, og derfor lille grad af approksimation og evt. mangler i volumenet. (se Figur 5)

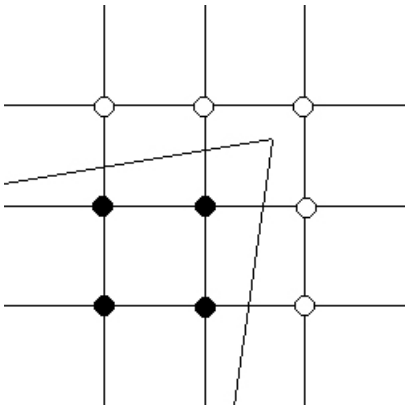
- have mange og små celler, og derfor en længere behandlingstid, men til gengæld en større grad af approksimation og en stor sandsynlighed for at fange alle elementer af volumenet. (se Figur 6)

Indtil videre haves to forskellige typer, primære og duale net.

2.2.3 Polygonaliseringscellen

Hvorledes den enkelte mindste celle behandles afhænger af, om der bruges primære eller duale net. For primære net indsættes nye vertices på kanterne af cellen, mens der for duale net indsættes vertices i et net som er parallelforskudt i forhold til det oprindelige net.

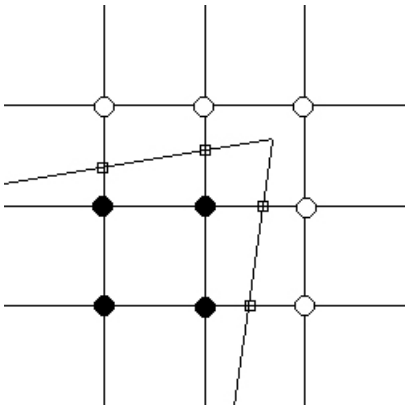
På Figur 7 ses et eksempel på et uniformt net med fire celler. De sorte prikker indikerer cellehjørner, der ligger inde i volumenet og de hvide indikerer cellehjørner, der ligger udenfor. Det betyder at enhver kant som ligger mellem en sort og en hvid prik nødvendigvis må skære overfladen. Således kan man nu på forskellig vis finde frem til, hvor overfladen er, og derpå indsætte vertices og danne de polygoner, der skal visualisere den. Selve indsættelsen varierer efter hvilken en nettype der bruges.



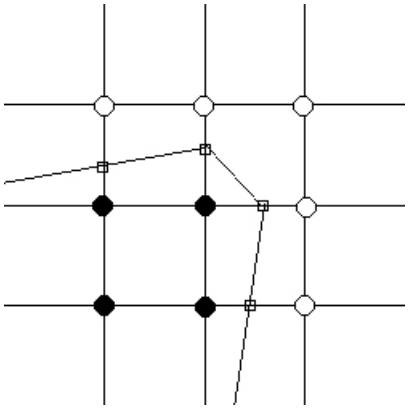
Figur 7: Fire polygonaliserings-celler, sorte prikker ligger inde i volumenet, hvide prikker udenfor

2.2.3.1 Primære net

Når der bruges et primær net, tager man en celle af gangen og kigger på de 8 hjørner den har *[Lorensen]*. For hver kant, der har to hjørner med modsat fortegn, indsættes en ny vertex, se Figur 8. Disse vertices findes typisk ved bisektion, hvor man tager et punkt på halvdelen af kanten og finder ud af om man har ændret fortegn i forhold til den vertex man har rykket sig fra, og således forsættes indtil man har nået et vist antal halveringer eller til man rammer overfladen eksakt. Disse nyindsatte vertices forbindes så i de polygoner, der derved visualiserer overfladen. Der er 2^8 forskellige kombinationer af celler, men de kan populært sagt, koges ned til 16, idet resten blot er rotationer eller spejlvendinger deraf. Oftest bruger man derfor et tabelopslag for at finde de kanter, hvor vertices skal indsættes. Der er dog nogle problemer ved denne metode. For det første er den i sin simple form ikke i stand til at vedligeholde skarpe kanter, se Figur 9, og for det andet kan der, ved nogle typer af celler, være tvetydighed om, hvordan overfladen snor sig i den pågældende celle. Det sidste af disse problemer gives der et løsningsforslag til under afsnit 2.6.3.



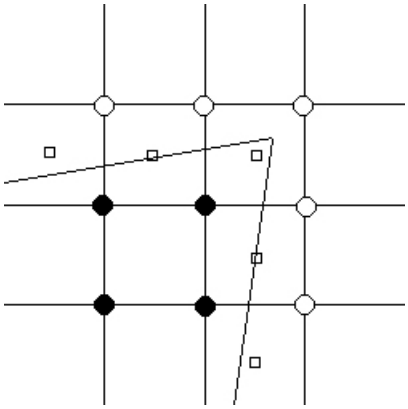
Figur 8: Vertices indsat på kanterne i cellerne der hvor overfladen skæres



Figur 9: Vertices bliver forbundet i de polygoner, der danner overfladen, I 2d vises det her som en linie. Det ses tydeligt at det skarpe hjørne er blevet skåret af

2.2.3.2 Duale net

I et dual net foregår det lidt anderledes, når de nye vertices skal indsættes *[Schaefer]*. I stedet for at kigge på alle hjørnerne i en celle af gangen, kigger man på de enkelte hjørner. For et hjørne undersøges det nu, om nogle af de seks af hjørnets kanter, skærer overfladen. For hver at de kanter, der måtte eksistere, indsættes de nye vertices nu, ikke på kanten, men i hvert center af de fire celler som deler kanten. Herefter forbindes disse fire vertices i to trekanter/polygoner, se Figur 10. Det er disse centre man refererer til som det duale net. De behøver ikke at være placeret nøjagtig således at de passerer gennem centrene på celler i primær-nettet, men for nemheds skyld er det den metode, der anvendes i dette projekt. Man kan forestille sig, at der bliver dannet en væg mellem de to modsatte vertices, der danner kanten. Denne væg repræsenterer overfladen, om end den er flad hvor overfladen kan have hvilken som helst facon. I afsnit 2.4 beskrives hvorledes vertices flyttes, så de placeres på overfladen og dermed danner en mere præcis approksimation.



Figur 10: Vertices bliver indsat i et dual net, som går gennem centrene af cellerne i primær-nettet

2.3 Besøgsceller

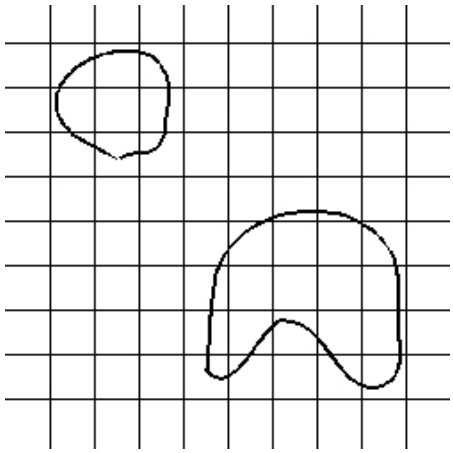
Når nu det endelige net er fastsat, og man derpå skal traversere igennem det, har man valget, om man vil gå igennem samtlige celler, eller om man vil forsøge kun at gå igennem de celler, der skærer overfladen.

2.3.1 Lineær søgning (brute force)

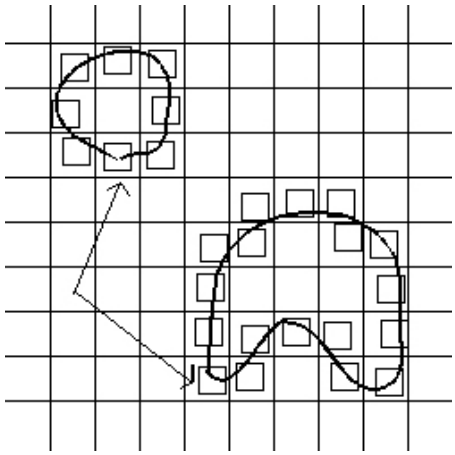
Ved lineær søgning undersøges samtlige celler i nettet for hvorvidt de skærer en overflade eller ej. Dette har den store fordel, at hvis et volumen består af flere adskilte elementer, vil man, givet at cellen er lille nok, finde alle disse elementer. Modsat må man påregne sig en beregningstid på $O(n^3)$, hvor n er antallet af celler. På Figur 11 ses et net og et volumen, der består af to elementer. Begge disse elementer vil blive fundet ved én gennemgang af nettet.

2.3.2 Overflade tracing (continuation)

Når der bruges overflade tracing, er man nødsaget til at have en startcelle, som i forvejen skærer overfladen, hvilket bruges i *[Bloomenthal]*. Denne celles position opgives som regel enten af brugeren, eller findes ved en hurtig tilfældig gennemsøgning af rummet. På Figur 12 er disse startceller angivet med pile. Derefter vil polygoniseren følge overfladen rundt ved kun at besøge naboceller, som deler en kant, der er gennemskåret af overfladen. Fordelen ved denne metode er at den ikke kræver nær så mange beregninger, og tiden anslås derfor til $O(n^2)$. Ulempen er, at man skal kalde polygoniseren med et nyt startpunkt for hvert element af det fulde volumen. Hvis der kun kaldes en gang, vil ét og kun ét element blive fundet og visualiseret.



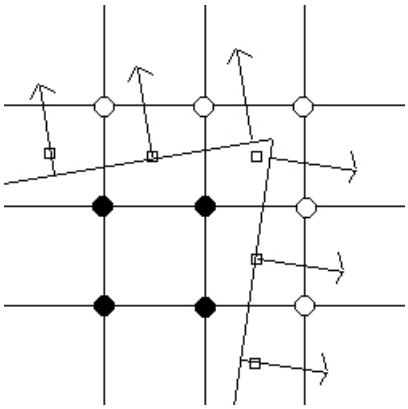
Figur 11: Når der benyttes lineær søgning, findes alle elementer af volumenet



Figur 12: Ved continuation, besøges kun de celler der støder op til hinanden og som skærer overfladen. Pilene markerer to forskellige startceller

2.4 Projicering og udglatning

Når nu hele ens net er blevet traverseret, og man har fået placeret samtlige vertices på det dual net, dvs. i centrene af cellerne af det primære net, har man en approksimeret overflade, der til forveksling kunne minde om en Lego® - figur, da samtlige indsatte polygoner ligger i koordinataksernes planer, se Billede 1 på side 42. Der bruges derfor en simpel projektion til at "fitte" vertices bedre til overfladen.



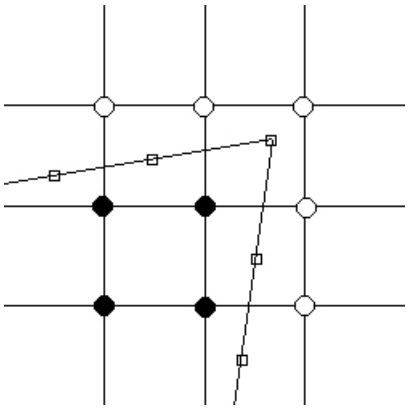
Figur 13: Normaler/gradienter for de nye vertices

2.4.1 Projicering

Når et vertex ligger på overfladen, har dette punkt en normal, som findes ved at beregne gradienten til punktet. Dette gøres ved metoden "differensers center", der er bestemt ved:

$$\nabla f(\bar{x}) = \left(\frac{f(x+\Delta)-f(x-\Delta)}{2\Delta}, \frac{f(y+\Delta)-f(y-\Delta)}{2\Delta}, \frac{f(z+\Delta)-f(z-\Delta)}{2\Delta}\right)$$

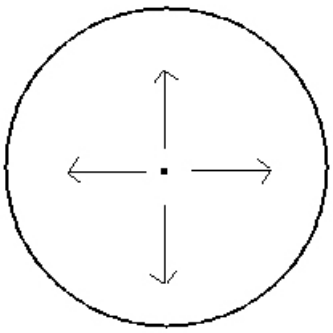
hvor x, y og z repræsenterer de specifikke koordinater i $\bar{x}$, og Δ er det stykke som bevæges væk fra punktet i koordinataksernes retning. Selv om et vertex ikke ligger på overfladen, befinder det sig stadig i et skalar felt, og gradienten vil stadig have en hvis lighed med normalens retning. Længden af gradienten kan give en ide om hvor langt fra overfladen punktet befinder sig. Når vertices derfor fittes til overfladen, beregnes gradienten, og vertexet flyttes derefter langs gradienten med et skridt af gangen med samme længde som gradienten. Dette gøres over flere omgange, indtil et vist antal repetitioner er opnået eller til punktet evalueres til at ligge tæt nok på overfladen. Som oftest er det nemt at finde placeringen af vertices, men i nogle tilfælde er denne metode tvetydig, som f.eks. ved det vertex som ligger nær hjørnet af overfladen på Figur 13. Når vertices er blevet placeret vil det se ud som på Figur 14, og en langt bedre approksimation er opnået, se Billede 2 på side 43.



Figur 14: Vertices er blevet fittet til overfladen, og i dette ideale tilfælde også til hjørnet

2.4.1.1 Nul-gradienter

Et af de særtilfælde, hvor placeringer ud fra projektion langs gradienten ikke fungerer, er hvis gradienten er meget lille eller meget stor. Et eksempel er et punkt placeret i centrum af en cylinder: Her vil gradienten være nul, og vertexet vil derfor ikke blive flyttet. Figur 15 illustrerer dette koncept.



Figur 15: Gradienten vil blive nul for et vertex placeret i centrum af en åben cylinder, her vist i et tværsnit

2.4.2 Udglatning

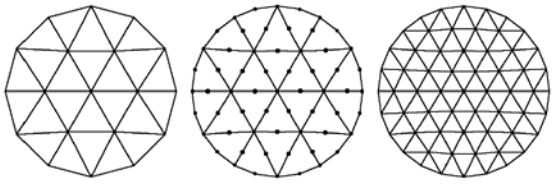
En sidste ting, der kan påvirke placeringen af vertices, er ved udglatning/smoothing. Når vertices bliver projekteret ned på en overflade, vil boksen de har haft formet oftest skabe nogle grimme, aflange og i nogle tilfælde kollapsede trekanter; slivers. For at komme disse til livs kan man udglatte overfladen, ved henholdsvis at trække i og skubbe til vertices, således at de får en balanceret placering i forhold til hinanden, en Laplacian smoothing. Ved denne smoothing behandles ét vertex af gangen, og flyttes til midtpunktet for alle de vertices det danner kant med. Dernæst placeres det i en midlertidig liste. Når alle vertices er blevet placeret således i forhold til dette slags massemidtpunkt for dets naboer, udskiftes hele listen af vertices på en gang. På Billede 3 på side 44 ses resultatet af en sådan smoothing.

2.5 Neddeling

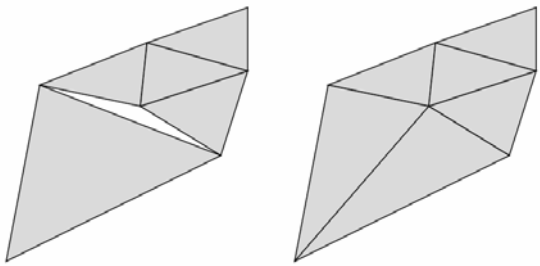
Når nu der benyttes et uniformt net har man ingen muligheder for at have lokal tilpasning af nettet i områder med stor variation, som man har ved et struktureret net. Derfor neddeler man de genererede trekanter i flere mindre trekanter, og ved at placere de nye vertices tættere på overfladen, kan man opnå en større grad af approksimation i områder med stor variation. I dette afsnit vil der kort blive beskrevet to forskellige metoder af neddeling, hvoraf √3-neddelingen *[Kobbelt]* er den som bliver brugt i dette projekt.

2.5.1 Uniform vs. Adaptiv

Der er valg man skal foretage, når man begynder at neddele et mesh i mindre trekanter, hvilket er om hvorvidt man vælger at neddele alle trekanter ligeligt, eller om man vælger kun at neddele de trekanter, der vurderes til at kunne optimere approksimationen af overfladen. Adaptiv neddeling er klart at foretrække, da man således kan undgå at neddele et mesh i områder, hvor der ikke er behov for det. Et typisk problem med adaptiv neddeling er dog, at man skal samle trekanter af forskellig grad af neddeling.



Figur 16: Illustration af 1-4 split, dyadisk neddeling. Figur lånt fra [Kobbelt]



Figur 17: Ved dyadisk adaptiv neddeling opstår der huller i meshet, når trekanter af forskellige neddelingsniveau mødes, figur lånt fra [Kobbelt]

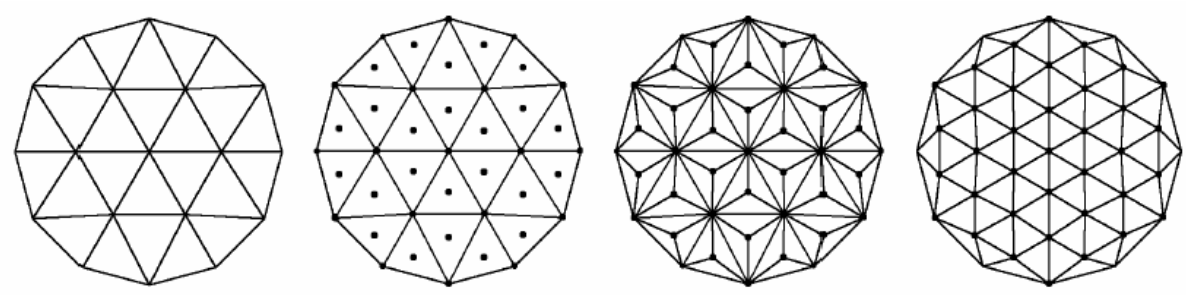
2.5.2 Dyadisk

Ved dyadisk neddeling, eller 1-4 split som det også kaldes, indsættes der en ny vertex for hver kant i en trekant. Disse vertices bliver så forbundet med hinanden, hvorved der dannes 4 nye trekanter, se Figur 16. Derefter udglattes og fittes de nye såvel som gamle vertices efter hvilke regler man nu måtte have sat op. Hvis der bliver brugt adaptiv neddeling, har denne metode den ulempe, at der opstår huller i meshet, der hvor to trekanter med forskellige neddeling deler kant. Derfor er man nødt til at erstatte den ikke-neddelte trekant med en trianglefan[8], hvilket oftest medfører grimme trekanter og dermed en forringelse af meshets kvalitet, se Figur 17.

2.5.3 √3

Ved √3-neddeling indsættes der også nye vertices, men i stedet for at placere en ny vertex på hver kant i en trekant, placeres der kun én vertex i centrum af trekanten. Dette giver flere fordele i forhold til dyadisk neddeling. For det første undgås der huller i meshet, der hvor der er trekanter af forskellige grad af neddeling, da de nye trekanter har samme kanter som deres forælder har. For det andet vokser antallet af trekanter kun med n^3 mod n^4 for dyadisk, hvilket gør at meshet neddeles langsommere om end antallet af trekanter stadig øges ekspotentielt. For et dyadisk neddelt net deles hver trekant i fire nye trekanter, for hver neddeling, mens at for √3 vil hver trekant blive delt i ni ny trekanter for hver anden neddeling. For dette projekt antages det derudover, at de gamle vertices ligger på overfladen, og det derfor kun er den nye vertex som skal fittes til overfladen, og derved skabes en bedre grad af approksimation.

Når neddelingen gentages flere gange foregår det efter følgende forløb: Hver trekant tildeles et generationsindex startende med nul. Hvis en trekant med et lige generationsindex skal neddeles, indsættes der en ny vertex i centrum, som fittes til overfladen. Hvis trekanten har et ulige antal, skal dens nabotrekant findes, og forudsat at de har samme generationsindex flippes den kant de deler, hvilket også er kanten mellem deres forældres trekanter. Derved opnås at alle nye vertices har valens 6, og gamle vertices har samme valens som de havde før neddelingen, se Figur 18.



Figur 18: Ved √3-neddeling indsættes en ny vertex i centrum af trekanten, og ved derefter flippes de gamle kanter. Figuren er lånt fra [Kobbelt]

2.6 Specielle tilfælde og situationer

I dette afsnit beskrives nogle specielle forhold, der kan skabe problemer i forbindelse med polygonaliseringen. Disse er typiske problemer som de fleste implementeringer søger at løse.

2.6.1 Yderkanter i dual – nettet

Når der arbejdes med et foruddefineret rum, hvor der ligger net-knuder på kanten, kan der opstå problemer, hvis funktionen strækker sig ud over disse grænser. Polygoniseren i dette projekt tager ikke højde for at skalere en model eller begrænse en funktion til kun at være defineret i det givne rum. Det er op til brugeren at sørge for, at dette hensyn bliver taget. Problemet ligger i, at når en net-knude bliver evalueret, og det ligger på yderkanten, så er der så og sige ikke nogen knude uden for kanten at sammenligne med, og derfor kan der ikke blive skabt nogen vertices udenfor rummet. Det bevirker selvfølgelig at hvis volumenet strækker sig derud, kommer der et hul i visualiseringen, der svarer til skæringen mellem det plan, der definerer yderkanten, og funktionen. Der er ingen simpel måde at løse dette problem på ud over at specificere hvilke grænser, rummet har, og gøre brugeren opmærksom på dette. I dette projekt er grænserne defineret som den boks som kan udstrækkes mellem (0,0,0) og (1,1,1), se evt. Billede 4 på side 45.

2.6.2 Skarpe kanter i modellen

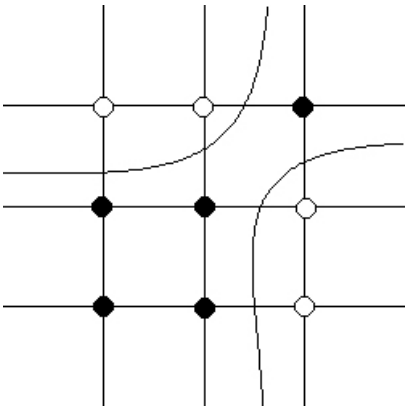
Denne polygoniser tager ikke højde for skarpe kanter i et volumen, da dens anvendelse hovedsageligt er på volumener baseret på radial basis functions, som pr. definition ikke er i stand til at skabe skarpe kanter og hjørner. Dette er dog et problem som mange polygonisere søger at løse.

2.6.3 Tvivlstilfælde

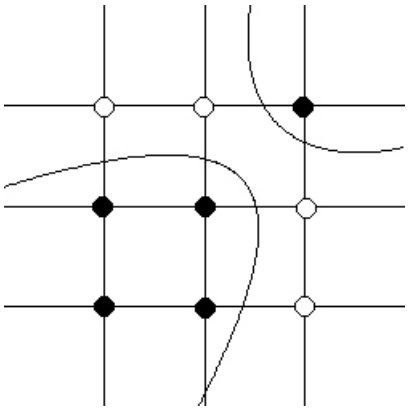
De sidste to problemer har begge at gøre med formen og størrelsen af polygonaliseringscellen.

2.6.3.1 Tvetydige celler

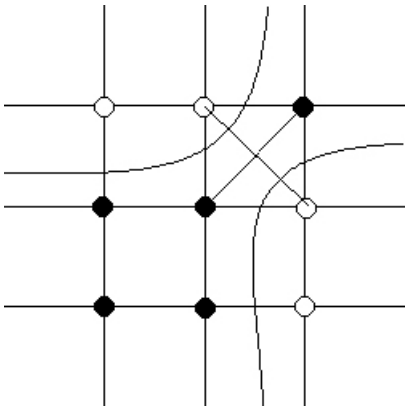
En tvetydig celle er en celle, hvor overfladen kan placeres på flere forskellige måder, se Figur 19 og Figur 20, *[Ning]*. For primære net kan man se, at der vil blive dannet to separate overflader, der vil vende enten den rigtige eller den forkerte vej. For duale net, er der et yderligere problem, idet at de to overflader kommer til at deles om den ene vertex, som er blevet indsat i den celle. Det betyder at overfladen derved kommer til at hænge sammen. Det problem kan dog løses hvis man detekterer sådan en situation ved at indsætte en ekstra vertex, men så skal man stadig bestemme sig for hvilken vej overfladen går, og tildele en vertex til hver overflade samling. En måde at løse dette problem er ved at neddele den kubiske celle i tolv tetraeder. Fordelen ved en tetraeder er, at den er entydig. Uanset hvordan den



Figur 19: Et eksempel på en tvetydig celle



Figur 20: Den anden måde overfladen kan gå gennem den tvetydige celle

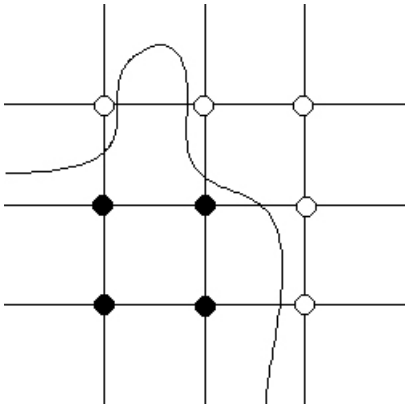


Figur 21: Ved at inddele cellen i utvetydige underceller kan man sikre sig mod problemet

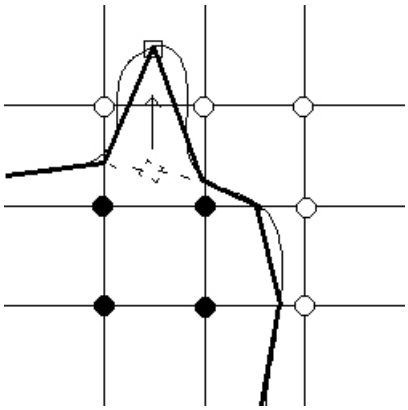
skæres, vil den altid kun kunne danne et enkelt skæringsplan, hvilket giver polygonerne. I et 2d eksempel vil det svare til at cellen bliver delt op i fire trekanter, som har fælles top i centrum af cellen, se Figur 21.

2.6.3.2 Udglatning af detaljer

En udglatning af en detalje sker når en bestemt egenskab/finesse ikke bliver fundet som følge af et net med for store celler. På Figur 22 kan man se, at overfladen skærer en kant to gange og danner en bule i den celle, den bevæger sig ind og ud af. Da begge disse skæringer ligger på en cellekant, har man ingen mulighed for at detektere den, og resultatet deraf bliver at bulen så og sige udglattes væk. Dette problem har flere løsninger. Men som en af hypoteserne i dette projekt påpeger, kan man ved hjælp af neddeling af polygonerne løse dette problem. På Figur 23 ses en illustration af neddeling, hvor den stiplede linie indikerer hvor linien lå før neddeling. Efter neddeling er linien nu i to dele, og kan tilpasses overfladen meget bedre.



Figur 22: En bule på overfladen bliver ikke fundet, da den kun befinder sig inden for en celle



Figur 23: Den nye indsatte vertex projiceres tættere på overfladen

3 Implementering

I dette afsnit beskrives den implementeringsspecifikke del af projektet - herunder funktionsbeskrivelser, programforløb m.m.
J. Andreas Bærentzens implementering af Jules Bloomenthals polygoniser er anvendt som skabelon for klassen. Koden til denne er dog ikke medtaget i rapporten, men medfølger i det samlede framework, der ligger på den medfølgende cd-rom.

3.1 Opbygning

Selve polygoniseren består kun af én klasse kaldet Polygonizer. Ud over get/set-funktioner har Polygonizer klassen kun en public funktion kaldet ***march()***. March() funktionen er ansvarlig for at starte og afvikle hele forløbet af polygonaliseringen. Polygonizer klassen har også kun én konstruktør, hvis argumenter er en implicit funktion og et antal steps. Resten af klassens variable og funktioner er private, da de kun bruges internt i klassen. Generaliseret ser det således ud:

```
class Polygonizer
{
    private:
    Private funktioner og variable

    public:

   konstruktør

    void march();

    get/set funktioner
};
```

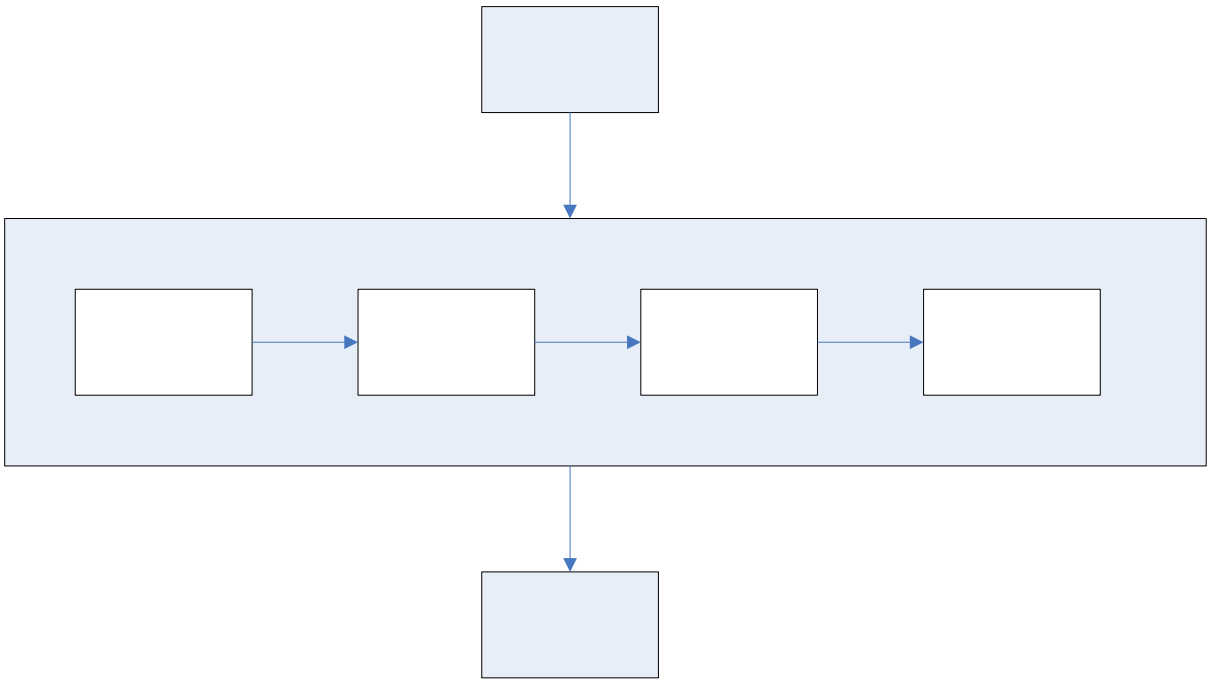
Den fulde kildekode er placeret som bilag i afsnit 7.4

3.2 Programforløb i funktionsbeskrivelse

Selve polygoniseren er kun en del af det større program, men er den eneste del, der er udviklet specifikt i dette projektforløb. Overordnet set kan forløbet visualiseres som vist på Figur 24, hvor hovedprogrammet kalder polygoniserens *march()* funktion, hvorefter polygoniseren danner et mesh, der til sidst bliver visualiseret i hovedprogrammet.

3.2.1 March()

Inde i selve polygoniser-delen i *march()* funktionen opbygges først det net af hjørner, der skal evalueres. Herefter undersøges hvert hjørne for en kant, der skæres af overfladen. Når en sådan kant findes, kaldes funktionen *build_triangles()*



Figur 24: Pipeline over programforløb

3.2.2 Build_triangles()

Denne funktion indsætter de nye vertices i de respektive celler på det duale net, men kun hvis en vertex ikke eksisterer i forvejen. Hvis en vertex eksisterer, bruges denne i stedet, og de to trekanter, der skal approksimere overfladen, dannes. Når hele meshet er blevet genereret, forsætter *march()* med at kalde funktionen *build_edge_table()*.

3.2.3 Build_edge_table()

Denne funktion opbygger en liste over hvilke kanter, de forskellige vertices har for dermed at have et opslagsværk til hurtigt at finde de vertices, som hænger sammen med hinanden. Dette bruges til udglatning senere. Herefter kalder *march()* funktionen *fit_vertices()*.

3.2.4 Fit_Vertices()

Funktionen *Fit_Vertices()* kalder, som det første, funktionen *smooth_vertices()*, der skal udglatte alle de eksisterende vertices. Herefter køres en løkke for hvert vertex, der stopper efter et af to kriterier: enten er vertexet blevet placeret inden for en given afstand af overfladen givet ud fra et fastsat kriterium, eller også er løkken blev kørt det maksimale antal gange. Inde i løkken findes gradienten for det specifikke vertex, og vertexet bliver derefter flyttet et step i retning mod overfladen.

3.2.5 Smooth_vertices()

Denne funktion gør kort og godt det, at den slår op i edge tabellen for hvert vertex og finder alle de vertices, hvormed det deler en kant. Herefter beregner den et midtpunkt for alle disse vertices tilsammen, og dette punkt bliver nu det specifikke vertexs nye position. Dette punkt skubbes nu ind i en midlertidig liste, indtil alle punkter er blevet processeret. Dette gøres for at skabe en middelværdi i forhold til de gamle placeringer for vertices, således at alle vertices behandles ens. Når alle vertices er blevet behandlet, skiftes den gamle vertex-liste ud med den midlertidige, og overfladen er nu blevet smoothed.

3.2.6 Subdivide()

Denne funktion er den sidste, der køres fra *march()* funktionen, og sørger for at neddele de trekanter, der har behov herfor. Funktionen kører yderst i en løkke som angiver den tilladte mængde af niveauer neddeling. Inde i løkken køres lineært igennem alle trekanter og for hver trekant findes dens center, som derefter evalueres til, hvor tæt det ligger på overfladen. Hvis den ligger uden for succeskriteriet, neddeles trekanten ved at kalde funktionen *split_triangle()*.

3.2.7 Split_triangle()

Hver trekant har en generation vedhæftet, således at startgenerationen er nul, og for hver gang en trekant deles bliver de nye trekanter tildelt en generation, der er en højere end dens forælder. Hvis en trekants generation er et lige tal, neddeles trekanten ved at indsætte en ny vertex i midten af trekanten, hvorefter denne projiceres tættere på overfladen. Hvis en trekants generation derimod er et ulige tal, undersøges det, om den første nabo, der findes ved at kalde funktionen *find_neighbour()*, har samme generation. Hvis dette er tilfældet flippes den originale kant mellem disse to trekanters forældre, ved at kalde funktionen *swap_edges()*.

3.2.8 Swap_edges()

Denne funktion vender en kant mellem to trekanter med ulige generation, således at kanten i stedet går mellem forældre-trekanternes centre. På denne måde sikres det, at såfremt alle 1. generations trekanter i en forælder neddeles til 2. generation, vil både den gamle og den nye vertex i forælderen have valens 6.

3.2.9 Find_neighbour()

Funktionen returnerer et indeks til den trekant, der har den kant der er modsat rettet kanten i den trekant, hvis index man gav som argument, derved ved man at det er nabotrekanten man har fat i. Hvis der ingen nabo trekant er, f.eks. i tilfælde af en yderkant i nettet, returneres -1 i stedet for.

4 Test specifikationer

I dette afsnit specificeres kort de forskellige test der er blevet udført, samt med hvilket formål. Samtlige tests er kørt på egen maskine med følgende specifikation:

CPU: AMD Athlon XP 2800+
Ram: 512 Mb
HD: 80 Gb IBM
Grafikkort: Asus V9980, NVidia GeForce 5950 Ultra chipset

4.1 Hastighedstests

4.1.1 Test 1 – general hastighedssammenligning

Formål:

At finde ud af ved sammenligning med J.Andreas Bærentzens implementering af Jules Bloomenthals polygoniser, om hvorvidt egen implementering er generelt hurtigere eller langsommere.

Specifikation:

For at give et generelt indblik i hastigheder, skal der måles tider baseret på et antal af forskellige funktioner/modeller, forskellige antal steps og forskellige grader af neddeling.

4.2 Test af udglatning/smoothing

4.2.1 Test 2 – vurdering af kvaliteten af udglatning

Formål:

At vurdere om meshet generelt er blevet pænere, dvs. slivers så vidt muligt er fjernet. Og dermed vise at denne implementering genererer pænere meshes end Bloomenthals polygoniser.

Specifikation:

Ved at beregne den gennemsnitlige mindste vinkel for trekanterne kan man afgøre om det er blevet færre slivers. I en pæn trekant vil vinklerne være forholdsvis ens størrelse altså omkring 60° mens i en sliver vil den mindste vinkel måske være tættere på 5°, altså jo mindre den gennemsnitlige mindste vinkel er, jo flere slivers er der. Afslut med en visuel sammenligning mellem de to polygonisere omtalt i test 1.

4.2.2 Test 3 – vurdering af gentagne kørsler af udglatningsalgoritmen

Formål:

At vurdere om det er tilstrækkeligt med én kørsel af udglatningsalgoritmen, da vertices formentlig har flyttet sig væsentligt efter at være blevet fittet til overfladen.

Specifikation:

Kør polygoniseren har et antal forskellige funktioner/modeller og et forskellige antal gentagelser af udglatning. Gør brug af metoden fra test 2. Vurder evt. visuelt om det gør nogen forskel om hvorvidt der køres en eller flere udglatninger.

4.3 Test af neddeling

4.3.1 Test 4 – vurdering af meshets approksimation af overfladen

Formål:

At vurdere om det endelige mesh er en bedre approksimation end ved brug af bloomenthals polygoniser og om der er flere detaljer i overfladen efter neddeling.

Specifikation:

Beregn en middel afvigelse af trekanters centre i forhold til overfladen, en mindre afvigelse bør betyde en bedre approksimation. Dette gøres for flere forskellige funktioner/modeller, forskellige antal steps og neddelingsniveau. Testen afsluttes med en visuel vurdering af hvorvidt overfladen er en bedre approksimation.

4.3.2 Test 5 – Sammenligning af antallet af trekanter

Formål:

At vurdere om min polygoniser generelt genererer færre trekanter end Bloomenthals, når der altså ikke benyttes neddeling, der ville øge antallet af trekanter ekspotentielt.

Specifikation:

Tæl antallet at trekanter i det endelige mesh for henholdsvis egen og Bloomenthals polygoniser, ved flere forskellige modeller og antal steps.

5 Resultater

5.1 Test 1

5.1.1 Resultater

Tider målt i sekunder					
	Steps				
model	10	20	30	40	50
Weird3, Jakobsen - ingen neddeling	0,039	0,171	0,444	0,877	1,549
Weird3, Jakobsen - 2 niveauer neddeling	0,058	0,309	0,758	1,454	2,496
Weird3, Jakobsen - 4 niveauer neddeling	0,060	0,310	0,811	1,527	2,565
Weird3, Jakobsen - 6 niveauer neddeling	0,062	0,329	0,816	1,556	2,665
Weird3, Bloomenthal	0,237	0,417	0,874	1,311	2,068
2 Kugler; Jakobsen - ingen neddeling	0,001	0,007	0,011	0,025	0,042
2 Kugler; Jakobsen - 2 niveauer neddeling	0,002	0,008	0,023	0,044	0,075
2 Kugler; Jakobsen - 4 niveauer neddeling	0,003	0,010	0,030	0,062	0,097
2 Kugler; Jakobsen - 6 niveauer neddeling	0,004	0,017	0,040	0,082	0,125
2 Kugler, Bloomenthal	0,123	0,121	0,227	0,293	0,332
Bunny, Jakobsen - ingen neddeling	0,113	0,646	1,757	3,504	6,268
Bunny, Jakobsen - 2 niveauer neddeling	0,399	1,710	3,498	6,211	9,977
Bunny, Jakobsen - 4 niveauer neddeling	0,794	3,065	4,945	7,556	11,167
Bunny, Jakobsen - 6 niveauer neddeling	1,383	5,190	6,879	8,978	12,469
Bunny, Bloomenthal	0,392	1,136	2,401	4,085	6,282
Bunny, Jakobsen - 6 niveauer neddeling, med smoothing under neddeling	1,920	6,730	8,554	11,060	13,707
Bunny, Jakobsen - 6 niveauer neddeling, andet kriterium	3,059	10,470	14,880	18,608	22,222

Tabel 1: Sammenligning af kørselstider, for forskellige modeller

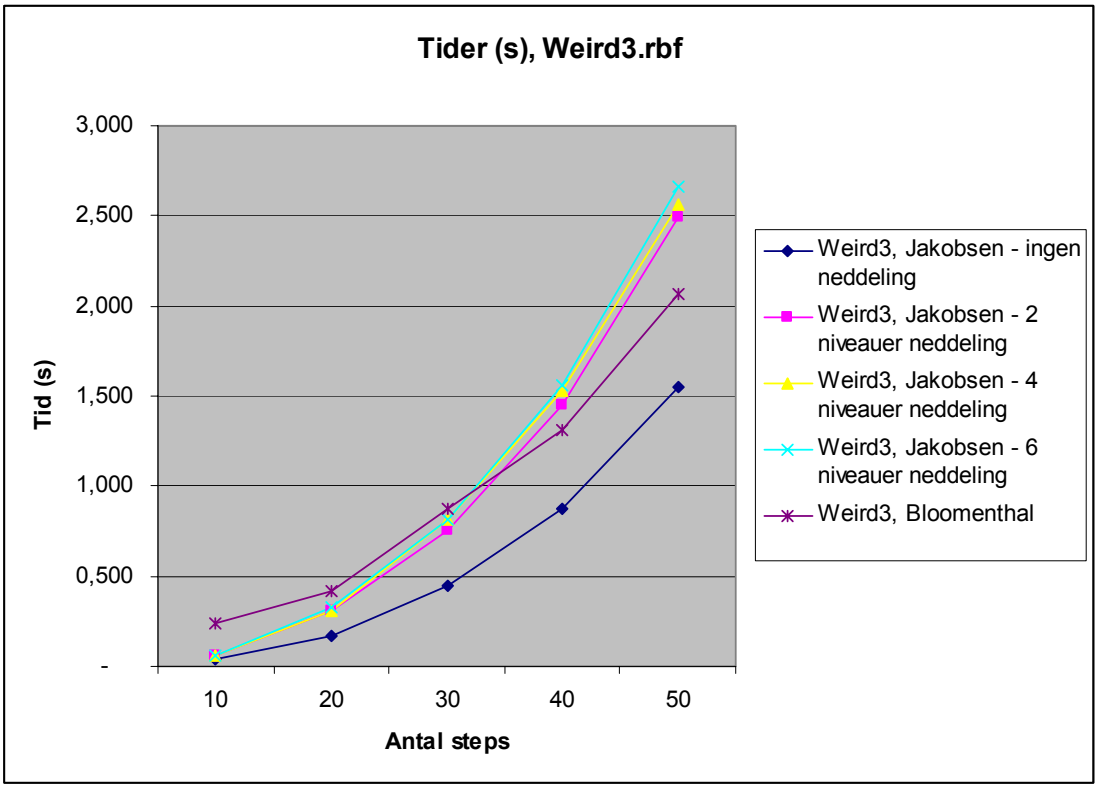


Diagram 1: Sammenligning af kørselstider for weird3.rbf

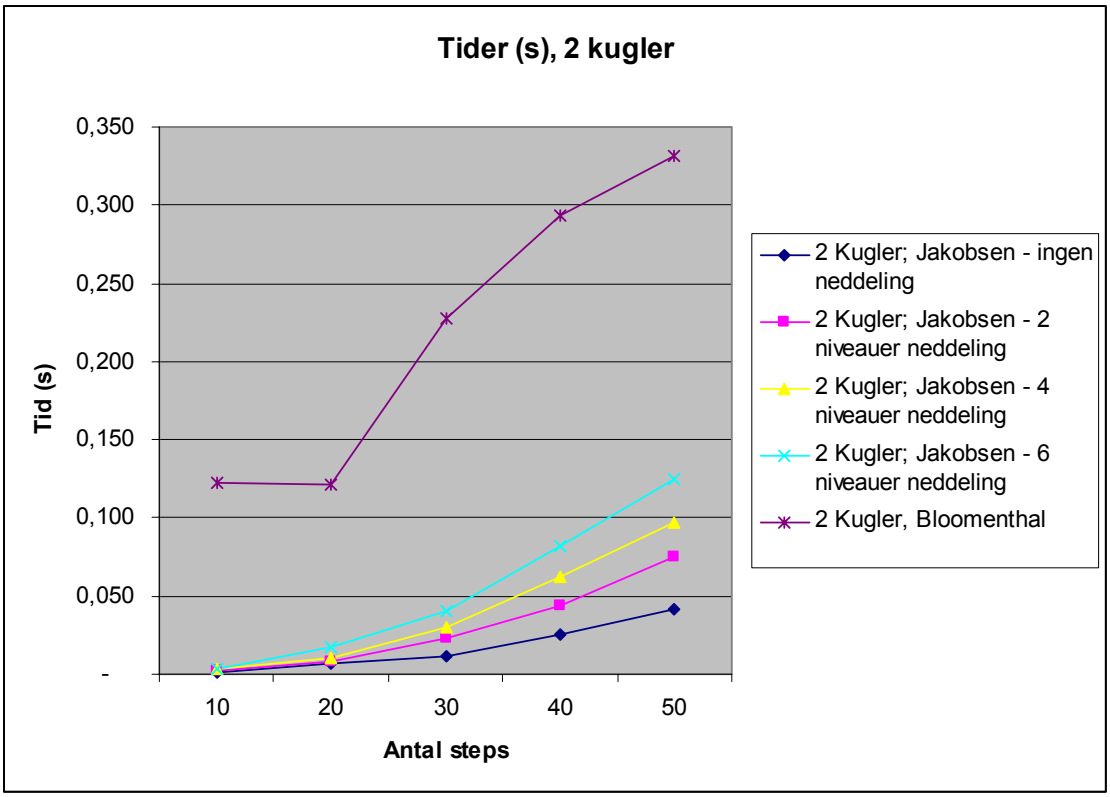


Diagram 2: Sammenligning af kørselstider for 2 kugler

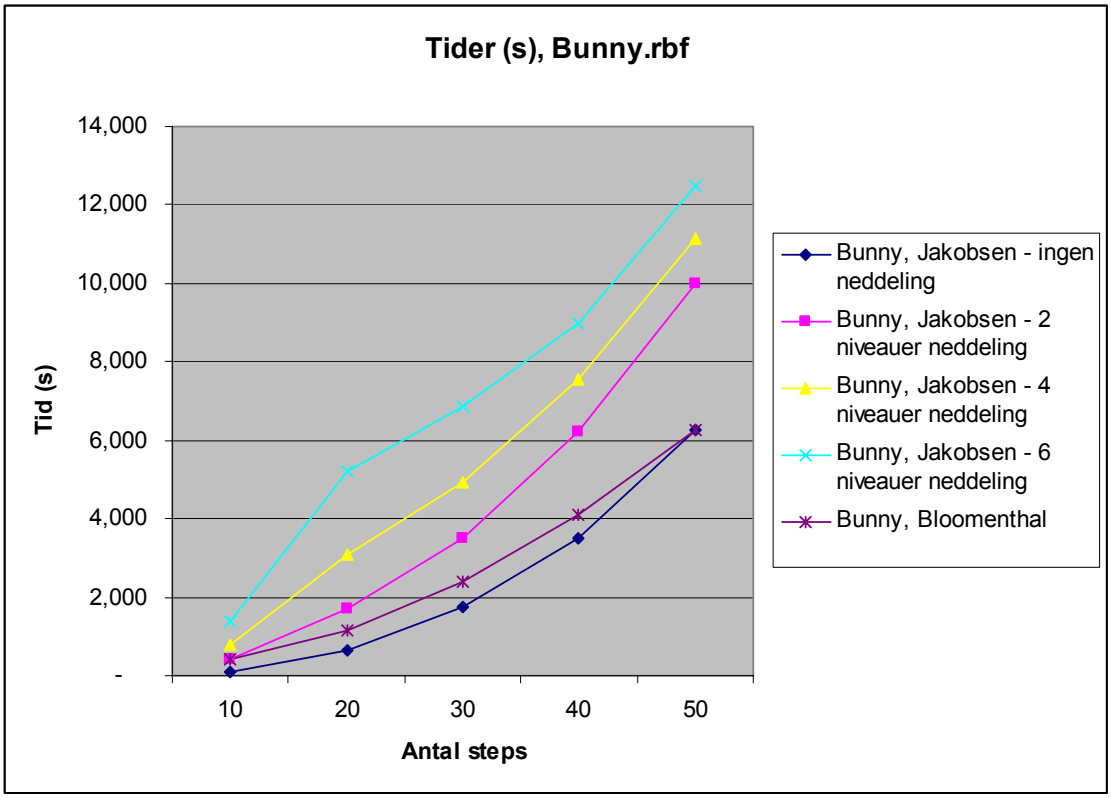


Diagram 3: Sammenligning af kørselstider for bunny.rbf

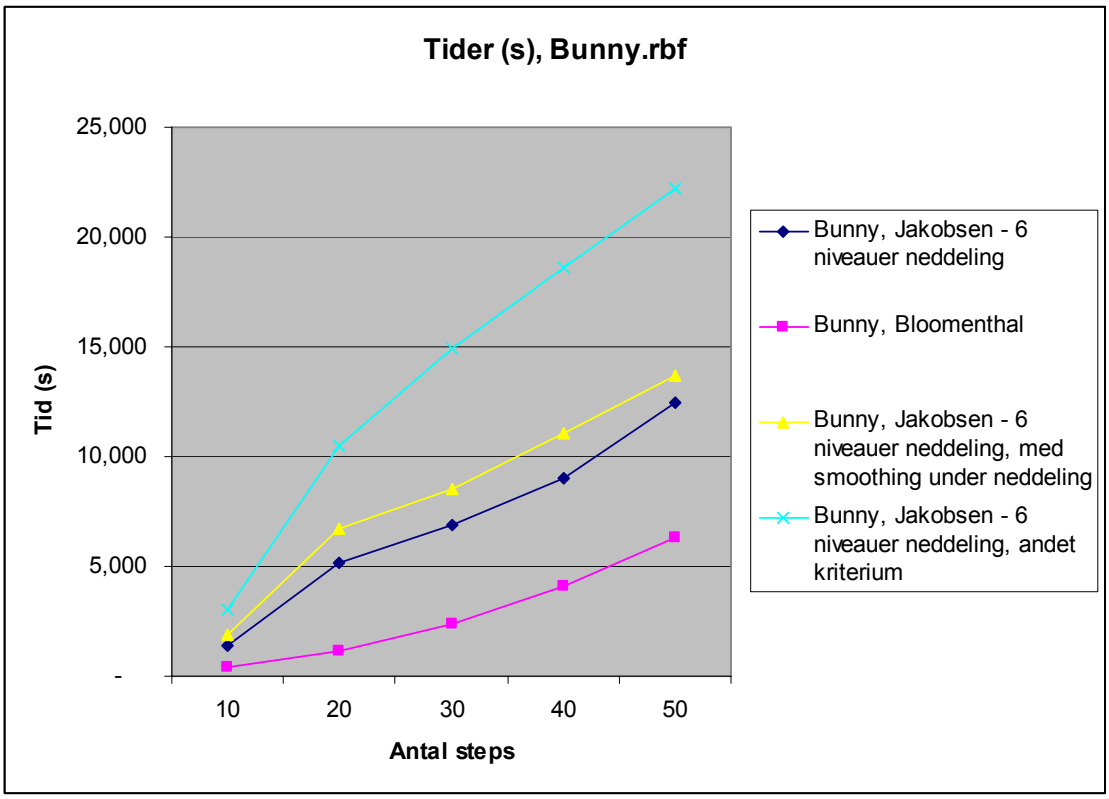


Diagram 4: Sammenligning af kørselstider for Bunny.rbf med forskellige modifikationer

5.1.2 Behandling

Generelt på Diagram 1 til og med Diagram 3 ses kørselstider for forskellige modeller, med henholdsvis egen og Bloomenthals polygoniser. Generelt kan man se at egen implementering er hurtigere, når der ikke benyttes neddeling, men så snart der begyndes på neddeling stiger behandlingstiderne. Og som også vist, kan man aflæse at jo mere neddeling der tilføjes, jo længere tid tager det.

På Diagram 4 er det skåret ud i pap, hvor man virkelig kan se, at når der er meget neddeling i et mesh, så stiger tiderne voldsomt.

Selv om denne test, står med nummer et, er tallene dog taget sideløbende med de andre test, så mht. smoothing under neddeling kommer der en forklaring dertil under test 2. Og mht. et andet kriterium for neddeling, så er der ændret på kriteriet for neddeling for at skabe mere neddeling.

Konklusionen på hastighedssammenligningerne må være, at uden neddeling er egen implementering hurtigst, men som forventet har neddelingen den bagdel at være tidskrævende. Så man må gøre op med sig selv om man vil have et pænere og mere nøjagtigt mesh eller om man vil have det hurtigt.

5.2 Test 2

5.2.1 Resultater

Gennemsnits minimum vinkel i grader					
	steps				
Model	10	20	30	40	50
Weird3, Jakobsen - ingen neddeling	41,53	40,88	40,83	40,69	40,68
2 Kugler; Jakobsen - ingen neddeling	40,85	40,62	40,34	40,34	40,32
Bunny, Jakobsen - ingen neddeling	41,04	40,01	40,33	40,76	40,86
Weird3, Jakobsen - 2 niveauer neddeling	41,53	40,88	40,83	40,69	40,68
2 Kugler; Jakobsen - 2 niveauer neddeling	35,31	39,91	40,14	40,23	40,26
Bunny, Jakobsen - 2 niveauer neddeling	36,31	36,67	38,56	39,40	40,08
Weird3, Jakobsen - 4 niveauer neddeling	41,53	40,88	40,83	40,69	40,68
2 Kugler; Jakobsen - 4 niveauer neddeling	33,48	39,77	40,14	40,10	39,97
Bunny, Jakobsen - 4 niveauer neddeling	35,49	35,13	37,35	38,37	39,38
Weird3, Jakobsen - 6 niveauer neddeling	41,53	40,88	40,83	40,69	40,68
2 Kugler; Jakobsen - 6 niveauer neddeling	33,39	39,60	40,14	39,93	39,77
Bunny, Jakobsen - 6 niveauer neddeling	33,30	32,72	35,69	37,29	38,60
Weird3, Bloomenthal	32,55	32,50	32,67	32,66	32,40
2 Kugler, Bloomenthal	36,00	31,19	31,49	31,35	31,53
Bunny, Bloomenthal	32,13	32,22	32,07	32,40	32,14
Bunny, Jakobsen - 6 niveauer neddeling, med smoothing efter neddeling	35,04	34,57	37,05	38,56	39,76
Bunny, Jakobsen - 6 niveauer neddeling, med smoothing under neddeling	39,41	38,56	39,89	40,28	40,92

Tabel 2: Sammenligning af gennemsnits mindste vinkler for adskillige modeller og neddelingsniveauer

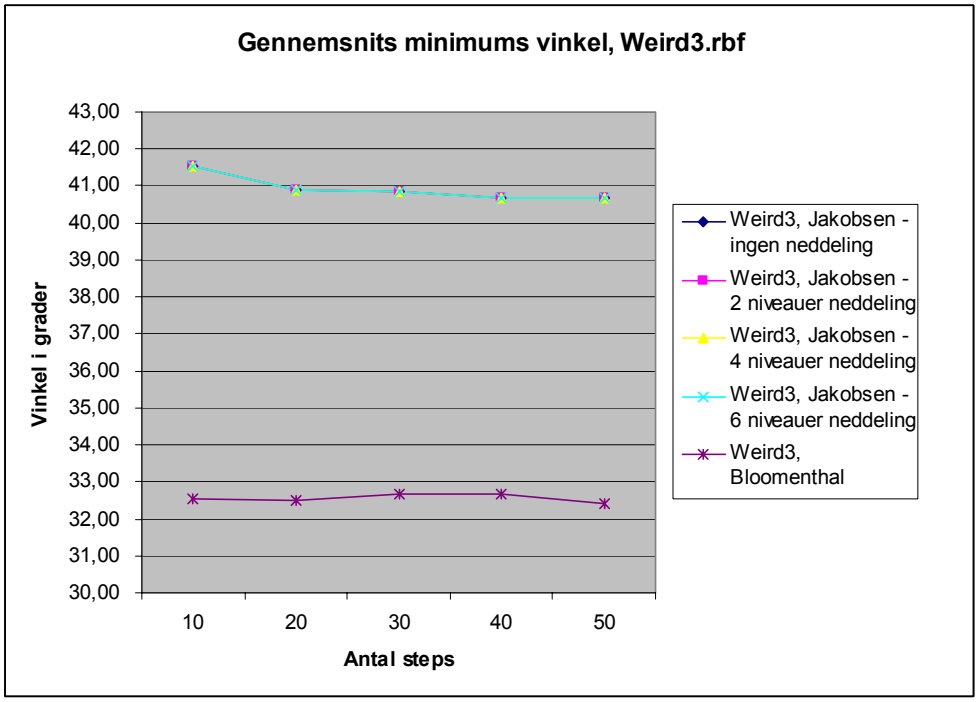


Diagram 5: Gennemsnits mindste vinkel for mesh for modellen weird3.rbf

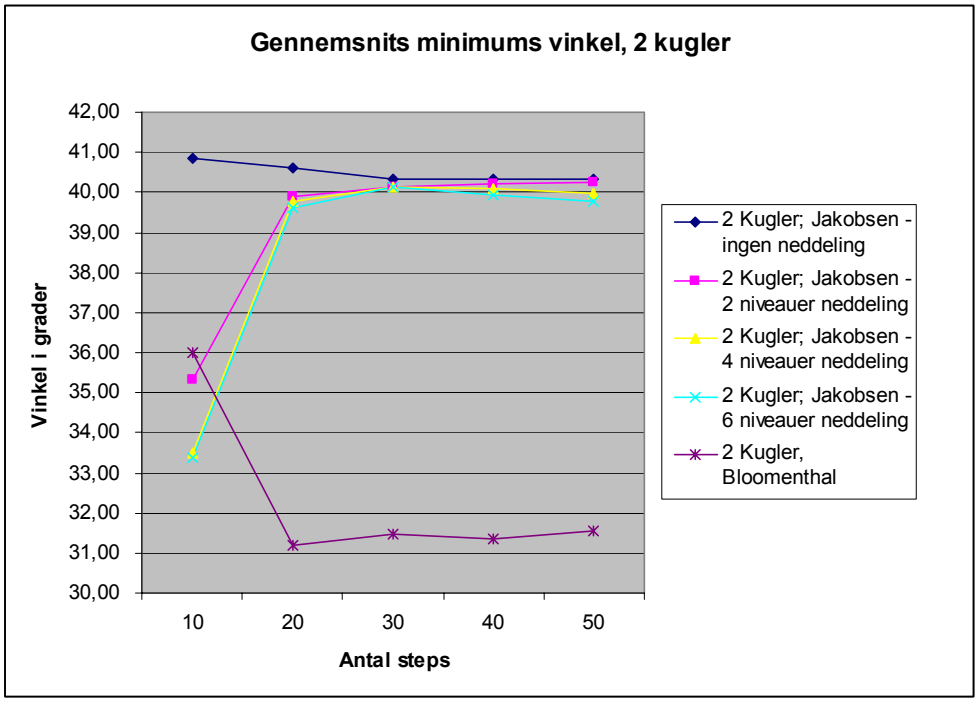


Diagram 6: Gennemsnits mindste vinkel for mesh for funktion af to sammen-smeltede kugler

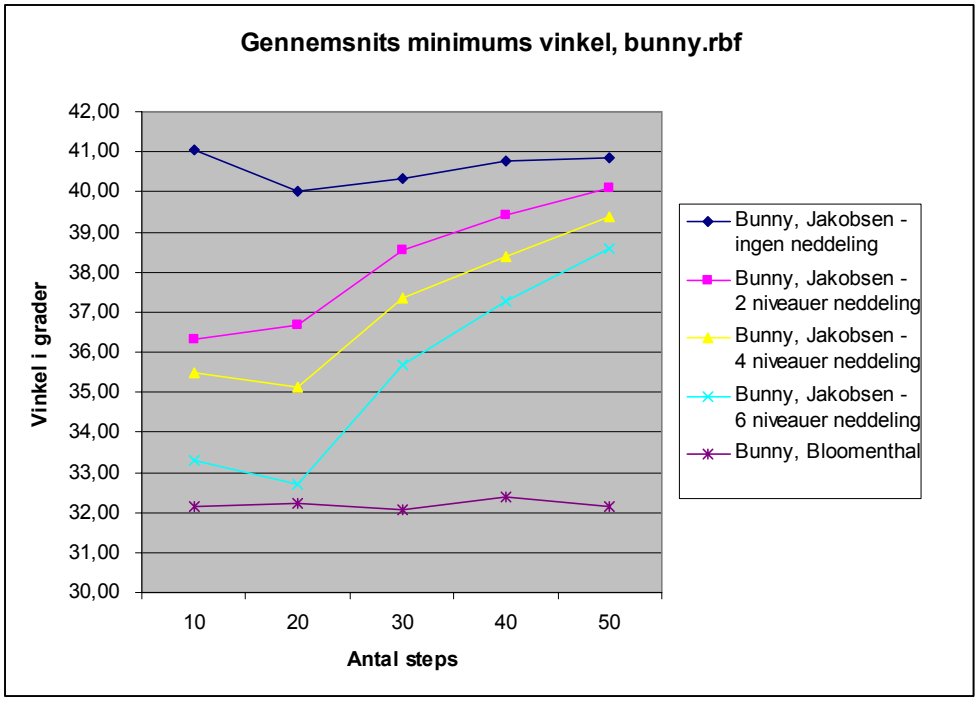


Diagram 7: Gennemsnits mindste vinkel for mesh for modellen bunny.rbf

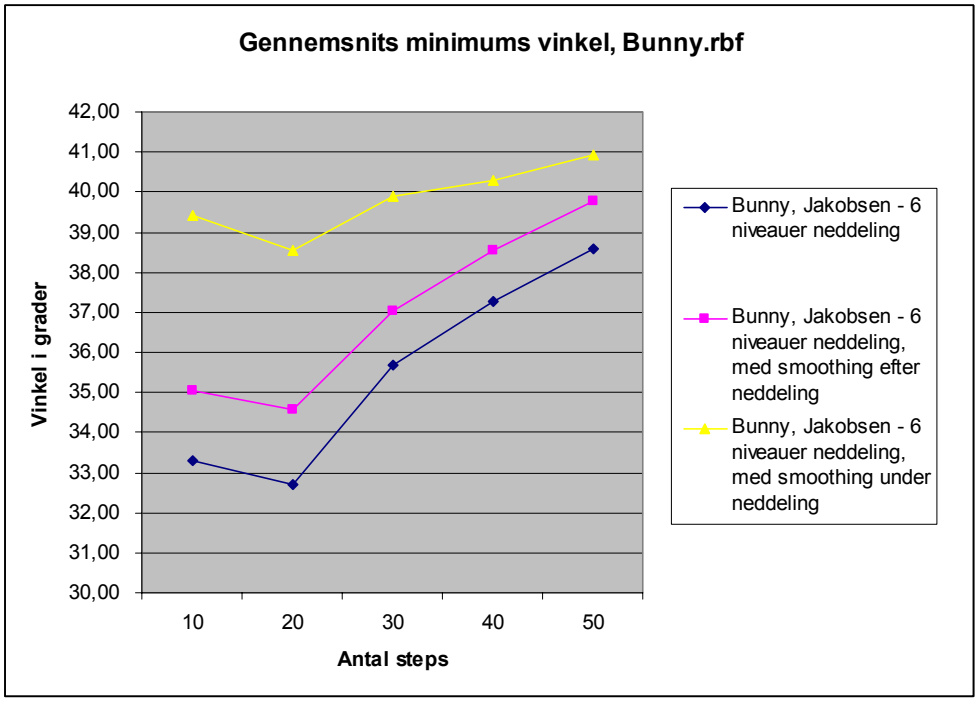


Diagram 8: Gennemsnits mindste vinkel, incl. smoothing enten efter eller under neddelingen

5.2.2 Behandling

Diagram 5 til og med Diagram 7 viser den gennemsnitlige vinkel for det endelige mesh, for 3 forskellige modeller, og for dette projekts implementering yderligere også for forskellige grader af neddeling.

Generelt formodes det at ved et lavt antal af steps, dvs. en stor polygonaliseringscelle, vil der være flere trekanter som er blevet neddelt, uanset hvor mange niveauer af neddeling der er blevet brugt. Derfor kan man ud fra resultaterne se at jo mere neddelt en model er blevet, jo grimmere er meshet blevet. Men det skal også bemærkes at denne polygoniser generelt danner bedre meshes end Bloomenthals, da stort set samtlige værdier er større.

På Diagram 5 ses at samtlige værdier er ens uanset neddelingsniveau, det formodes at det skyldes at modellerne slet ikke er blevet neddelt, formentlig pga. dårligt sat succeskriterium.

På Billede 5 til og med Billede 10 kan ses sammenligninger fra de tre modeller med henholdsvis den ene og den anden polygoniser. Her kan man på Bloomenthals billeder tydeligt se hvordan slivers bliver dannet typisk rundt om de områder, der har en niveauforskel i rummet.

På Diagram 8 vises en sammenligning for hvad der sker med den gennemsnitlige minimums vinkel, når man benytter udglatningsalgoritmen henholdsvis under og efter neddelingen mod slet ikke at benytte den. Det viser sig at hvis algoritmen benyttes mellem hvert neddelingsniveau i neddelingen, så opnår man et langt pænere mesh. Hvis algoritmen kun benyttes efter neddelingen, ses også en lille forbedring. Dog tager det også forholdsvis længere tid hvis algoritmen benyttes. På Billede 10 og Billede 11 kan der ses en sammenligning, hvor der er blevet brugt udglatning mellem hvert neddelingsniveau på det sidste billede, kan man se specielt omkring poterne at meshet bliver væsentligt pænere.

Det skal dog nævnes at der er en væsentlig risiko for at man kommer til at udglatte detaljer, hvis polygonaliseringscellen er forholdsvis stor. For hvis en trekanterne er forholdvis stor bliver dens vertices også flyttet længere ved udglatning, og dermed kan det risikere at virke modsat at fittingen til overfladen.

Det kan herudaf konkluderes, at udglatningsalgoritmen danner et pænere mesh med færre slivers.

5.3 Test 3

5.3.1 Resultater

Gennemsnits minimum vinkel			
6 neddelingsniveauer	**Steps**		
model	20	30	40
Weird3, Jakobsen, 1 smoothing kørsel	40,88	40,83	40,69
Weird3, Jakobsen, 3 smoothing kørsler	41,41	41,47	41,26
Weird3, Jakobsen, 5 smoothing kørsler	41,46	41,50	41,25
2 Kugler; Jakobsen, 1 smoothing kørsel	39,60	40,14	39,93
2 Kugler; Jakobsen, 3 smoothing kørsler	39,94	40,70	40,64
2 Kugler; Jakobsen, 5 smoothing kørsler	40,22	40,94	40,43
Bunny, Jakobsen, 1 smoothing kørsel	32,73	35,63	37,27
Bunny, Jakobsen, 3 smoothing kørsler	33,66	36,57	38,23
Bunny, Jakobsen, 5 smoothing kørsler	33,94	36,84	38,66

Tabel 3: Sammenligning af gennemsnits mindste vinkler for adskillige modeller og forskellige antal kørsler af udglatning

5.3.2 Behandling

Det anses ikke for nødvendigt at have diagrammer for at se kendsgerningerne i disse data. Den gennemsnitlige mindste vinkel i meshet ændres ikke synderligt uanset hvor mange gange udglatningsalgoritmen køres. Derfor er det mest hensigtsmæssige kun at køre algoritmen en gang inden vertices bliver fittet til overfladen.

5.4 Test 4

5.4.1 Resultater

Gennensnit vurdering af trekanters centre					
	Steps				
model	10	20	30	40	50
Weird3, Jakobsen - ingen neddeling	0,00401	0,00086	0,00036	0,00020	0,00013
Weird3, Jakobsen - 2 niveauer neddeling	0,00202	0,00086	0,00036	0,00020	0,00013
Weird3, Jakobsen - 4 niveauer neddeling	0,00197	0,00086	0,00036	0,00020	0,00013
Weird3, Jakobsen - 6 niveauer neddeling	0,00197	0,00086	0,00036	0,00020	0,00013
Weird3, Bloomenthal	0,00263	0,00065	0,00029	0,00016	0,00010
2 Kugler; Jakobsen - ingen neddeling	0,00567	0,00141	0,00060	0,00036	0,00024
2 Kugler; Jakobsen - 2 niveauer neddeling	0,00418	0,00138	0,00060	0,00035	0,00024
2 Kugler; Jakobsen - 4 niveauer neddeling	0,00376	0,00136	0,00060	0,00035	0,00024
2 Kugler; Jakobsen - 6 niveauer neddeling	0,00373	0,00135	0,00060	0,00035	0,00024
2 Kugler, Bloomenthal	0,00522	0,00126	0,00058	0,00031	0,00021
Bunny, Jakobsen - ingen neddeling	0,00987	0,00378	0,00142	0,00074	0,00045
Bunny, Jakobsen - 2 niveauer neddeling	0,00617	0,00271	0,00119	0,00066	0,00043
Bunny, Jakobsen - 4 niveauer neddeling	0,00401	0,00192	0,00096	0,00059	0,00040
Bunny, Jakobsen - 6 niveauer neddeling	0,00282	0,00140	0,00081	0,00054	0,00038
Bunny, Bloomenthal	0,00829	0,00242	0,00097	0,00053	0,00033
Bunny, Jakobsen - 6 niveauer neddeling, andet kriterium	0,00173	0,00094	0,00052	0,00037	0,00030

Tabel 4: Sammenligning af vurderinger af trekanters centrum for alle trekanter i meshet

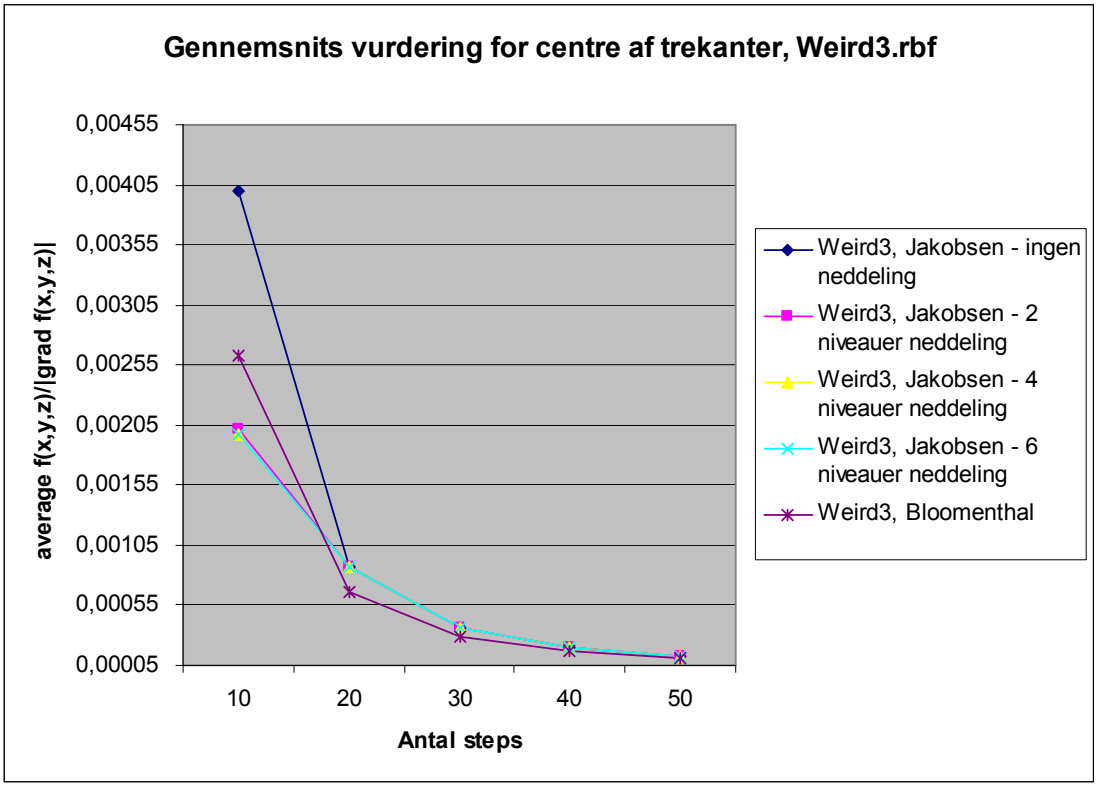


Diagram 9: Gennemsnits vurdering af trekanters centre, for modellen Weird3.rbf

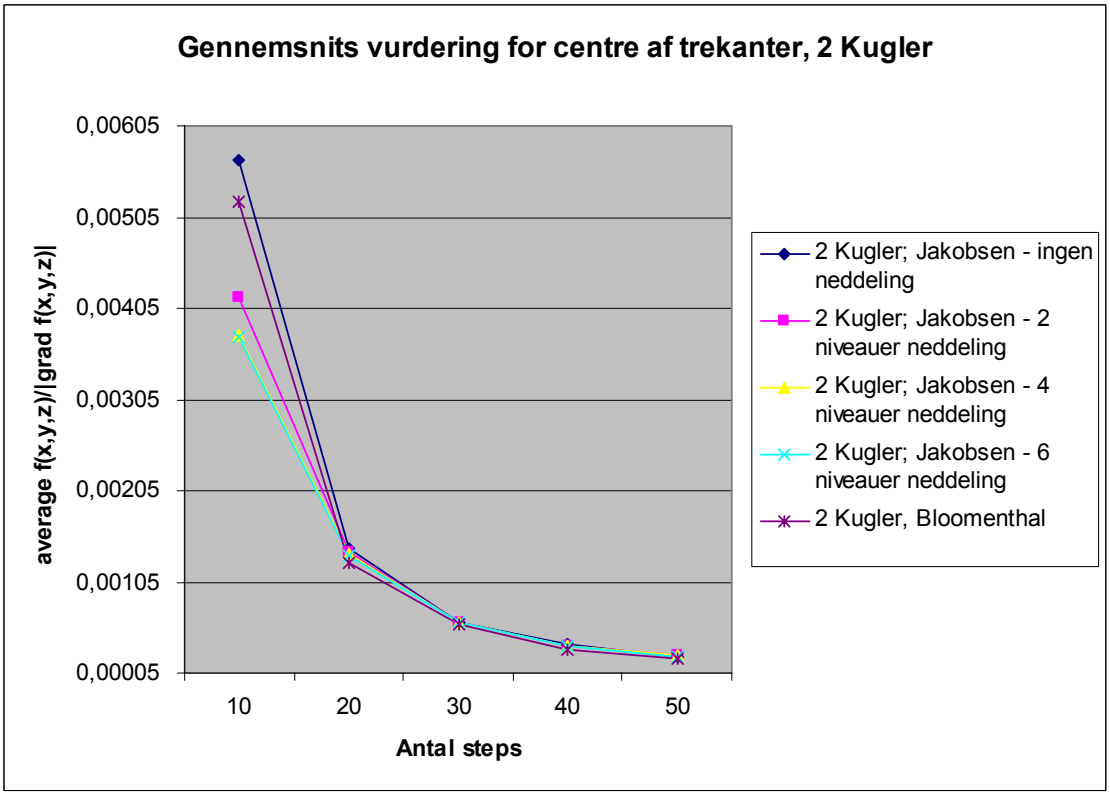


Diagram 10: Gennemsnits vurdering af trekanters centre, for funktionen af 2 kugler

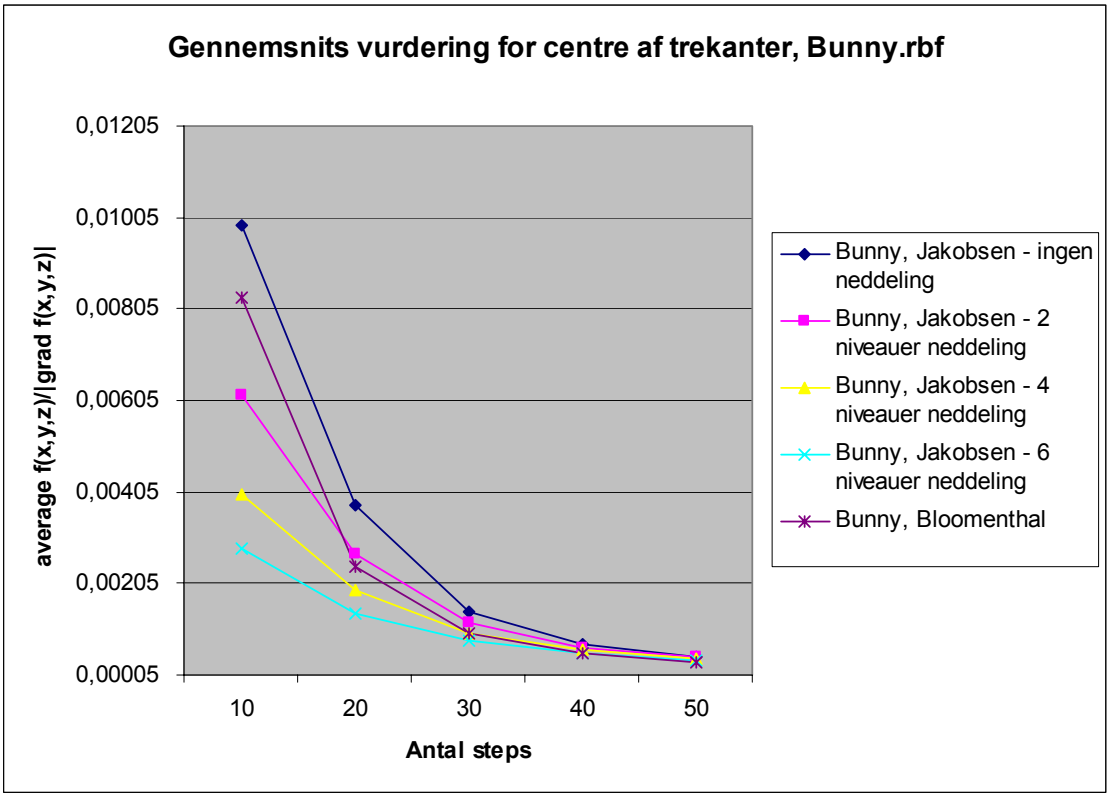


Diagram 11: Gennemsnits vurdering af trekanters centre, for modellen Bunny.rbf

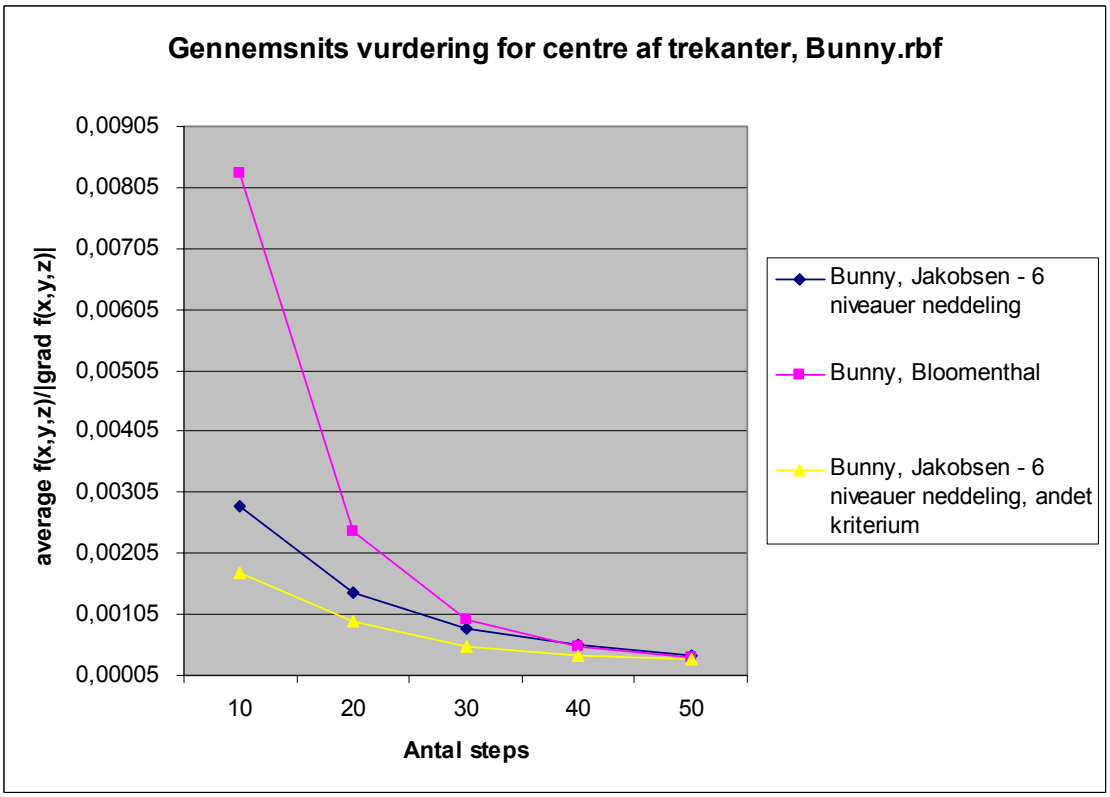


Diagram 12: Gennemsnits vurdering af trekanters centre, for modellen Bunny.rbf + en sammenligning for neddeling med et andet neddelingskriterium

5.4.2 Behandling

På Diagram 9 til og med Diagram 11 ses en sammenligning af gennemsnitlige vurderinger for trekanternes centre i et mesh. Grunden til at der er valgt centrene er, at de eksisterende vertices, må formodes at være placeret på overfladen eller meget tæt på, mens at centrene kan være dårligere placeret i forhold til overfladen. Til gengæld når en trekant bliver neddelt, vil centrene i de nye trekanter formodes til ligeledes at ligge tættere på overfladen.

En vurdering er gjort ud fra udtrykket: $\frac{f(\bar{x})}{|\nabla f(\bar{x})|}$ som også er det udtryk som bruges når der fittes til overfladen. Det angiver en approksimativ vurdering af afstanden til overfladen, da vi forudsætter, at vi er så tæt på overfladen, at den formodes at kunne lokalt approksimeres som lineær.

Af førnævnte diagrammer kan aflæses, at min polygoniser generelt har et dårligere fittet mesh, når der ikke bruges neddeling, men så snart neddelingen tages i brug, kan det ses en væsentlig forbedring. Man kan ligeledes se at forbedringerne at mest drastiske ved et lille antal steps, altså ved store polygonaliseringsceller. Hvilket også var at formode, og deraf kan også uddrages at jo mindre cellerne bliver, jo mindre neddeling er der også. På Diagram 12 kan det ses lidt mere tydeligt, hvor der også er en sammenligning med en endnu mere neddelt model, da der er brugt et andet neddelingskriterium. På Billede 12 ses Bunny'en neddelt med det nye kriterium.

Det kan heraf konkluderes, at neddelingen af meshet, forbedrer meshets approksimering af overfladen, og at denne forbedring er størst hvis polygonaliseringscellerne er store.

5.5 Test 5

5.5.1 Resultater

Antal trekanter					
	Steps				
model	10	20	30	40	50
Weird3, Jakobsen - ingen neddeling	720	3.408	8.112	14.656	23.238
Weird3, Bloomenthal	1.296	5.200	11.668	20.732	32.416
2 Kugler, Jakobsen - ingen neddeling	288	1.200	2.992	5.392	8.496
2 Kugler, Bloomenthal	396	1.892	4.208	7.500	11.752
Bunny, Jakobsen - ingen neddeling	408	2.308	5.678	10.214	16.218
Bunny, Bloomenthal	1.012	4.196	9.540	16.760	26.392

Tabel 5: Sammenligning af antallet af trekanter mellem egen og Bloomenthals polygoniser

5.5.2 Behandling

Det læses direkte af tabellen, at denne polygoniser generelt danner færre trekanter, når der ikke benyttes neddeling. Det er nok også med til at danne grundlag for, hvorfor den er hurtigere end Bloomenthals, når der ikke benyttes neddeling.

6 Konklusion

Baseret hovedsageligt på resultaterne fra test 2 og test 4 må det konkluderes, at de to hypoteser holder. Det bliver genereret et pænere mesh ved at tilføje en udglatningsalgoritme, og den behøves kun at køre én gang. Derudover er der en væsentlig forbedring i meshets approksimation af overfladen, når der bliver benyttet √3-neddeling, og i særdeleshed ved store polygonaliseringsceller. Det skal dog bemærkes at kørselstiderne bliver væsentlig forhøjet når der benyttes neddeling, og det er noget man må tage højde for, hvis man ønsker hastighed frem for præcision.

6.1 Fremtidigt arbejde

Der er nogle løse ender som mangler at bliver løst og/eller implementeret. Der er der ingen håndtering af tvivlsceller, hvor man altså ikke ved hvorledes overfladen gebærder sig, dette kan som beskrevet i teoriafsnittet løses med neddeling af cellen i tetraeder. Der er ikke implementeret nogen hensynstagen til mulige 0-gradienten, hvilket kan forskyde vertices til utilsigtede positioner. Til sidst mangler der også en del-implementering til √3-neddelingen, hvilket er, hvis der skal neddeles en trekant med ulige generationsindex dvs. der skal flippes en kant, og dens nabo ikke har samme generationsindex, så skal naboen neddeles før kanten kan flippes.

7 Bilag

7.1 Kildeliste

[Lorensen] William E. Lorensen and Harvey E. Cline, Marching Cubes: A High Resolution 3D Surface Construction Algorithm, *Computer Graphics (Proceedings of SIGGRAPH '87), Vol. 21, No. 4, pp. 163-169*

[Turk] Greg Turk & James F. O'Brian, Modelling with Implicit Surfaces that Interpolate, *Appears in ACM Transactions on Graphics, Vol. 21 No. 4, October 2002*

[Carr]: Carr, J. C., Beatson, R. K., Cherrie, J. B., Mitchell, T. J., Fright, W. R., McCallum, B. C., and Evans, T. R. 2001. Reconstruction and representation of 3d objects with radial basis functions. *In Proceedings of ACM SIGGRAPH 2001, ACM, 67-76.*

[Reuter]: Patrick Reuter, Ireneusz Tobor, Christophe Schlick, Sebastian Dedieu, Point-based Modelling and Rendering using Radial Basis Functions.
LaBRI – Université Bordeaux I – France

[Bloomenthal]: Jules Bloomenthal, An Implicit Surface Polygonizer

[Bloomenthal2]: Jules Bloomenthal, Polygonization of Implicit Surfaces

[Bloomenthal3]: Jules Bloomenthal and Keith Ferguson, Polygonization of Non-manifold Implicit Surfaces. *Department of Computer Science, The University of Calgary*

[Ning] Paul Ning and Jules Bloomenthal, An evaluation of Implicit Surface Tilers

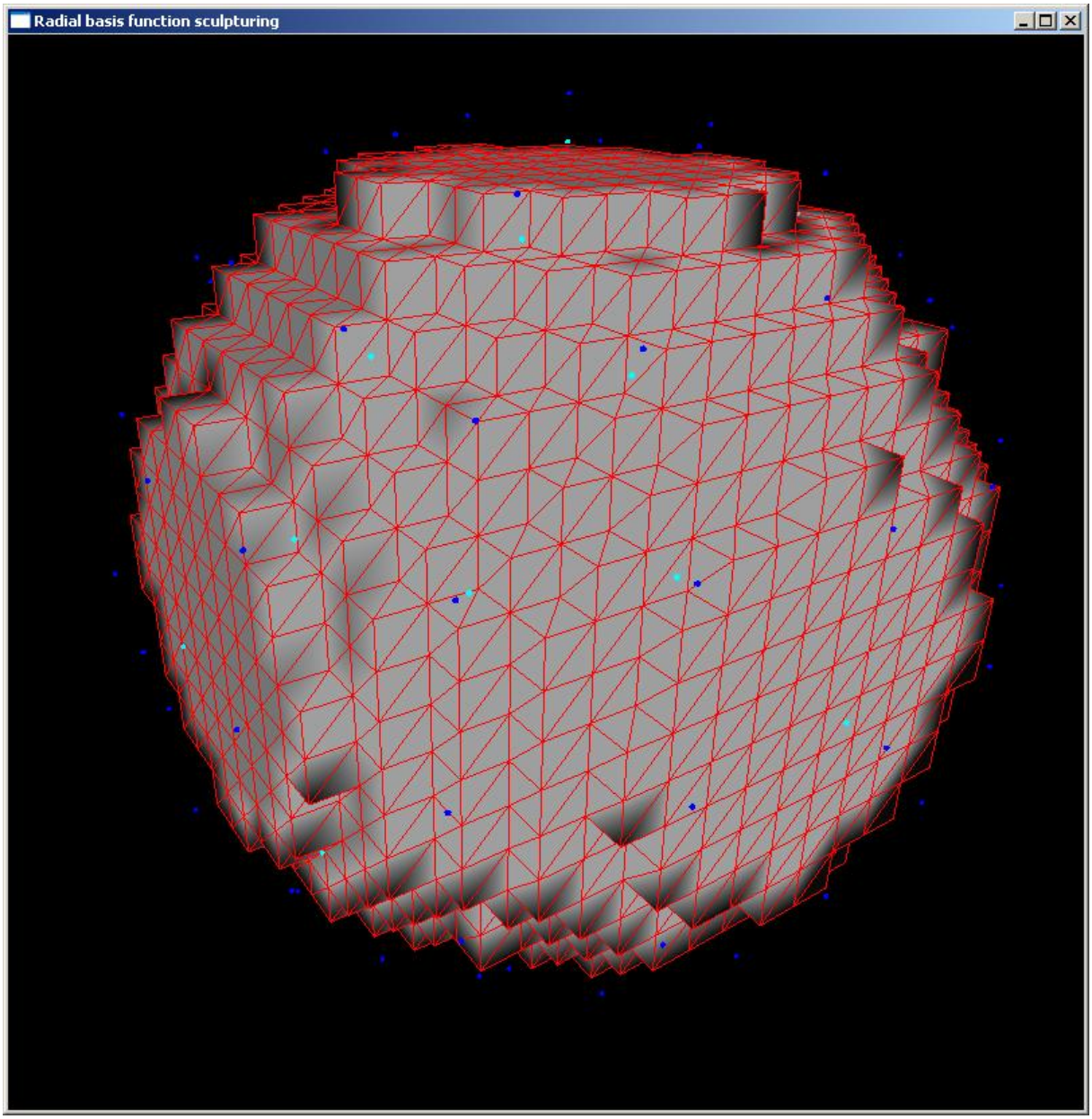
[Schaefer] Scott Schaefer et al., Dual Contouring of Hermite Data, *Rice University*

[Schaefer2] Scott Schaefer and Joe Warren, Dual Marching Cubes: Primal Contouring of Dual Grids, *Rice University*

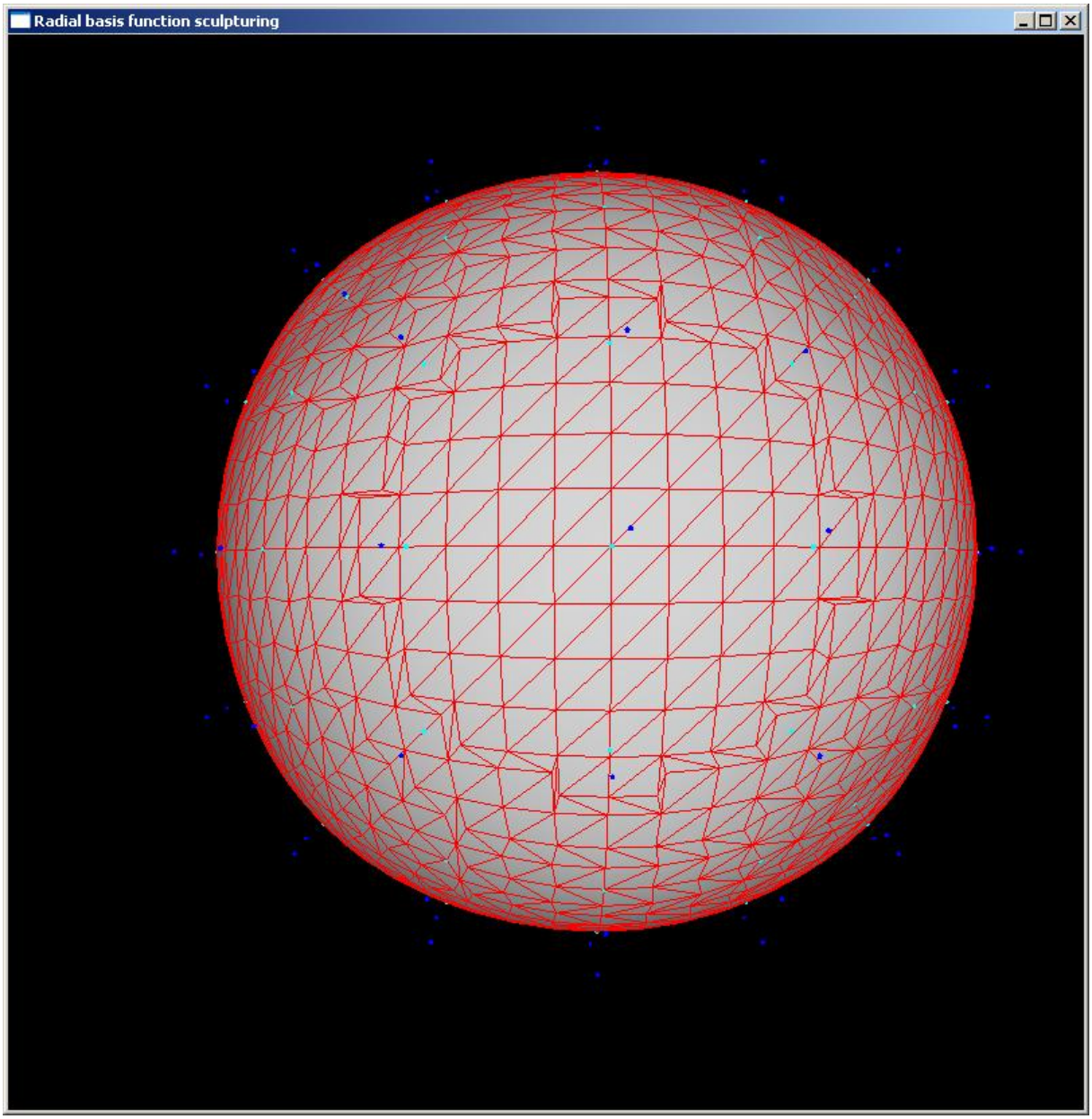
[Kobbelt] Leif Kobbelt, √3-subdivision, *Max-Planck Institute for Computer Sciences*

7.2 Figurlister

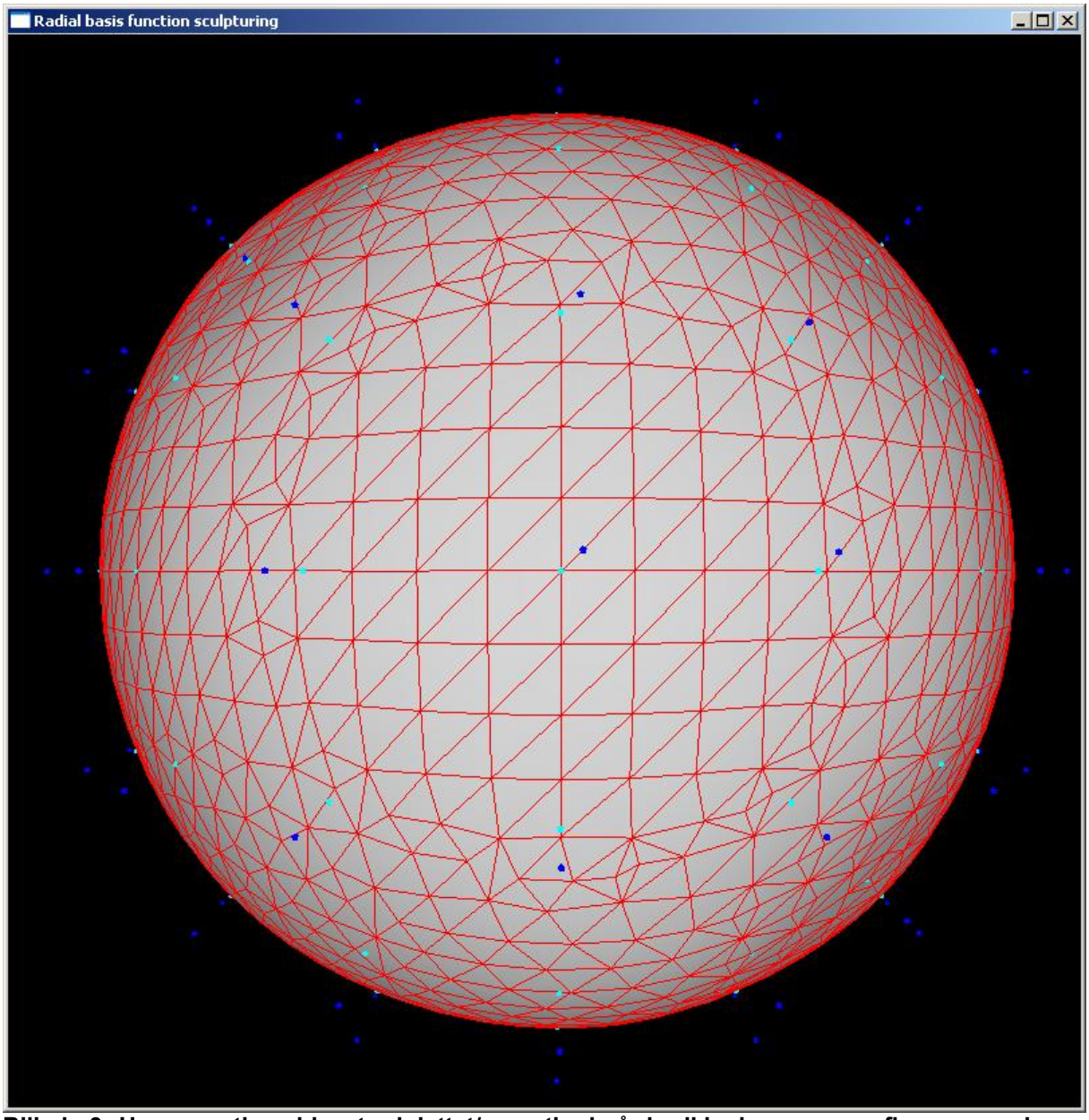
7.3 Billeder



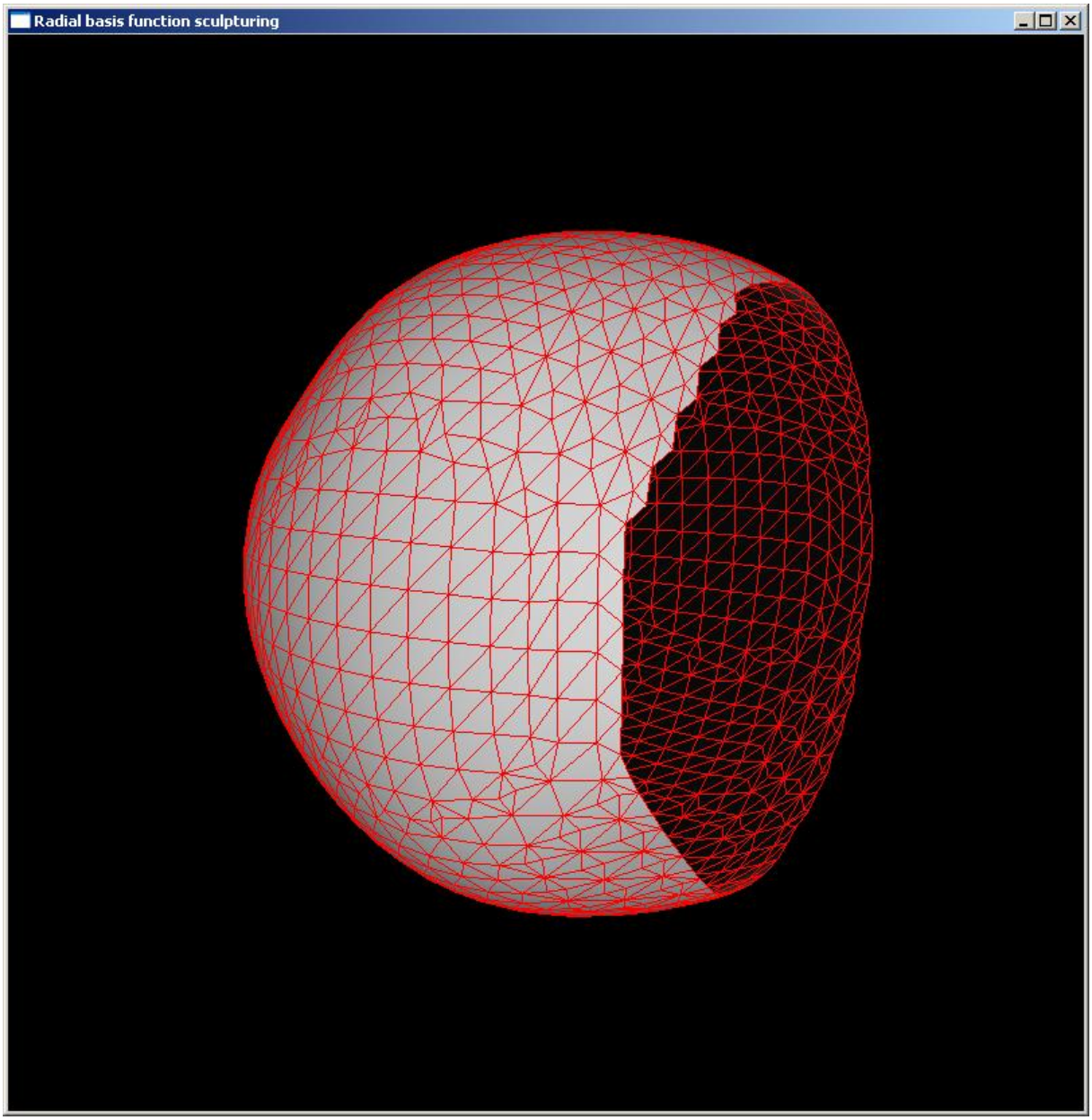
Billede 1: En approksimeret kugle, inden vertices er blevet fittet til overfladen



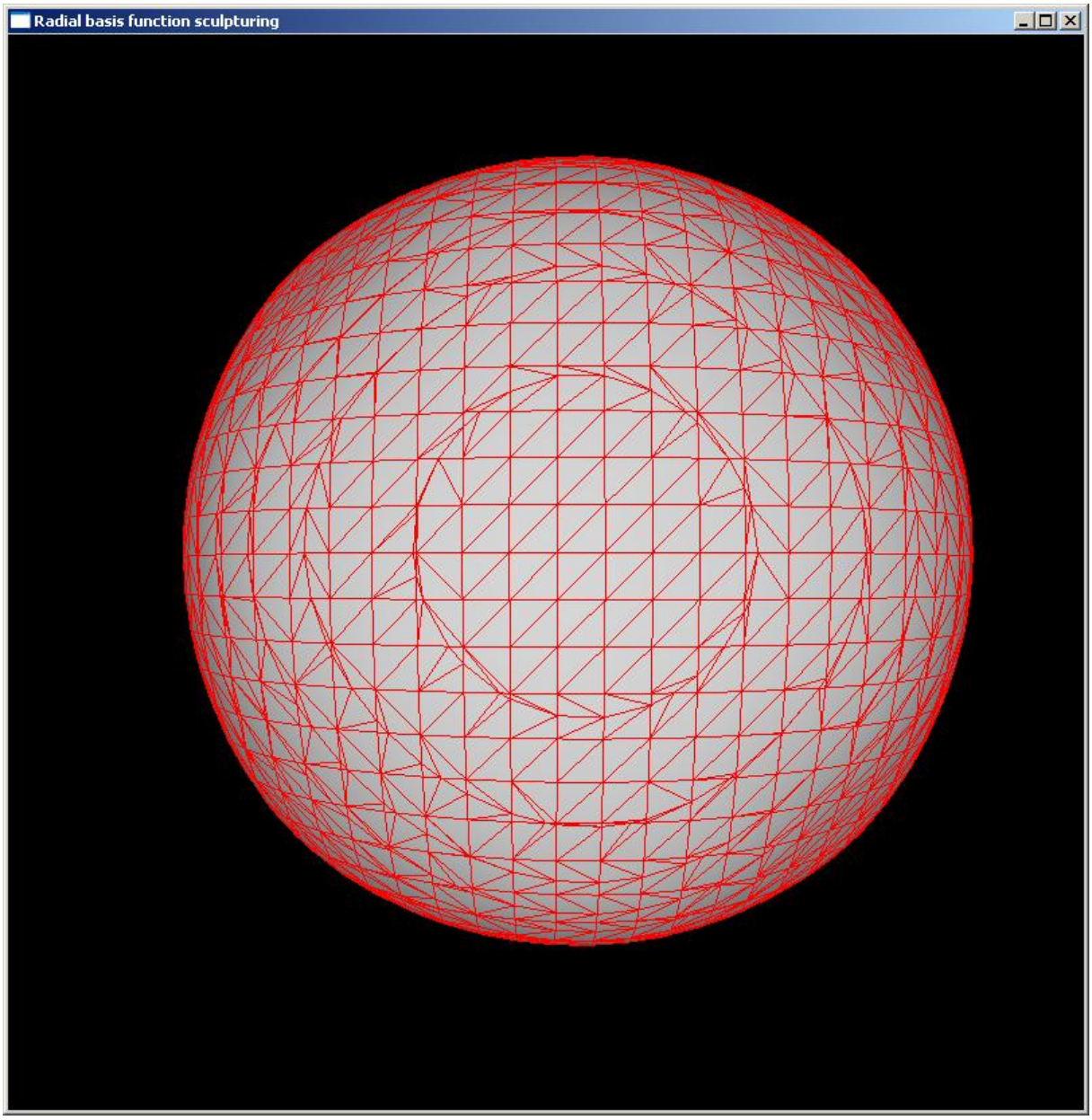
Billede 2: En kugle efter vertices er blevet fittet. Bemærk at dette har tendens til at danne såkaldte slivers



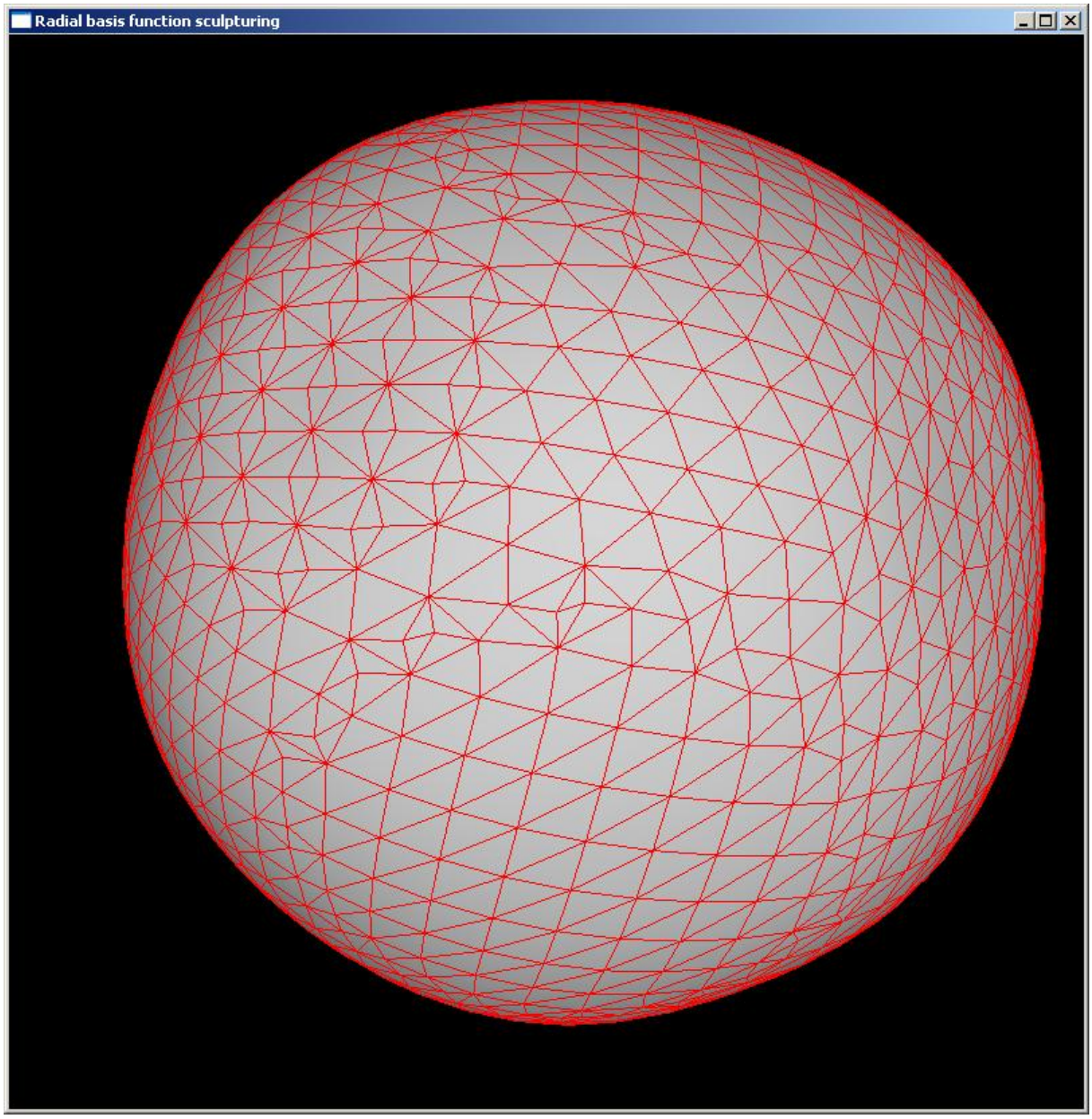
Billede 3: Her er vertices blevet udglattet/smoothed så der ikke længere er aflange og grimme trekanter



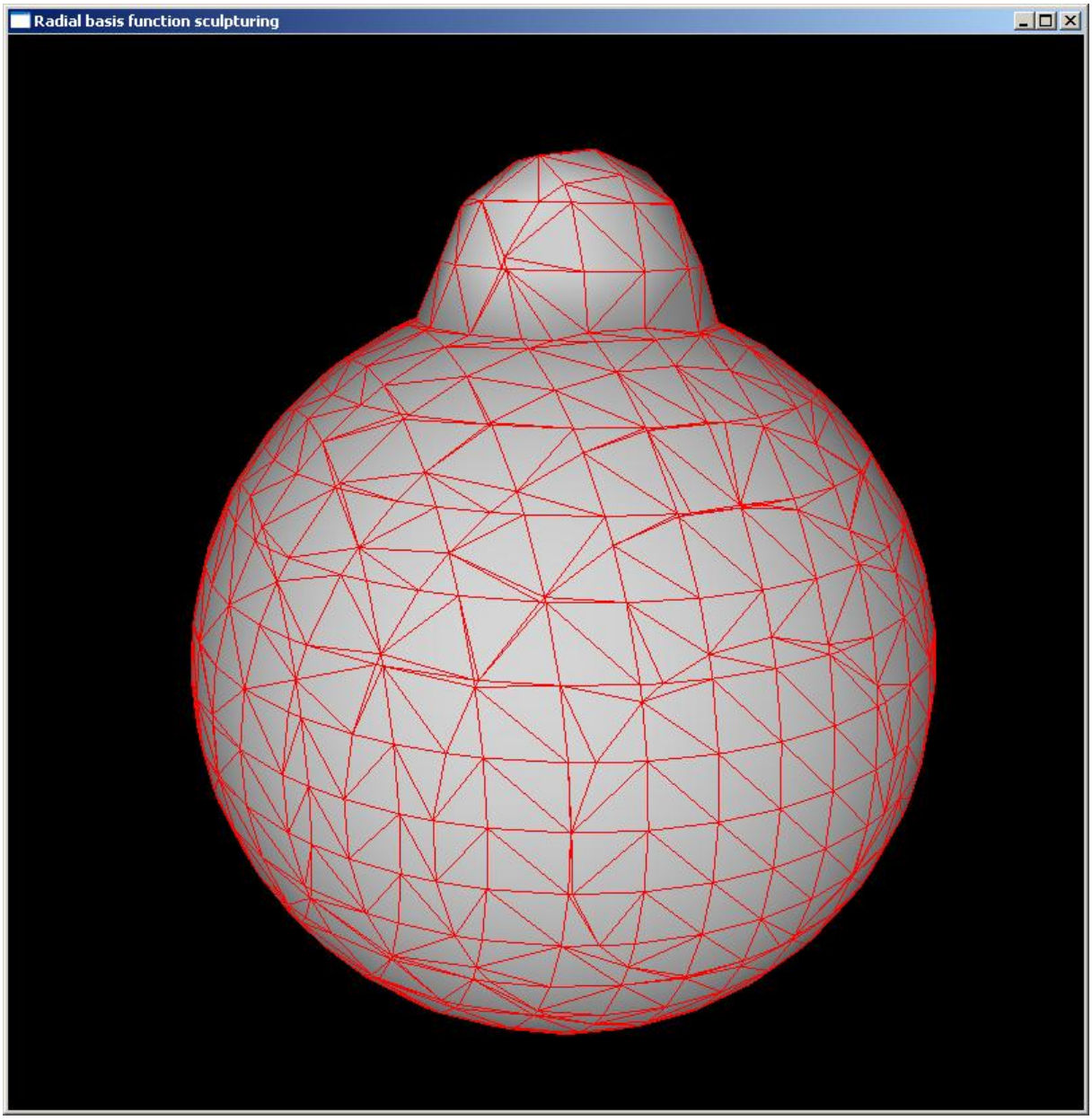
Billede 4: Polygoniseren tager ikke højde for volumen der strækker sig ud over de definerede grænser



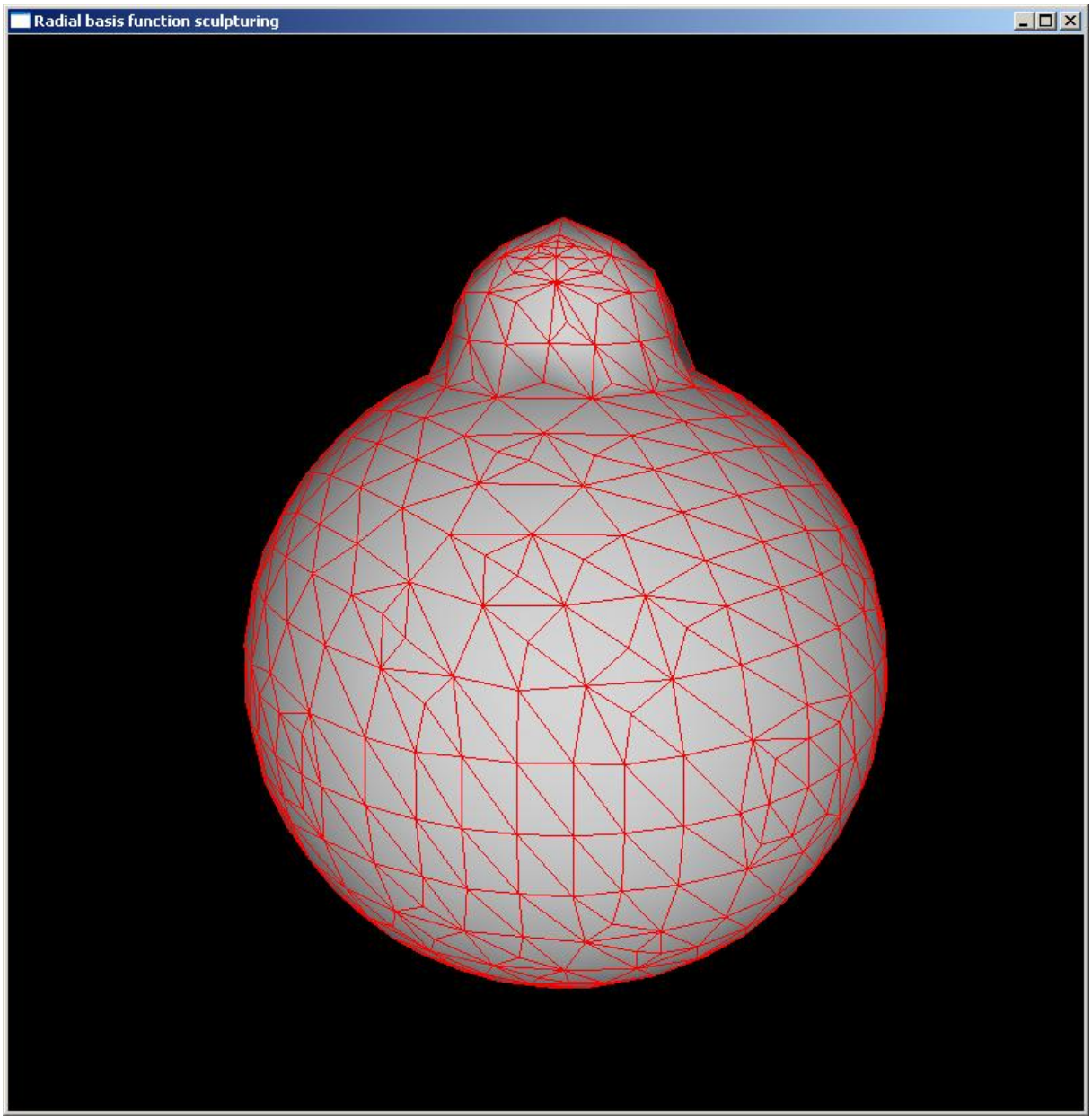
Billede 5: Weird3.rbf ved 20 steps med Bloomenthals polygoniser



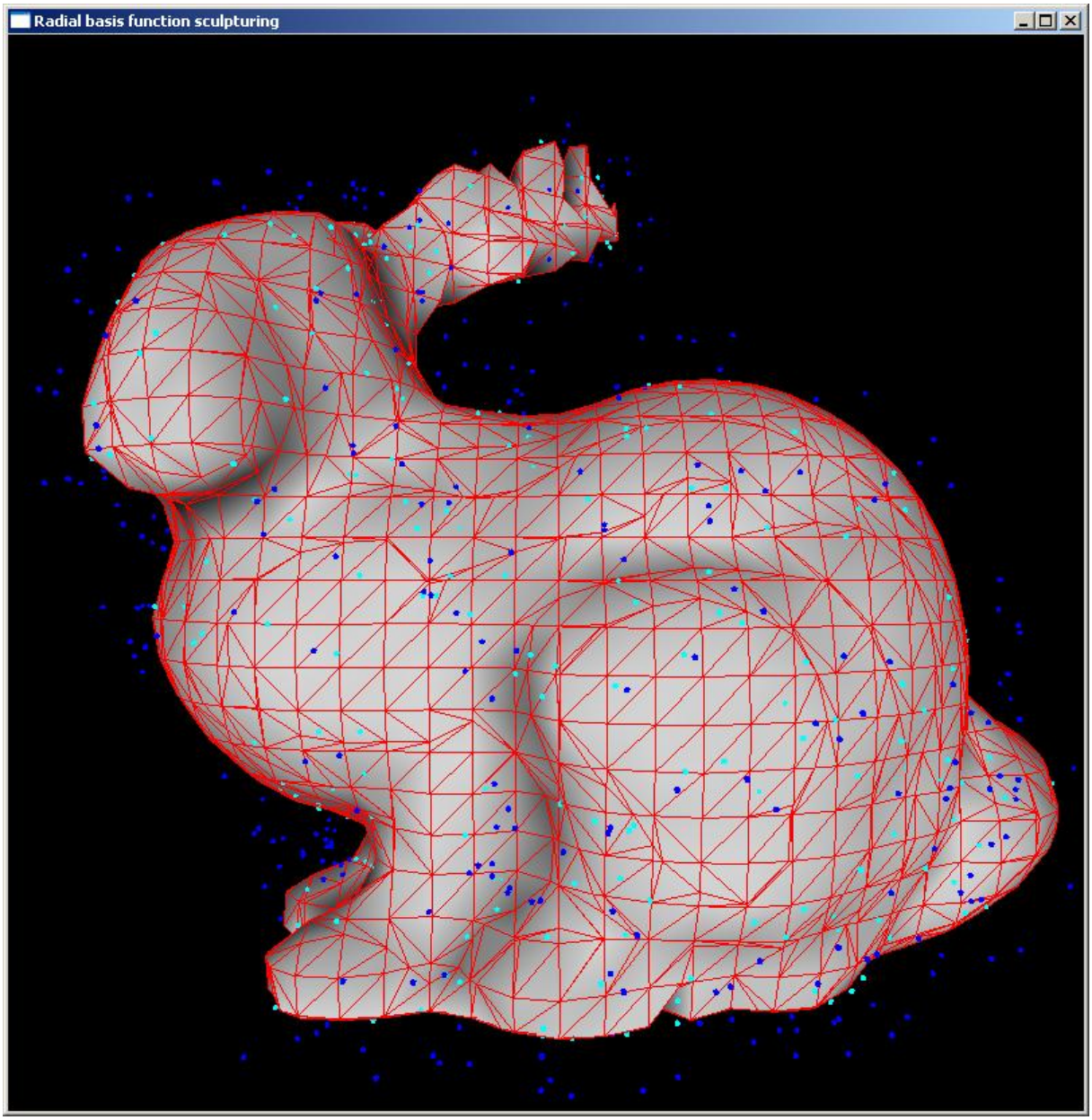
Billede 6: Weird3.rbf ved 20 steps med Jakobsens polygoniser



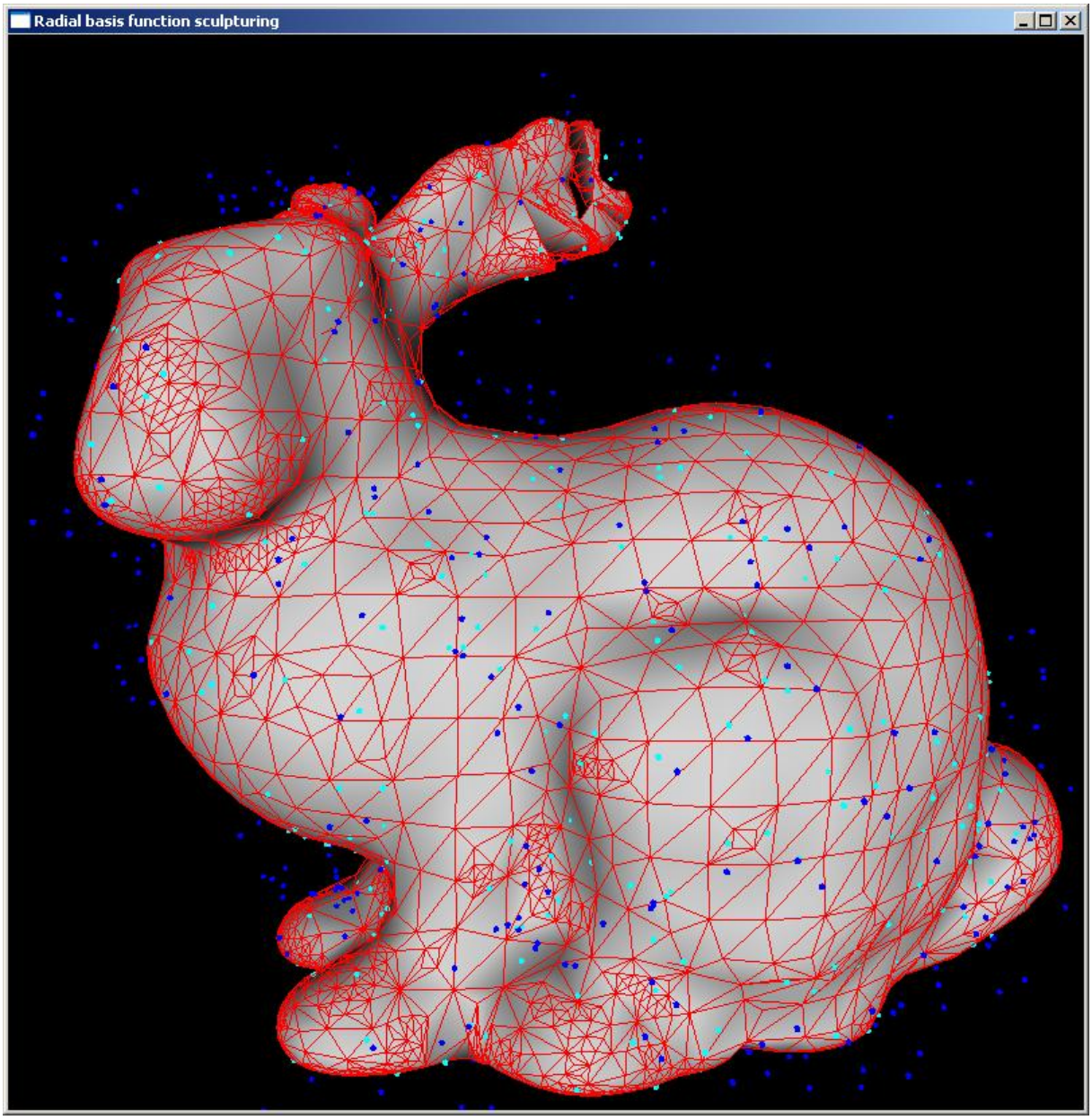
Billede 7: Funktionen af 2 kugler ved 20 steps med Bloomenthals polygoniser



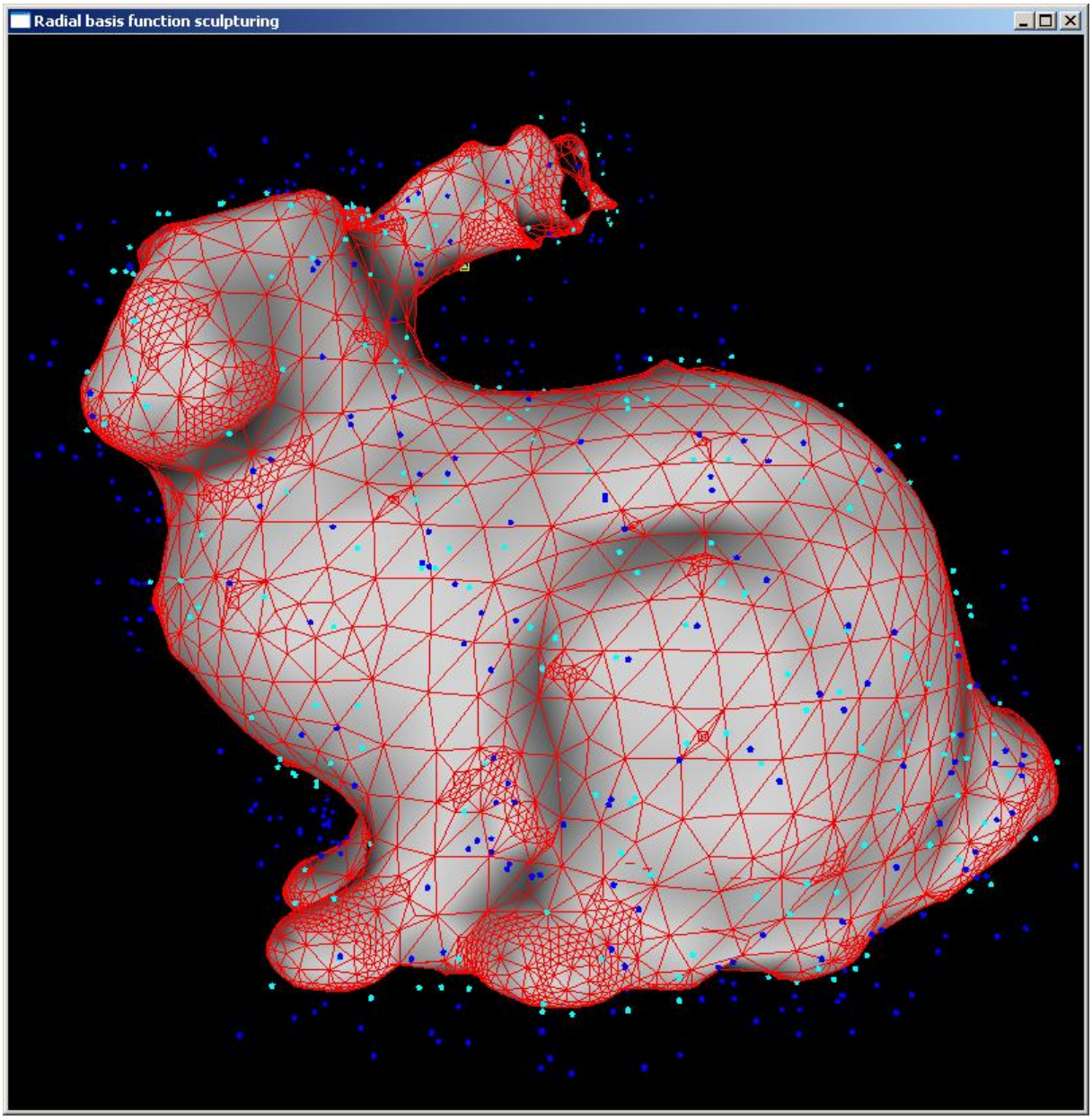
Billede 8: Funktionen af 2 kugler ved 20 steps med Jakobsens polygoniser



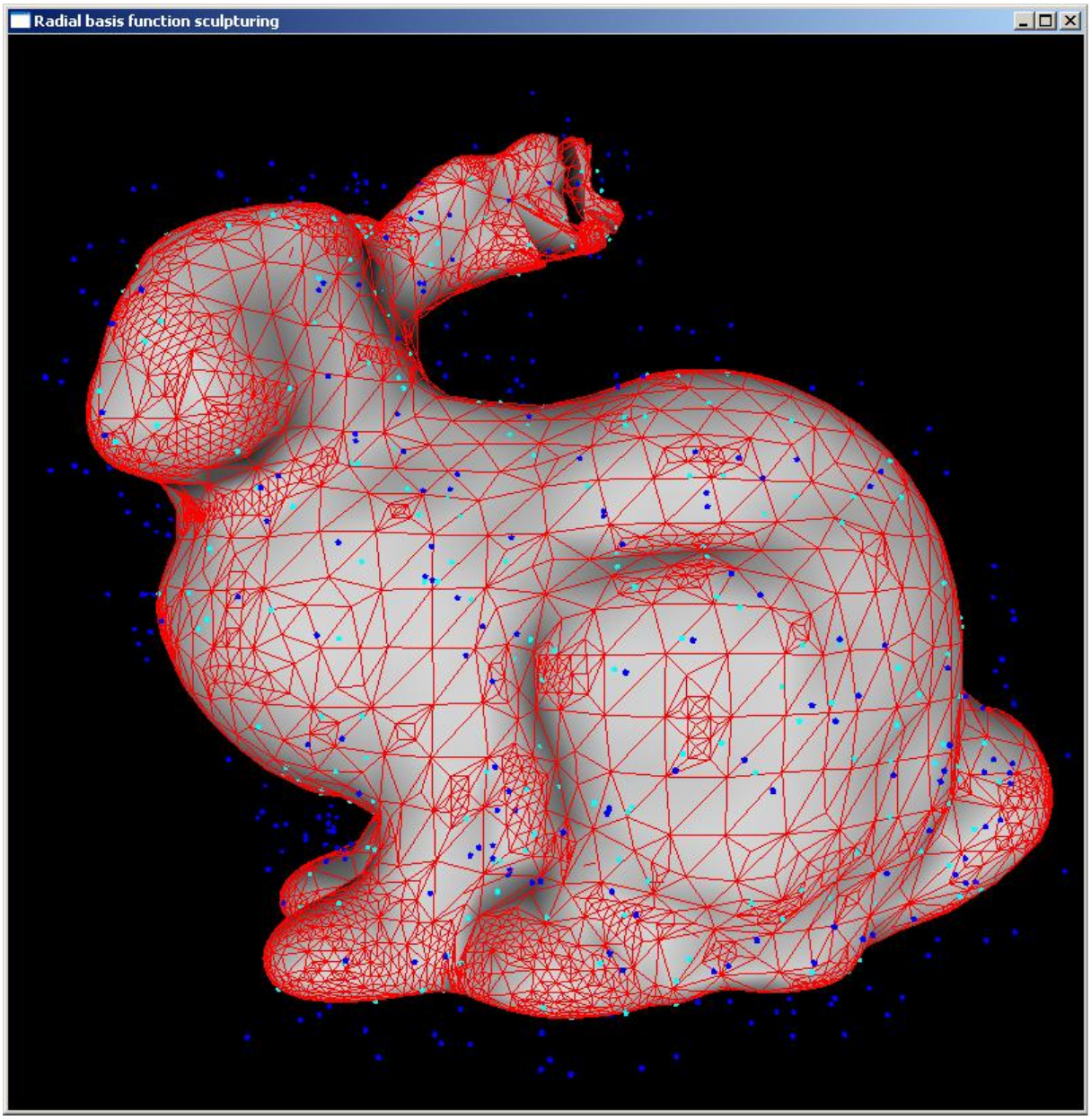
Billede 9: Bunny.rbf ved 20 steps med Bloomenthals polygoniser



Billede 10: Bunny.rbf ved 20 steps med Jakobsens polygoniser



Billede 11: Bunny.rbf ved 20 steps incl. med udglatning mellem hvert neddelingsniveau, med Jakobsens polygoniser



Billede 12: Bunny.rbf ved 20 steps, med andet neddelingskriterium, med Jakobsens polygoniser

7.4 Kildekode

```
/*********************************************************************
polygonizer.h
*********************************************************************/
#ifndef POLYGONIZER_H
#define POLYGONIZER_H

#include <vector>
#include <CGLA/Vec3f.h>
#include "ImplicitFunction.h"

/** TRIANGLE struct contains the indices of the vertices comprising
     the triangle */
struct TRIANGLE
{
   int v[3];       //Indexes of vertices
   int generation;    //Increased one time for each subdivision
   int neighbours[3];//Indexes of neighbouring triangles
};

struct edge_triangle
{
   int vertice; //second vertice index
   int triangle;//triangle index
};

/** Polygonizer is the class used to perform polygonization.*/
class Polygonizer
{
   std::vector<CGLA::Vec3f> gnormals;
   std::vector<CGLA::Vec3f> gvertices;
   std::vector<TRIANGLE> gtriangles;
   std::vector<std::vector<edge_triangle> > edge_table;

   ImplicitFunction* func;
   int bounds;

   std::vector<float> voxelgrid;
   std::vector<int> dualgrid;

   void build_triangles(int dir, int x, int y, int z);
   void fit_vertices();
   void smooth_vertices(int iteration);
   void subdivide();
   void build_edge_table();
   int find_neighbour(int index);
   void split_triangle(int triangle_index, CGLA::Vec3f triangle_center, float
criteria, std::vector<TRIANGLE> &temp_triangles);
   void swap_edges(int triangle1, int triangle2);

   public:

   /** Constructor of Polygonizer. The first argument is the ImplicitFunction
        that we wish to polygonize. The final arg. is the amount of steps we
traverse
        through R[0:1]
   */
     Polygonizer(ImplicitFunction* _func, int _bounds): func(_func),
bounds(_bounds) {}

   /** March erases the triangles gathered so far and builds a new
        polygonization.
```

```
   */
   void march();

   /** Return number of triangles generated after the polygonization.
        Call this function only when march has been called. */
   int no_triangles() const
   {
      return (int)gtriangles.size();
   }

   /** Return number of vertices generated after the polygonization.
        Call this function only when march has been called. */
   int no_vertices() const
   {
      return (int)gvertices.size();
   }

   /** Return number of normals generated after the polygonization.
        Of course the result of calling this function is the same as
        no_vertices.
        Call this function only when march has been called. */
   int no_normals() const
   {
      return (int)gnormals.size();
   }

   void set_info(int b)
   {
      bounds = b;
   }

   /// Return triangle with index i.
   TRIANGLE& get_triangle(int i)
   {
      return gtriangles[i];
   }

   /// Return vertex with index i.
   CGLA::Vec3f& get_vertex(int i)
   {
      return gvertices[i];
   }

   /// Return normal with index i.
   CGLA::Vec3f& get_normal(int i)
   {
      return gnormals[i];
   }
};

#endif
```

```
/**************************************************************************
polygonizer.cpp
**************************************************************************/
#include <stdlib.h>
#include <math.h>
#include <iostream>
#include <vector>
#include <list>
#include <sys/types.h>
#include "Jakobsen/polygonizer.h"

using namespace std;
using namespace CGLA;

///Number of iterations of vertice movement towards surface
#define MAX_FIT_ITERATION 15
#define MAX_SMOOTH_ITERATION 1

/**
 *Marches through the grid and checks at each gridpoint if its inside the volume.
 *If so, neighbours are checked if they are outside the volume, and if they are,
 *a quad(two triangles) are build on the dual grid in the direction of that
neighbour.
 */
void Polygonizer::march()
{
   voxelgrid.clear();
   dualgrid.clear();
   gvertices.clear();
   gnormals.clear();
   gtriangles.clear();
   edge_table.clear();
   for(int y = 0;y < bounds;y++)
   {
      for(int z = 0;z < bounds;z++)
      {
         for(int x = 0;x < bounds;x++)
         {
            float value = func->eval((float)x/(bounds-1),(float)y/(bounds-
1),(float)z/(bounds-1));
            voxelgrid.push_back(value);
         }
      }
   }
   int dualsize = (bounds+1)*(bounds+1)*(bounds+1);
   for(int i = 0;i < dualsize; i++)
      dualgrid.push_back(-1);

   for(int y = 0;y < bounds;y++)
   {
      for(int z = 0;z < bounds;z++)
      {
         for(int x = 0;x < bounds;x++)
         {
            int i = x+ bounds*z + bounds*bounds*y;
            ///if voxel is inside volumen
            if(voxelgrid[i] > 0)
            {
               ///if voxel is not on left border
               if(x!=0)
               {
                  ///if left neighbour is outside volume
                  if(voxelgrid[i-1] < 0)
                  {
                     ///build traingles to the left of x,y,z
```

```
      build_triangles(0,x,y,z);
   }
}
else
{
   ///maybe build border triangles to ensure closed surface
}
///if voxel is not on right border
if(x!=bounds-1)
{
   ///if right neighbour is outside volume
   if(voxelgrid[i+1] < 0)
   {
      ///build traingles to the right of x,y,z
      build_triangles(1,x,y,z);
   }
}
else
{
   ///maybe build border triangles to ensure closed surface
}
///if voxel is not on lower border
if(z!=0)
{
   ///if lower neighbour is outside volume
   if(voxelgrid[i-bounds] < 0)
   {
      ///build traingles down of x,y,z
      build_triangles(2,x,y,z);
   }
}
else
{
   ///maybe build border triangles to ensure closed surface
}
///if voxel is not on upper border
if(z!=bounds-1)
{
   ///if upper neighbour is outside volume
   if(voxelgrid[i+bounds] < 0)
   {
      ///build traingles up of x,y,z
      build_triangles(3,x,y,z);
   }
}
else
{
   ///maybe build border triangles to ensure closed surface
}
///if voxel is not on near border
if(y!=0)
{
   ///if near neighbour is outside volume
   if(voxelgrid[i-(bounds*bounds)] < 0)
   {
      ///build traingles to the near of x,y,z
      build_triangles(4,x,y,z);
   }
}
else
{
   ///maybe build border triangles to ensure closed surface
}
///if voxel is not on far border
if(y!=bounds-1)
```

```
            {
               ///if far neighbour is outside volume
               if(voxelgrid[i+(bounds*bounds)] < 0)
               {
                  ///build traingles to the far of x,y,z
                  build_triangles(5,x,y,z);
               }
            }
            else
            {
               ///maybe build border triangles to ensure closed surface
            }
         }
      }
   }
   build_edge_table();
   fit_vertices();
   subdivide();
}

/**
 *Builds the initial triangles at the vertices in the dualgrid.
 *Adds only the vertices and normals used in the triangles to their
 *respective vectors. If a vertice already have been added, its reused
 *instead of creating double vertices.
 */
void Polygonizer::build_triangles(int dir, int x, int y, int z)
{
   float halfstep = (float)1/(2*(bounds-1));
   float left = (float)x/(bounds-1) - halfstep;
   float right = (float)x/(bounds-1) + halfstep;
   float near = (float)y/(bounds-1) - halfstep;
   float far = (float)y/(bounds-1) + halfstep;
   float down = (float)z/(bounds-1) - halfstep;
   float up = (float)z/(bounds-1) + halfstep;
   Vec3f vertice[4];
   int dual_index[4];
   TRIANGLE t1,t2;

   switch(dir)
   {
   ///Build to the left
   case 0:
      {
         ///left bottom far corner
         dual_index[0] = x+ (bounds+1)*z + (bounds+1)*(bounds+1)*(y+1);
         vertice[0] = Vec3f(left,far,down);
         ///left bottom near corner
         dual_index[1] = x+ (bounds+1)*z + (bounds+1)*(bounds+1)*y;
         vertice[1] = Vec3f(left,near,down);
         ///left top near corner
         dual_index[2] = x+ (bounds+1)*(z+1) + (bounds+1)*(bounds+1)*y;
         vertice[2] = Vec3f(left,near,up);
         ///left top far corner
         dual_index[3] = x+ (bounds+1)*(z+1) + (bounds+1)*(bounds+1)*(y+1);
         vertice[3] = Vec3f(left,far,up);
             for(int i=0;i<4;i++)
         {
            if(dualgrid[dual_index[i]] == -1)
            {
               dualgrid[dual_index[i]] = (int)gvertices.size();
               Vec3f n(1,0,0);
               gnormals.push_back(n);
               gvertices.push_back(vertice[i]);
```

```
        }
      }
      t1.v[0] = dualgrid[dual_index[0]];
      t1.v[1] = dualgrid[dual_index[1]];
      t1.v[2] = dualgrid[dual_index[2]];
      t1.generation = 0;
          gtriangles.push_back(t1);
      t2.v[0] = dualgrid[dual_index[0]];
      t2.v[1] = dualgrid[dual_index[2]];
      t2.v[2] = dualgrid[dual_index[3]];
      t2.generation = 0;
          gtriangles.push_back(t2);
   }
break;
///Build to the right
case 1:
   {
      ///right bottom near corner
      dual_index[0] = (x+1)+ (bounds+1)*z + (bounds+1)*(bounds+1)*y;
      vertice[0] = Vec3f(right,near,down);
      ///right bottom far corner
      dual_index[1] = (x+1)+ (bounds+1)*z + (bounds+1)*(bounds+1)*(y+1);
      vertice[1] = Vec3f(right,far,down);
      ///right top far corner
      dual_index[2] = (x+1)+ (bounds+1)*(z+1) + (bounds+1)*(bounds+1)*(y+1);
      vertice[2] = Vec3f(right,far,up);
      ///right top near corner
      dual_index[3] = (x+1)+ (bounds+1)*(z+1) + (bounds+1)*(bounds+1)*y;
      vertice[3] = Vec3f(right,near,up);
          for(int i=0;i<4;i++)
      {
         if(dualgrid[dual_index[i]] == -1)
         {
            dualgrid[dual_index[i]] = (int)gvertices.size();
            Vec3f n(-1,0,0);
            gnormals.push_back(n);
            gvertices.push_back(vertice[i]);
         }
      }
      t1.v[0] = dualgrid[dual_index[0]];
      t1.v[1] = dualgrid[dual_index[1]];
      t1.v[2] = dualgrid[dual_index[2]];
      t1.generation = 0;
          gtriangles.push_back(t1);
      t2.v[0] = dualgrid[dual_index[0]];
      t2.v[1] = dualgrid[dual_index[2]];
      t2.v[2] = dualgrid[dual_index[3]];
      t2.generation = 0;
          gtriangles.push_back(t2);
   }
break;
///Build to the down
case 2:
   {
      ///left bottom near corner
      dual_index[0] = x+ (bounds+1)*z + (bounds+1)*(bounds+1)*y;
      vertice[0] = Vec3f(left,near,down);
      ///left botton far corner
      dual_index[1] = x+ (bounds+1)*z + (bounds+1)*(bounds+1)*(y+1);
      vertice[1] = Vec3f(left,far,down);
      ///right bottom far corner
      dual_index[2] = (x+1)+ (bounds+1)*z + (bounds+1)*(bounds+1)*(y+1);
      vertice[2] = Vec3f(right,far,down);
      ///right bottom near corner
      dual_index[3] = (x+1)+ (bounds+1)*z + (bounds+1)*(bounds+1)*y;
```

```
      vertice[3] = Vec3f(right,near,down);
          for(int i=0;i<4;i++)
      {
         if(dualgrid[dual_index[i]] == -1)
         {
            dualgrid[dual_index[i]] = (int)gvertices.size();
            Vec3f n(0,0,1);
            gnormals.push_back(n);
            gvertices.push_back(vertice[i]);
         }
      }
      t1.v[0] = dualgrid[dual_index[0]];
      t1.v[1] = dualgrid[dual_index[1]];
      t1.v[2] = dualgrid[dual_index[2]];
      t1.generation = 0;
          gtriangles.push_back(t1);
      t2.v[0] = dualgrid[dual_index[0]];
      t2.v[1] = dualgrid[dual_index[2]];
      t2.v[2] = dualgrid[dual_index[3]];
      t2.generation = 0;
          gtriangles.push_back(t2);
   }
break;
///Build to the up
case 3:
   {
      ///left top near corner
      dual_index[0] = x+ (bounds+1)*(z+1) + (bounds+1)*(bounds+1)*y;
      vertice[0] = Vec3f(left,near,up);
      ///right top near corner
      dual_index[1] = (x+1)+ (bounds+1)*(z+1) + (bounds+1)*(bounds+1)*y;
      vertice[1] = Vec3f(right,near,up);
      ///right top far corner
      dual_index[2] = (x+1)+ (bounds+1)*(z+1) + (bounds+1)*(bounds+1)*(y+1);
      vertice[2] = Vec3f(right,far,up);
      ///left top far corner
      dual_index[3] = x+ (bounds+1)*(z+1) + (bounds+1)*(bounds+1)*(y+1);
      vertice[3] = Vec3f(left,far,up);
          for(int i=0;i<4;i++)
      {
         if(dualgrid[dual_index[i]] == -1)
         {
            dualgrid[dual_index[i]] = (int)gvertices.size();
            Vec3f n(0,0,-1);
            gnormals.push_back(n);
            gvertices.push_back(vertice[i]);
      }
      }
      t1.v[0] = dualgrid[dual_index[0]];
      t1.v[1] = dualgrid[dual_index[1]];
      t1.v[2] = dualgrid[dual_index[2]];
      t1.generation = 0;
          gtriangles.push_back(t1);
      t2.v[0] = dualgrid[dual_index[0]];
      t2.v[1] = dualgrid[dual_index[2]];
      t2.v[2] = dualgrid[dual_index[3]];
      t2.generation = 0;
          gtriangles.push_back(t2);
   }
break;
///Build to the near
case 4:
   {
      ///left bottom near corner
      dual_index[0] = x+ (bounds+1)*z + (bounds+1)*(bounds+1)*y;
```

```
      vertice[0] = Vec3f(left,near,down);
      ///right bottom near corner
      dual_index[1] = (x+1)+ (bounds+1)*z + (bounds+1)*(bounds+1)*y;
      vertice[1] = Vec3f(right,near,down);
      ///right top near corner
      dual_index[2] = (x+1)+ (bounds+1)*(z+1) + (bounds+1)*(bounds+1)*y;
      vertice[2] = Vec3f(right,near,up);
      ///left top near corner
      dual_index[3] = x+ (bounds+1)*(z+1) + (bounds+1)*(bounds+1)*y;
      vertice[3] = Vec3f(left,near,up);
          for(int i=0;i<4;i++)
      {
         if(dualgrid[dual_index[i]] == -1)
         {
            dualgrid[dual_index[i]] = (int)gvertices.size();
            Vec3f n(0,1,0);
            gnormals.push_back(n);
            gvertices.push_back(vertice[i]);
         }
      }
      t1.v[0] = dualgrid[dual_index[0]];
      t1.v[1] = dualgrid[dual_index[1]];
      t1.v[2] = dualgrid[dual_index[2]];
      t1.generation = 0;
          gtriangles.push_back(t1);
      t2.v[0] = dualgrid[dual_index[0]];
      t2.v[1] = dualgrid[dual_index[2]];
      t2.v[2] = dualgrid[dual_index[3]];
      t2.generation = 0;
          gtriangles.push_back(t2);
   }
break;
///Build to the far
case 5:
   {
      ///right bottom far corner
      dual_index[0] = (x+1)+ (bounds+1)*z + (bounds+1)*(bounds+1)*(y+1);
      vertice[0] = Vec3f(right,far,down);
      ///left bottom far corner
      dual_index[1] = x+ (bounds+1)*z + (bounds+1)*(bounds+1)*(y+1);
      vertice[1] = Vec3f(left,far,down);
      ///left top far corner
      dual_index[2] = x+ (bounds+1)*(z+1) + (bounds+1)*(bounds+1)*(y+1);
      vertice[2] = Vec3f(left,far,up);
      ///right top far corner
      dual_index[3] = (x+1)+ (bounds+1)*(z+1) + (bounds+1)*(bounds+1)*(y+1);
      vertice[3] = Vec3f(right,far,up);
          for(int i=0;i<4;i++)
      {
         if(dualgrid[dual_index[i]] == -1)
         {
            dualgrid[dual_index[i]] = (int)gvertices.size();
            Vec3f n(0,-1,0);
            gnormals.push_back(n);
            gvertices.push_back(vertice[i]);
         }
      }
         t1.v[0] = dualgrid[dual_index[0]];
      t1.v[1] = dualgrid[dual_index[1]];
      t1.v[2] = dualgrid[dual_index[2]];
      t1.generation = 0;
          gtriangles.push_back(t1);
      t2.v[0] = dualgrid[dual_index[0]];
      t2.v[1] = dualgrid[dual_index[2]];
      t2.v[2] = dualgrid[dual_index[3]];
```

```
        t2.generation = 0;
            gtriangles.push_back(t2);
      }
   break;
   }
}

/**
 *Projects the vertices closer to the surface by moving them along their gradient
vectors
 *and smoothes them at the same time by moving them closer to the center of their
neighbours.
 *The fitting stops either if the points are at a given distance of the surface or
if the number of iterations
 *surpasses the threshold.
 */
void Polygonizer::fit_vertices()
{
   int smooth_iteration = 0;
   do
   {
      smooth_vertices(smooth_iteration);
      for(int i=0;i<(int)gvertices.size();i++)
      {
         int count = 0;
         float f;
         Vec3f p = gvertices[i];
         Vec3f grad;
         do
         {
            ///f(p)
            f = func->eval(p[0],p[1],p[2]);
            float delta = (float)1/(bounds*10);
            float dx = func->eval(p[0]+delta,p[1],p[2]) - func->eval(p[0]-
delta,p[1],p[2]);
            float dy = func->eval(p[0],p[1]+delta,p[2]) - func->eval(p[0],p[1]-
delta,p[2]);
            float dz = func->eval(p[0],p[1],p[2]+delta) - func->eval(p[0],p[1],p[2]-
delta);
            ///Grad f(p)
            grad = Vec3f(dx/(2*delta),dy/(2*delta),dz/(2*delta));
            float length = grad.length();
            float step = f/length;
            grad.normalize();
            Vec3f q = grad * step;
            p -= q;
            count++;
         }while(abs(f) > (1./(bounds*count)) && count < MAX_FIT_ITERATION);
         gvertices[i] = p;
         gnormals[i] = grad;
      }
      smooth_iteration++;
   }while(smooth_iteration < MAX_SMOOTH_ITERATION);
}

/**
 *Smoothes a vertex by moving i a bit closer to th center of its neighbours
 *moving it less for each iteration in order not to disturb the fitting process
 */
void Polygonizer::smooth_vertices(int iteration)
{
   vector<CGLA::Vec3f> temp_vertices;
   for(int i=0;i<(int)gvertices.size();i++)
   {
      Vec3f center(0,0,0);
```

```
      bool edge = true;
      int list_size = (int)edge_table[i].size();
      ///calculate the center of neighbours
      for(int j=0;j<list_size;j++)
      {
         int temp = edge_table[i][j].vertice;
         for(int h =0;h < (int)edge_table[temp].size();h++)
         {
            if(edge_table[temp][h].vertice == i)
               edge = false;
         }
         center += gvertices[temp];
      }
      center /= list_size;
      ///Calculate the new center
      Vec3f tmp = ((1/(iteration+1.7)) * (center - gvertices[i]))+ gvertices[i];
      if(edge == true)
      {
         temp_vertices.push_back(gvertices[i]);
      }
      else
      {
         temp_vertices.push_back(tmp);
      }
   }
   gvertices.swap(temp_vertices);
   temp_vertices.clear();
}

/**
 *Builds an edge table.
 *The index in the edge table points to the index of the first vertice in an edge.
 *At this index there is a vector with the size of the amount of edges that this
given
 *vertex begins. Each entity in that vector holds the information of the other
vertice in
 *the edge and the triangle index that edge belongs to.
 */
void Polygonizer::build_edge_table()
{
   edge_table.clear();

   for(int h =0;h<(int)gvertices.size();h++)
   {
      vector<edge_triangle> temp;
      edge_table.push_back(temp);
   }
   for(int i =0;i<(int)gtriangles.size();i++)
   {
      for(int j=0;j<3;j++)
      {
         int index = gtriangles[i].v[j];
         edge_triangle t;
         t.triangle = i;
         if(j!=2)
            t.vertice = gtriangles[i].v[j+1];
         else
            t.vertice = gtriangles[i].v[0];
         edge_table[index].push_back(t);
      }
   }
}

int Polygonizer::find_neighbour(int index)
{
```

```
   int result;
   int end_vert = gtriangles[index].v[1];
   int size = (int)edge_table[end_vert].size();
   for(int j=0;j<size;j++)
   {
      if(edge_table[end_vert][j].vertice == gtriangles[index].v[0])
      {
         return edge_table[end_vert][j].triangle;
      }
      else
         result = -1;
   }
   return -1;
}

void Polygonizer::split_triangle(int index, Vec3f center, float criteria,
vector<TRIANGLE> &temp_triangles)
{
   TRIANGLE t = gtriangles[index];
   if(t.generation % 2 == 0)
   {
      Vec3f grad;
      float f;
      float count = 0;
      do
      {
         f = func->eval(center[0],center[1],center[2]);
         float delta = (float)1/bounds;
         float dx = func->eval(center[0]+delta,center[1],center[2]) - func-
>eval(center[0]-delta,center[1],center[2]);
         float dy = func->eval(center[0],center[1]+delta,center[2]) - func-
>eval(center[0],center[1]-delta,center[2]);
         float dz = func->eval(center[0],center[1],center[2]+delta) - func-
>eval(center[0],center[1],center[2]-delta);
         ///Grad f(p)
         grad = Vec3f(dx/(2*delta),dy/(2*delta),dz/(2*delta));
         float length = grad.length();
         float step = f/length;
         grad.normalize();
         Vec3f q = grad * step;
         center -= q;
         count++;
      }while(abs(f) > criteria && count < MAX_FIT_ITERATION);
      int c = (int)gvertices.size();
      gvertices.push_back(center);
      gnormals.push_back(grad);
      TRIANGLE tmp[3];
      tmp[0].v[0] = t.v[0];
      tmp[0].v[1] = t.v[1];
      tmp[1].v[0] = t.v[1];
      tmp[1].v[1] = t.v[2];
      tmp[2].v[0] = t.v[2];
      tmp[2].v[1] = t.v[0];
      for(int g=0;g<3;g++)
      {
         tmp[g].v[2] = c;
         tmp[g].generation = t.generation +1;
         temp_triangles.push_back(tmp[g]);
      }
   }
   else
   {
      int neighbour_index = find_neighbour(index);
      if(neighbour_index != -1)
      {
```

```
        if(gtriangles[index].generation == gtriangles[neighbour_index].generation)
           swap_edges(index, neighbour_index);
/*      else if(gtriangles[index].generation ==
gtriangles[neighbour_index].generation+1)
        {
           put in the code for splitting down the neighbour
           and then swap edges
        }*/
     }
   }
}

void Polygonizer::swap_edges(int index1,int index2)
{
   TRIANGLE t1,t2;
   t1 = gtriangles[index1];
   t2 = gtriangles[index2];
   gtriangles[index1].v[0] = t1.v[2];
   gtriangles[index1].v[1] = t1.v[0];
   gtriangles[index1].v[2] = t2.v[2];
   gtriangles[index1].generation++;
   gtriangles[index2].v[0] = t2.v[2];
   gtriangles[index2].v[1] = t1.v[1];
   gtriangles[index2].v[2] = t1.v[2];
   gtriangles[index2].generation++;
}

void Polygonizer::subdivide()
{
   vector<TRIANGLE> temp_triangles;
   for(int g=0;g <6 ;g++)
   {
      temp_triangles.clear();
      for(int i=0;i<(int)gtriangles.size();i++)
      {
         TRIANGLE t = gtriangles[i];
         if(t.generation == g)
         {
            Vec3f center = (gvertices[t.v[0]] + gvertices[t.v[1]] +
gvertices[t.v[2]])/3;
            float f = func->eval(center[0],center[1],center[2]);
            float delta = (float)1/bounds;
            float dx = func->eval(center[0]+delta,center[1],center[2]) - func-
>eval(center[0]-delta,center[1],center[2]);
            float dy = func->eval(center[0],center[1]+delta,center[2]) - func-
>eval(center[0],center[1]-delta,center[2]);
            float dz = func->eval(center[0],center[1],center[2]+delta) - func-
>eval(center[0],center[1],center[2]-delta);
            ///Grad f(p)
            Vec3f grad = Vec3f(dx/(2*delta),dy/(2*delta),dz/(2*delta));
            float length = grad.length();
            float criteria = 0.075/(bounds*(g+1));
            if(abs(f)/length > criteria)
            {
               split_triangle(i, center, criteria, temp_triangles);
            }
            else
            {
               temp_triangles.push_back(gtriangles[i]);
            }
         }
         else
         {
            temp_triangles.push_back(gtriangles[i]);
         }
```

```
      }
      if(g%2 == 0)
      {
         gtriangles.swap(temp_triangles);
      }
      build_edge_table();
   }
}
```

```
/**************************************************************************
Funktionen der er brugt til test 2 og test 3, som beregner den gennemsnitlige
minimums vinkel
**************************************************************************/
void calc_min_angle()
{
   float avg_angle = 0;
   for(int j=0;j<pols[0]->no_triangles(); ++j)
   {
      TRIANGLE t = pols[0]->get_triangle(j);
      Vec3f verts[3];
      for(int h=0;h<3;h++)
         verts[h] = pols[0]->get_vertex(t.v[h]);
      Vec3f arm1 = verts[1]-verts[0];
      Vec3f arm2 = verts[2]-verts[0];
      Vec3f arm3 = verts[1]-verts[2];
      Vec3f arm4 = verts[0]-verts[2];
      float length1 = arm1.length();
      float length2 = arm2.length();
      float length3 = arm3.length();
      float length4 = arm4.length();
      float temp1 = dot(arm1,arm2)/(length1*length2);
      float temp2 = dot(arm3,arm4)/(length3*length4);
         float angle1 = acos(temp1)/(2*M_PI)*360;
         float angle2 = acos(temp2)/(2*M_PI)*360;
      float angle3 = 180 - angle1 - angle2;
      float min_angle = min(angle1,min(angle2,angle3));
      avg_angle += min_angle;
   }
   avg_angle /= pols[0]->no_triangles();
   cout << avg_angle << endl;
}

/**************************************************************************
Funktionen der er brugt til test 4, som beregner den gennemsnitlige vurdering af
trekanternes centrum
**************************************************************************/
void calc_avg_f()
{
   float avg_f = 0;
   for(int j=0;j<pols[0]->no_triangles(); ++j)
   {
      TRIANGLE t = pols[0]->get_triangle(j);
      Vec3f verts[3];
      for(int h=0;h<3;h++)
         verts[h] = pols[0]->get_vertex(t.v[h]);
      Vec3f vertex = (verts[0] + verts[1] + verts[2]) /3;
      float delta = 1./pol_steps;
      float f = objects[0]->eval(vertex[0],vertex[1],vertex[2]);
      float dx = objects[0]->eval(vertex[0]+delta,vertex[1],vertex[2]) -
objects[0]->eval(vertex[0]-delta,vertex[1],vertex[2]);
      float dy = objects[0]->eval(vertex[0],vertex[1]+delta,vertex[2]) -
objects[0]->eval(vertex[0],vertex[1]-delta,vertex[2]);
      float dz = objects[0]->eval(vertex[0],vertex[1],vertex[2]+delta) -
objects[0]->eval(vertex[0],vertex[1],vertex[2]-delta);
      Vec3f grad = Vec3f(dx/(2*delta),dy/(2*delta),dz/(2*delta));
      float l = grad.length();
      float temp = abs(f)/l;
      avg_f += abs(temp);
   }
   avg_f /= pols[0]->no_triangles();
   cout << avg_f << endl;
}
```

7.5 Ordforklaringer

[1] En samling af sammenhængende polygoner/trekanter
[2] Et program som kan generere en samling af polygoner(trekanter), der sat sammen danner overfladen af et volumen.
[3] Et API baseret på C++, specifikt til computergrafik
[4] En sliver er en aflang, og geometrisk grim trekant
[5] Flertal - Punkter på overfladen af volumenet; Røder til funktionen $f(\bar{x}) = 0$. Ental - vertex
[6] Den box som har min og max værdier i to modstående hjørner, og således dækker hele volumenet
[7] En child celle er en af de 8 nye celler som en celle (paretn) neddeles i.
[8] En trianglefan er en vifte af sammenhængende trekanter