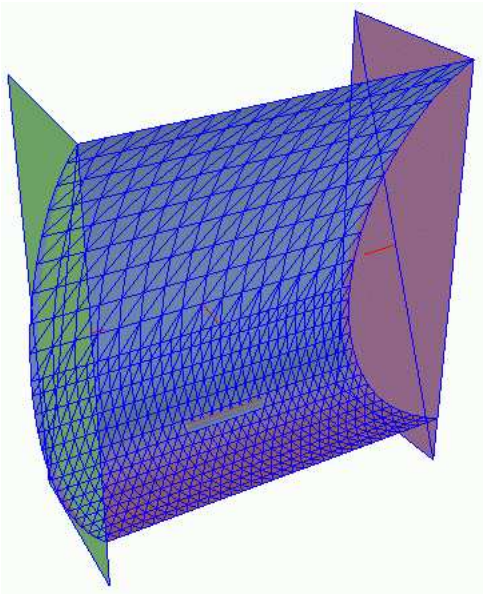
Optimierung von Reflektoren in Straßenleuchten mittels Bézier-Raum-Deformation

Von der Fakultät Mathematik und Physik der Universität Stuttgart zur Erlangung der Würde eines Doktors der Natuwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von

Andreas App

aus Stuttgart



Hauptberichter:	Prof. Dr. U. Reif
Mitberichter:	Prof. Dr. K. Höllig
Tag der mündlichen Prüfung:	8. September 2005

Fachbereich Mathematik der Universität Stuttgart

2006

Titelbild:
Triangulierte Reflektorkonfiguration mit Lampenzylinder

Inhaltsverzeichnis

Danksagung

Mein besonderer Dank gilt Herrn Prof. Dr. Ulrich Reif für die vielen Anregungen und die kompetente und motivierende Betreuung dieser Arbeit. Ohne seine Unterstützung wäre diese Arbeit nicht zu Stande gekommen.

Ganz herzlich bedanke ich mich bei Herrn Prof. Dr. Klaus Höllig, der mir Unterschlupf an seinem Lehrstuhl gewährt hat, indem er mir ein Arbeitszimmer in Stuttgart zur Verfügung gestellt hat und bereit war, die Aufgabe des Mitberichters zu übernehmen.

Außerdem möchte ich auch der Firma Philips meinen Dank aussprechen, die diese Arbeit im Rahmen eines Drittmittelprojekts gefördert hat.

Weiterhin bedanke ich mich bei allen Mitarbeitern des Lehrstuhls für Numerik und geometrische Modellierung für die freundliche Atmosphäre und die bereitwillige Hilfe bei auftretenden Problemen, die nicht immer nur mathematischer Natur waren.

Schließlich danke ich noch meinen Eltern, die mir meine Ausbildung nicht zuletzt auch finanziell ermöglicht haben.

1 Einleitung

Diese Arbeit entstand im Rahmen eines Drittmittelprojekts als Zusammenarbeit zwischen dem Mathematischen Institut A (mittlerweile Institut für Mathematische Methoden in den Ingenieurwissenschaften, Numerik und geometrische Modellierung) der Universität Stuttgart und der Firma Philips, genauer der Abteilung Philips Lighting in Miribel bei Lyon. Der Projekttitel lautete dabei:

Study and realization of a software for fast calculation of 3 dimensional reflector design based on beam tracing and optimization algorithms.

Ziel dieses Projekts war die Entwicklung eines Programmpakets, das in der Lage ist, vollautomatisch die Reflektorflächen eines Beleuchtungssystems im Hinblick auf eine vorgegebene Lichtverteilung zu optimieren. Die verwendeten Beleuchtungssysteme werden dabei hauptsächlich für die Straßenbeleuchtung eingesetzt, und die Anfangs-Reflektorkonfigurationen sind von Lichttechnikingenieuren der Firma Philips, die auch die Ziel-Lichtverteilung vorgeben, entworfene Flächen.

Die interne Bezeichnung des Projekts lautet dabei CALMIR2, was auch der Name des entstandenen Programmpakets ist, wobei diese Abkürzung für

Cal*culation of* Mir*rors in* Mir*ibel*

steht und, wie im Namen angedeutet, das zweite Projekt dieser Art innerhalb dieser Abteilung ist. Das erste Projekt ([Phi96], [Phi97]) war jedoch deutlich rudimentärer ausgelegt und beinhaltete lediglich einen Raytracer, der sowohl im direkten Modus (d. h. die Strahlen werden im optischen System generiert und bis zum Verlassen desselben verfolgt) als auch im indirekten Modus (hierbei werden die Strahlen von außen in das optische System geleitet) betrieben werden kann, sowie Umrechnungsformeln, auf die in Kapitel 6 näher eingegangen wird.

Der ursprünglich für dieses Projekt vorgesehene Beamtracer ([Gre95]), der Lichtstrahlen als Kegel behandelt, wurde schon bald durch das einfachere Konzept eines Raytracers, der Lichtstrahlen als Geraden betrachtet, ersetzt. Die dabei verwendeten physikalischen Beziehungen sowie die technischen Randbedingungen werden in Kapitel 2 vorgestellt.

Für die Reflektor-Flächen werden in dieser Arbeit NURBS-Flächen verwendet, deren Lichtverteilungen mit Hilfe eines Raytracers bewertet werden. Es hat sich jedoch gezeigt, daß die Ermittlung des Schnittpunkts eines Lichtstrahls mit einer NURBS-Fläche sowie die für die Reflexion notwendige Berechnung der Flächennormalen an dieser Stelle zu aufwendig sind. Deshalb ist es für einen schnellen Raytracer-Lauf notwendig, zunächst die NURBS-Fläche durch eine Triangulierung zu approximieren, mit der diese Berechnungen deutlich schneller durchgeführt werden können. Im Rahmen dieser Arbeit entstand ein

neuer Ansatz für die Berechnung dieser Triangulierung, der auf sogenannten orthogonalen B-Splines beruht und in Kapitel 3 beschrieben wird.

Die Modifikationen, die an den Reflektorflächen im Laufe der Optimierung durchgeführt werden, beruhen auf einer sogenannten Bézier-Raum-Deformation. Die Eigenschaften dieses Verfahrens, sowie die Beschreibung des Optimierungs-Algorithmus selbst werden in Kapitel 4 erläutert.

Da das Programmpaket CALMIR2 in Zusammenarbeit mit der Industrie entstand, unterliegt der gesamte Quellcode dem Copyright der Firma Philips und darf im Rahmen dieser Arbeit nicht veröffentlicht werden. In Kapitel 5 wird daher nur auf die verwendeten Bibliotheken, die grafische Oberfläche, sowie Besonderheiten des Programms eingegangen.

Die umfangreichen Tests, die mit diesem Programm durchgeführt wurden, sind in [App00b] zusammengefaßt. Da die Gesamtzahl der Testläufe den Rahmen dieser Arbeit sprengen würde, sind in Kapitel 6 nur ausgewählte Tests vorgestellt, um das Verhalten des Programms zu illustrieren.

Schließlich wird im letzten Kapitel eine Zusammenfassung der Ergebnisse aus dieser Arbeit gegeben. Falls die Firma Philips sich zu einer Fortsetzung dieses Projekts entschließt, werden in Kapitel 7 auch Möglichkeiten aufgezeigt, in welche Richtung man das Projekt weiter vorantreiben könnte.

Im Hinblick auf den Titel dieser Arbeit ist vielleicht noch eine Erläuterung der Begriffe „Lampe“ und „Leuchte“ notwendig. Für einen Lichttechnikingenieur ist eine „Lampe“ eine künstliche Lichtquelle bzw. ein Beleuchtungskörper, im Hausgebrauch typischerweise eine Glühlampe oder eine Leuchtstoffröhre. Bei der Straßenbeleuchtung kommen jedoch üblicherweise andere Lampen zum Einsatz. So z. B. die Quecksilberdampflampe, die ein eher grünlich-weißes Spektrum besitzt, das aber durch Zusätze anderer Elemente (insbesondere Halogene), deutlich bessere Farbwiedergabeeigenschaften zeigt. Die konsequente Weiterentwicklung sind die heute weit verbreiteten Halogen-Metalldampflampen, die ein breites Farbspektrum aufweisen. Des weiteren werden Natriumdampflampen verwendet, die eine sehr gute Lichtausbeute besitzen. Diese Lampen sind an ihrem auffällig orange-gelben Licht erkennbar, wobei auch hier durch Zusätze anderer Elemente das Lichtspektrum verbessert werden kann.

Im Gegensatz dazu definiert das internationale Handbuch der Lichttechnik ([OSR05]) eine „Leuchte“ als *Gerät, das zur Verteilung, Filterung oder Umformung des Lichtes von Lampen dient, einschließlich der zur Befestigung, zum Schutz und der Energieversorgung der Lampen notwendigen Bestandteile.* Die Reflektorflächen, die im Zentrum dieser Arbeit stehen, sind damit also eindeutig Bestandteil der Leuchte. Auch wenn umgangssprachlich oft der Begriff Lampe für eine Leuchte verwendet wird, werden in dieser Arbeit beide Begriffe im technischen Sinne – d. h. wie oben definiert – verwendet, so auch im Titel dieser Arbeit.

2 Grundlagen

In diesem Kapitel sollen die physikalischen und technischen Grundlagen sowie die zugehörigen mathematischen Modelle erläutert werden, die im Rahmen dieser Arbeit Verwendung finden.

In CALMIR2 wird zur Bewertung einer Reflektorkonfiguration ein Raytracer verwendet, der Lichstrahlen von der Lichtquelle ausgehend über evtl. mehrere Reflexionen an den Reflektorflächen bis zum Verlassen des optischen Systems verfolgt.

Im Folgenden soll dabei auf die physikalischen Aspekte der wichtigsten Komponenten, nämlich der Lichtquelle sowie der Reflektorflächen, näher eingegangen werden. Für weitergehende Informationen sei auf [Phi96] und [Phi97] verwiesen.

2.1 Lichtquelle

Ein typisches Modell einer Lichtquelle ist ein Zylinder mit Lambertscher Abstrahlcharakteristik entlang der Mantelfläche. Dieses Modell wird auch in dieser Arbeit verwendet.

Ein Lambertscher Strahler emittiert dabei seine Strahlung nach dem sogenannten Lambertschen Cosinus-Gesetz, dabei ist die Lichtstärke I nur vom Cosinus des Neigungswinkels ϑ zwischen dem Lichtstrahl und der Oberflächennormalen abhängig,

$$I(\vartheta) = L\, dA\, \cos(\vartheta)\,, \tag{2.1}$$

wobei dA ein Oberflächenelement und L die Leuchtdichte der Lichtquelle bezeichnet. Das folgende Bild veranschaulicht dies, wobei der Kreis die Lichtstärke in die entsprechende Richtung angibt.

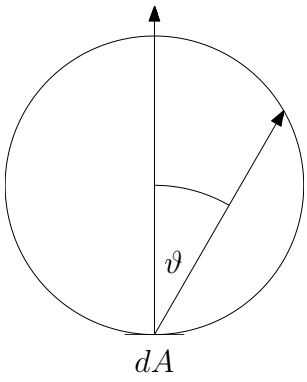


Abbildung 2.1: Lambertsche Abstrahlcharakteristik

Kennzeichnend für einen Lambertschen Strahler ist, daß die Leuchtdichte L in jede Richtung gleich ist, d. h. er emittiert seine Strahlung völlig diffus. Da die Lichtstärke I gleich

dem Quotienten aus Lichtstrom und durchstrahltem Raumwinkel ist,

$$I = \frac{d\Phi}{d\Omega}\,, \tag{2.2}$$

ergibt sich für den Lichtstrom Φ pro Oberflächenelement

$$\frac{d\Phi(\vartheta)}{dA} = L\, d\Omega\, \cos(\vartheta)\,, \tag{2.3}$$

wobei

$$d\Omega = \sin(\vartheta)\, d\vartheta\, d\varphi \tag{2.4}$$

den Raumwinkel mit Azimutwinkel φ bezeichnet. Somit ergibt sich qualitativ für den Lichtfluß in Abhängigkeit des Abstrahlwinkels ϑ zur Oberflächennormalen folgendes Bild.

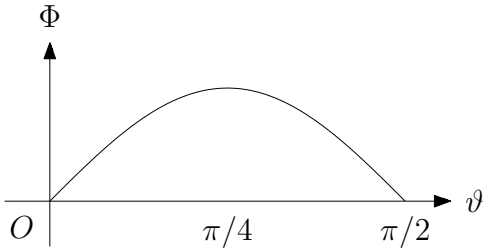


Abbildung 2.2: Lichtfluß in Abhängigkeit des Abstrahlwinkels

Integriert man Gleichung (2.3) über die gesamte Hemisphäre, so erhält man den Lichtstrom $d\Phi/dA$, der von einem Oberflächenelement dA der Lichtquelle in den gesamten Halbraum abgestrahlt wird. Dies wird auch als spezifische Ausstrahlung R bezeichnet

$$R = \frac{d\Phi}{dA} = L \int_{\varphi=0}^{2\pi} \int_{\vartheta=0}^{\pi/2} \cos(\vartheta)\, \sin(\vartheta)\, d\vartheta\, d\varphi = \pi L\,. \tag{2.5}$$

Zur numerischen Simulation dieser Abstrahlcharakteristik gibt es zwei verschiedene Ansätze, um die Lichtstrahlen zu erzeugen, und zwar

- die winkelabhängige Strahlgenerierung und
- die zufällige Strahlgenerierung,

welche beide im folgenden näher erläutert werden.

2.1.1 Winkelabhängige Strahlgenerierung

Bei der winkelabhängigen Strahlgenerierung wird zunächst der Lichtquellen-Zylinder mit Länge l und Durchmesser d in n_l Scheiben der Dicke $\Delta_l = l/n_l$ und n_d Sektoren mit Winkel $\pi d/n_d$ unterteilt, wobei jedes Teilstück des Zylindermantels ein Oberflächenelement approximiert. Dann werden für jedes Teilstück $n_\square$ Strahlen generiert, die die Lambertsche Abstrahlcharakteristik modellieren, und von denen jeder den Lichtstrom

$$\Phi_{\text{Strahl}} = \Phi/(n_l n_d n_\square) \tag{2.6}$$

trägt, wobei Φ der gesamte Lichtstrom der Quelle ist.

Dazu wird zunächst der Zenitwinkelbereich $\vartheta = 0$ bis $\pi/2$ des Halbraums jedes Oberflächenstücks in n_ϑ Sektoren der Größe $\Delta_\vartheta = \pi/(2n_\vartheta)$ unterteilt. Danach wird für einen vorgegeben Winkel ϑ der Azimutwinkelbereich $\varphi = 0$ bis 2π in

$$n_\varphi(\vartheta) \approx \text{const.}\, \cos(\vartheta)\sin(\vartheta) \tag{2.7}$$

Sektoren der Größe $2\pi/n_\varphi$ unterteilt, um die Raumwinkel zu approximieren, wobei hier das Lambertsche Gesetz aus den Gleichungen (2.3) und (2.4) nachgeahmt wird. Die maximale Anzahl Sektoren $n_{\varphi,\max}$ wird für $\vartheta = \pi/4$ erreicht, womit sich n_φ als

$$n_\varphi(\vartheta) \approx 2n_{\varphi,\max}\cos(\vartheta)\sin(\vartheta) \tag{2.8}$$

ausdrücken läßt. Für die Anzahl der Lichtstrahlen pro Zylindermantel-Teilstück ergibt sich damit

$$n_\square = \sum_\vartheta n_\varphi(\vartheta) = \sum_\vartheta 2n_\varphi(\vartheta)n_\vartheta\Delta_\vartheta/\pi \tag{2.9}$$

und mit Gleichung (2.8)

$$n_\square \approx \frac{4n_{\varphi,\max}n_\vartheta}{\pi}\sum_\vartheta \cos(\vartheta)\sin(\vartheta)\Delta_\vartheta \tag{2.10}$$

bzw.

$$n_\square \approx \frac{4n_{\varphi,\max}n_\vartheta}{\pi}\int_{\vartheta=0}^{\pi/2}\cos(\vartheta)\sin(\vartheta)\,d\vartheta = 2n_{\varphi,\max}n_\vartheta/\pi\,. \tag{2.11}$$

Wählt man in natürlicher Weise

$$n_{\varphi,\max}/(2\pi) = n_\vartheta/(\pi/2)\,, \tag{2.12}$$

so erhält man

$$n_\square \approx 8n_\vartheta^2/\pi\,. \tag{2.13}$$

Im CALMIR2 wird schließlich zur Berechnung der Anzahl der Zenit- und Azimutsektoren aus den Gleichungen (2.8), (2.12) und (2.13)

$$n_\vartheta = \left[\sqrt{n_\square \pi/8}\right] \tag{2.14}$$

$$n_\varphi = [8 n_\vartheta \cos(\vartheta)\sin(\vartheta)] \tag{2.15}$$

verwendet, wobei $[\cdot]$ die Rundungsfunktion zur nächstgelegenen ganzen Zahl ist.

2.1.2 Zufällige Strahlgenerierung

Als Variante des Lambertschen Cosinus-Gesetzes (2.3) und (2.4) tritt bei manchen Lichtquellen das sogenannte $\cos^n$-Gesetz mit

$$d\Phi = L\, dA\, \cos^n(\vartheta)\sin(\vartheta)\, d\vartheta\, d\varphi \tag{2.16}$$

auf, wobei $n \geq 1$ eine Materialkonstante ist. Mit der zufälligen Strahlgenerierung kann auch dieses Modell erfaßt werden.

Hierzu wird zunächst ein zufälliger Abstrahlpunkt auf dem Zylindermantel der Lichtquelle gewählt. Der Startpunkt für den Lichtstrahl soll dabei gleichverteilt über die Länge

$$l_{\mathrm{rnd}} = l\, \mathrm{rnd}_1 \tag{2.17}$$

und den Umfang

$$u_{\mathrm{rnd}} = \pi d\, \mathrm{rnd}_2 \tag{2.18}$$

sein, wobei rnd eine gleichverteilte (Pseudo-)Zufallsvariable im Intervall $[0, 1)$ ist.

Für die Strahlrichtung in den Halbraum um die Oberflächennormale durch den Abstrahlpunkt wird zunächst der Azimutwinkel zufällig aber gleichverteilt über den gesamten Vollkreis gewählt

$$\varphi_{\mathrm{rnd}} = 2\pi\, \mathrm{rnd}_3\ . \tag{2.19}$$

Für den Zenitwinkel überlegt man sich zunächst, daß die Stammfunktion von $\cos^n(\vartheta)\sin(\vartheta)$ bis auf Konstante gerade gleich

$$\cos^{n+1}(\vartheta) \tag{2.20}$$

ist. Wenn rnd_4 wieder eine auf $[0, 1)$ gleichverteilte Zufallsvariable ist, liefert also

$$\vartheta_{\mathrm{rnd}} = \arccos(\sqrt[n+1]{\mathrm{rnd}_4}) \tag{2.21}$$

eine Zufallsvariable mit Dichtefunktion $-(n+1)\cos^n(\vartheta)\sin(\vartheta)$ zwischen $\vartheta = 0$ und $\pi/2$. Dies ist bis auf Konstante wie in (2.16) gefordert.

Integriert man Gleichung (2.16) über den Umfang von 0 bis πd, über die Länge von 0 bis l, über den Azimutwinkel von 0 bis 2π und über den Zenitwinkel von 0 bis $\pi/2$, so erhält man für den gesamten Lichtstrom der Quelle

$$\Phi = L \int_{u=0}^{\pi d} du \int_{\ell=0}^{l} d\ell \int_{\varphi=0}^{2\pi} d\varphi \int_{\vartheta=0}^{\pi/2} \cos^n(\vartheta) \sin(\vartheta)\, d\vartheta = \frac{2}{n+1} \pi^2 L\, d\, l\,. \tag{2.22}$$

Sendet man nun n_{Strahlen} wie oben beschrieben zufällig generierte Strahlen aus, so transportiert jeder den Lichtstrom

$$\Phi_{\text{Strahl}} = \Phi / n_{\text{Strahlen}}\,, \tag{2.23}$$

im Vergleich zu (2.6).

2.2 Reflexion und Streuung

Nach der Erzeugung eines Lichstrahls an der Lichtquelle wird der Strahl durch den Raytracer weiter verfolgt, bis er eventuell auf eine Reflektorfläche trifft.

Im Gegensatz zu idealen Spiegeln, bei denen der Winkel zwischen dem auftreffenden Strahl und dem Normalenvektor zur Oberfläche gleich dem Winkel des reflektierten Strahls zum Normalenvektor ist, geht man bei realistischeren Modellen von einer Streuung des Lichts aus.

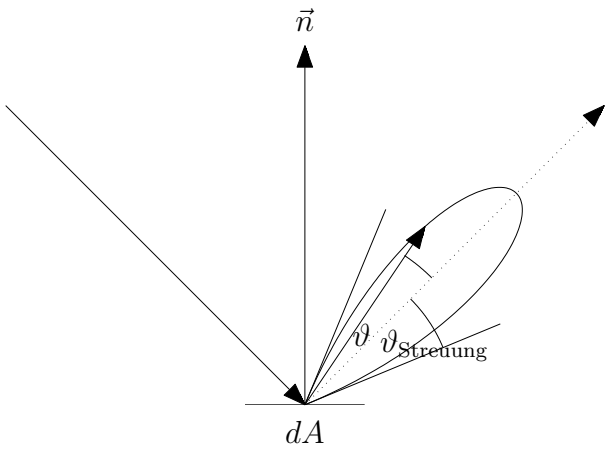


Abbildung 2.3: Enge Streuung

Dabei gibt man vor, daß sich der reflektierte Strahl innerhalb eines Kegels mit halbem Öffnungswinkel $\vartheta_{\text{Streuung}}$ befindet, wobei ein kleinerer Öffnungswinkel einen besseren Spiegel beschreibt.

Bei CALMIR2 wird ein zufälliger Strahl innerhalb des Kegels gewählt, wobei die Wahrscheinlichkeit vom Cosinus des Winkels ϑ zur Oberflächennormalen abhängt,

$$\vartheta_{\text{rnd}} = 2\vartheta_{\text{Streuung}} \arccos\left(\sqrt[b+1]{\text{rnd}}\right)/\pi\,, \tag{2.24}$$

ähnlich wie bei (2.21).

Für $\vartheta_{\text{Streuung}} = \pi/2$ spricht man vom sogenannten Phong-Modell, wobei $b \geq 1$ als Phong-Exponent bezeichnet wird und die Perfektion des Spiegels angibt. Sehr hohe Werte von b beschreiben nahezu perfekte Spiegel, kleinere Werte beschreiben matte Oberflächen.

Alternativ zur sogenannten „engen Streuung“, bei der die Achse des Streukegels dem perfekt reflektierten Lichtstrahl entspricht, gibt es für sehr matte Oberflächen auch noch das Modell der „breiten Streuung”, bei der die Achse des Kegels die Oberflächennormale ist.

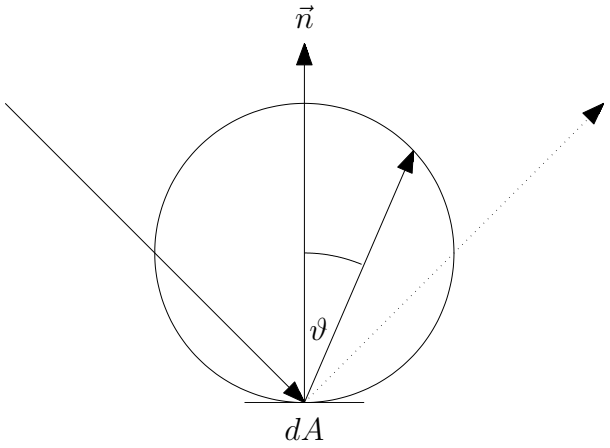


Abbildung 2.4: Breite Streuung

Für Lampenreflektoren wie sie in diesem Projekt Verwendung finden, ist dieses Modell jedoch ungeeignet.

2.3 Normalen-Interpolation

Wie in Kapitel 3 beschrieben, berechnet der Raytracer nicht direkt die Reflexionen mit den Reflektorflächen, sondern lediglich mit einer approximativen Triangulierung derselben. Dies hat den Vorteil, daß die Schnittpunkte und Flächennormalen, die für die Reflexionen benötigt werden, deutlich einfacher und damit schneller zu berechnen sind.

Bei diesem Verfahren kann jedoch auch die Flächennormale interpoliert werden, um den Fehler bei der Approximation durch die Dreiecke zu verringern. Dazu wird zu jedem Eckpunkt P_0, P_1, P_2 eines Dreiecks auch die zugehörige Normale $\vec{n}_0, \vec{n}_1, \vec{n}_2$ der ursprünglichen

Reflektorfläche einmalig berechnet und abgespeichert. Trifft nun ein von der Quelle S – dies kann die Lampe selbst oder ein vorheriger Reflexionspunkt sein – ausgehender Strahl das Dreieck im Punkt P, so kann dieser Schnittpunkt durch

$$P = P_0 + \alpha_1(P_1 - P_0) + \alpha_2(P_2 - P_0) \tag{2.25}$$

dargestellt werden. Für die interpolierte Flächennormale $\vec{\tilde{n}}$ wird dann das gewichtete Mittel der für die Eckpunkte gespeicherten Flächennormalen

$$\vec{\tilde{n}} = (1 - \alpha_1 - \alpha_2)\vec{n}_0 + \alpha_1\vec{n}_1 + \alpha_2\vec{n}_2 \tag{2.26}$$

verwendet. Dieses Verfahren läßt sich auch mit der Streuung aus Abschnitt 2.2 kombinieren. Alternativ kann auch herkömmlich die Normale $\vec{n}$ des Dreiecks für die Berechnung der Reflexionen verwendet werden.

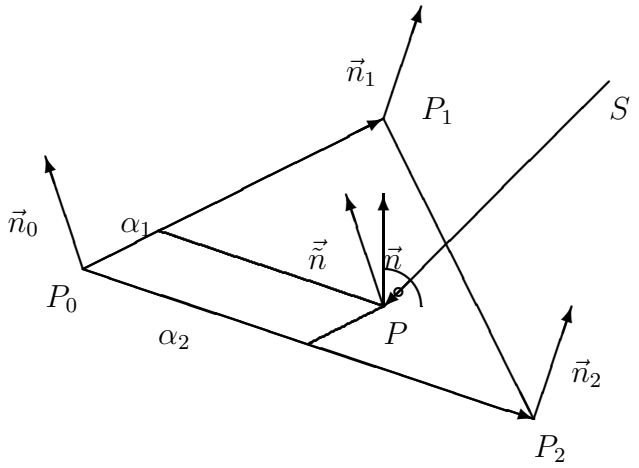


Abbildung 2.5: Normalen-Interpolation

2.4 Koordinatensysteme und Begrenzungen

In CALMIR2 finden mehrere Koordinatensysteme Verwendung. Das erste ist ein kartesisches Koordinatensystem mit x-, y- und z-Achse. Beim Start von CALMIR2 wird der kleinste achsenparallele Quader

$$Q = [x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}] \times [z_{\min}, z_{\max}] \tag{2.27}$$

in diesem Koordinatensystem bestimmt, der die Lampe und die komplette Initial-Konfiguration des Reflektors enthält. Die Öffnung der Leuchte zeigt dabei in Richtung der negativen z-Achse und liegt in der $z_{\min}$-Ebene parallel zur x-y-Ebene. Aufgrund der technischen Beschränkungen des Gehäuses dürfen die Reflektorflächen während der Optimierung diese Box nicht verlassen, lediglich in der Tiefe (d. h. in Richtung der positiven z-Achse) ist eine Ausdehnung bis zur doppelten Größe erlaubt. Des weiteren ist die Austrittsöffnung in der

$z_{\min}$-Ebene fixiert, d. h. alle Kanten des Reflektors in dieser Ebene dürfen nur innerhalb dieser Ebene verändert werden.

Alle geometrischen Eingabegrößen werden bezüglich dieses kartesischen Koordinatensystems an das Programm CALMIR2 übergeben. Dazu gehören:

- der Lampenmittelpunkt
- die Richtung der Lampenachse
- die Kontrollpunkte der Splines, die die Reflektorflächen darstellen.

Weiterhin werden in diesem Programm drei Kugelkoordinatensysteme zum Berechnen der Lichtverteilung verwendet. Diese sind:

- das A-α-System mit $A \in [-\pi, \pi]$, $\alpha \in [-\pi/2, \pi/2]$ und

$$\begin{aligned} x &= r\cos(\alpha)\sin(A) \\ y &= r\sin(\alpha) \\ z &= r\cos(\alpha)\cos(A) \end{aligned} \tag{2.28}$$

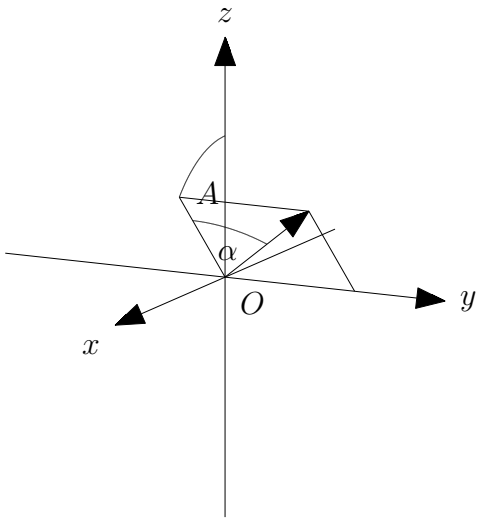


Abbildung 2.6: A-α-System

- das B-β-System mit $B \in [-\pi, \pi]$, $\beta \in [-\pi/2, \pi/2]$ und

$$\begin{aligned} x &= r\sin(\beta) \\ y &= r\cos(\beta)\sin(B) \\ z &= r\cos(\beta)\cos(B) \end{aligned} \tag{2.29}$$

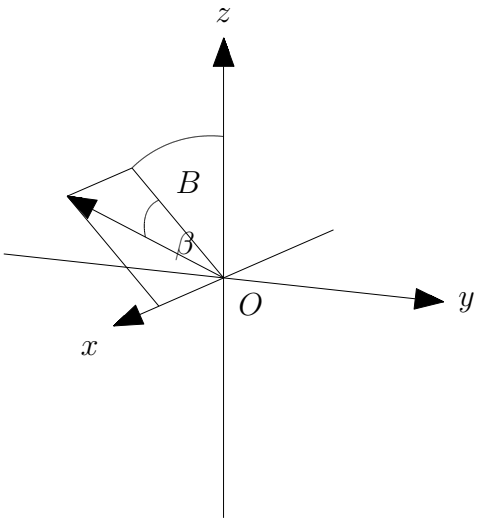


Abbildung 2.7: B-β-System

- das C-γ-System mit $C \in [0, 2\pi]$, $\gamma \in [0, \pi]$ und

$$\begin{aligned} x &= r \sin(\gamma) \cos(C) \\ y &= r \sin(\gamma) \sin(C) \\ z &= r \cos(\gamma) \end{aligned} \tag{2.30}$$

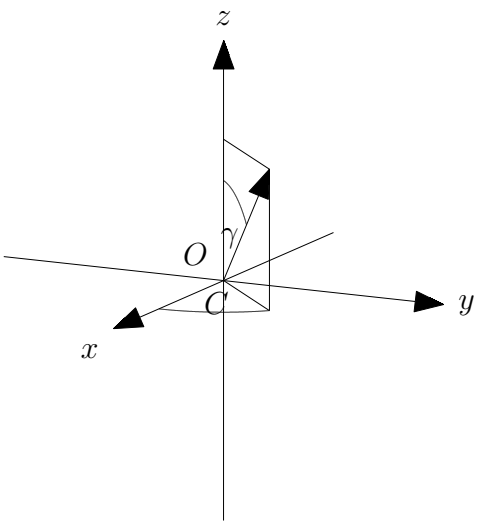


Abbildung 2.8: C-γ-System

Der Raytracer läuft je nach Wahl des Benutzers im A-α- oder B-β-System. Die Ein- und Ausgabe der vorgegebenen und erreichten Lichtverteilung erfolgt aus Kompatibilitätsgründen jedoch immer im C-γ-System.

2.5 Befestigungszylinder

Zur Befestigung des Lampenzylinders sind ein oder zwei Zylinder vorgesehen, die dieselbe Achse besitzen wie die Lichtquelle. Diese Befestigungszylinder setzen unmittelbar oberhalb oder unterhalb des Lampenzylinders an und reichen bis zu den Reflektorflächen oder sogar darüber hinaus. Zur einfacheren Berechnung werden diese Befestigungszylinder bis zum Schnitt mit den Begrenzungsebenen aus Abschnitt 2.4 fortgesetzt.

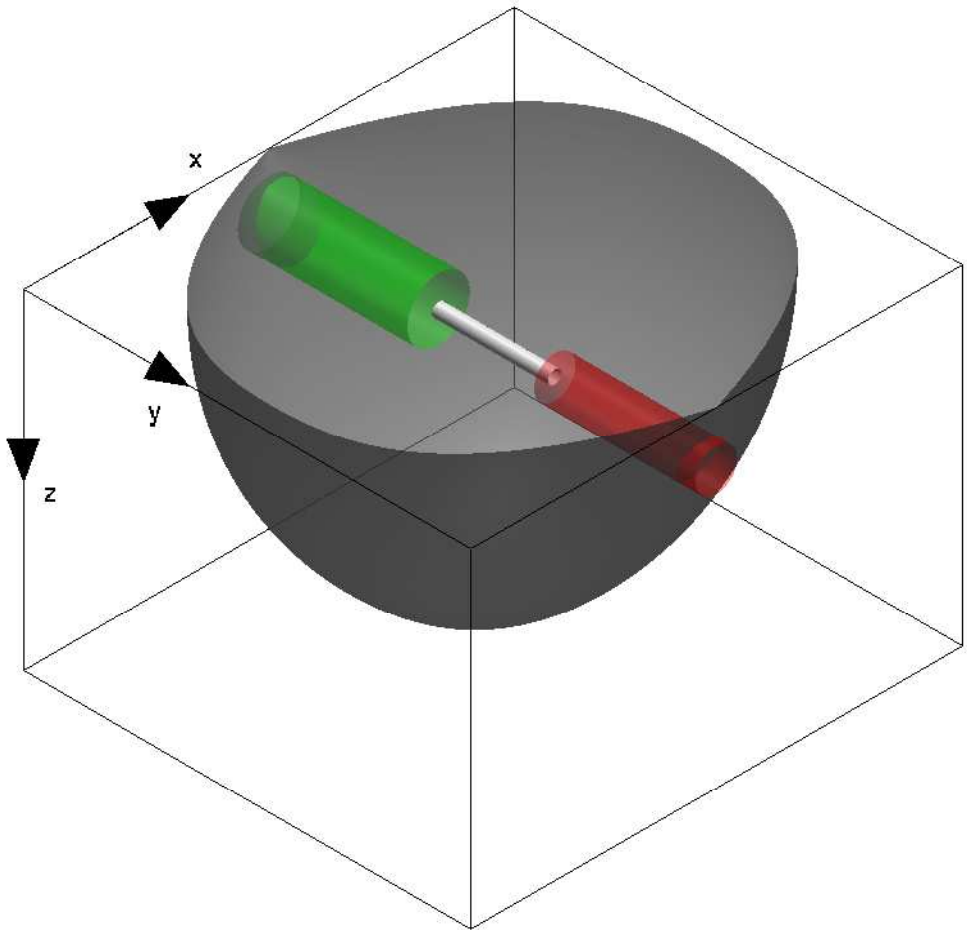


Abbildung 2.9: Lampe mit Befestigungszylindern, Reflektor und Begrenzungsquader

Der Benutzer kann die Durchmesser beider Zylinder beliebig vorgeben, wobei der Durchmesser null bedeutet, daß dieser Befestigungsylinder nicht vorhanden ist. Für die Berechnung der Lichtverteilung durch den Raytracer werden Lichtstrahlen, die einen der Befestigungszylinder oder den Lampenzylinder treffen, als absorbiert betrachtet.

2.6 Zielfunktion

Die Ziellichtverteilung, die in CALMIR2 ausgehend von der vorgegebenen Initialkonfiguration der Reflektorflächen erreicht werden soll, wird über eine Tabelle im C-γ-System vorgegeben. Der Zenitwinkelbereich wird dazu in n_C Sektoren der Größe $\Delta C = \pi/n_C$ und der Azimutwinkelbereich in n_γ Sektoren der Größe $\Delta\gamma = 2\pi/n_\gamma$ aufgeteilt. Gibt $\widetilde{\Phi}_{i,j}$ den Soll-Lichtstrom im Sektor i, j ($0 \le i < n_C$, $0 \le j < n_\gamma$), und $\Phi_{i,j}$ den Ist-Lichtstrom in diesem Sektor an, so sind

$$\widetilde{\Phi} = \sum_{\substack{0 \le i < n_C \\ 0 \le j < n_\gamma}} \widetilde{\Phi}_{i,j} \tag{2.31}$$

$$\Phi = \sum_{\substack{0 \le i < n_C \\ 0 \le j < n_\gamma}} \Phi_{i,j} \tag{2.32}$$

der Soll- bzw. der Ist-Wert des gesamten Lichtstroms. Die Zielfunktion, die in CALMIR2 während der Optimierung minimiert wird, lautet

$$M = w \sqrt{\frac{\sum_{i,j} w_{i,j} \left(\frac{\Phi_{i,j}}{\Phi} - \frac{\widetilde{\Phi}_{i,j}}{\widetilde{\Phi}} \right)^2}{\sum_{i,j} w_{i,j} \left(\frac{\widetilde{\Phi}_{i,j}}{\widetilde{\Phi}} \right)^2}} + (1-w) \left(1 - \frac{\Phi}{\widetilde{\Phi}} \right) , \tag{2.33}$$

wobei wie oben über $0 \le i < n_C$ und $0 \le j < n_\gamma$ summiert wird.

Die beiden Summations-Terme beschreiben dabei einerseits die Differenz pro Sektor und andererseits die Differenz in der gesamten Lichtausbeute, wobei der Benutzer mit der Größe w ($0 \le w \le 1$) diese beiden Teile gewichten kann. In der Praxis haben sich für w Werte im Bereich $0.85 \le w \le 0.95$ bewährt, und für das Programm wird hier der Default-Wert $w = 0.9$ verwendet. Schließlich bieten die Größen $w_{i,j}$ noch die Möglichkeit, die einzelnen Sektoren unterschiedlich zu gewichten, wobei der Default-Wert $w_{i,j} = 1$ verwendet wird.

3 Bewertung der Reflektorkonfiguration

Eines der Kernprobleme bei jeder Optimierung ist es, zuerst die Güte der aktuellen Datenkonfiguration zu beurteilen. In diesem Projekt wird dazu ein Raytracer verwendet, der eine vorgegebene Anzahl von Lichtstrahlen ausgehend von der Lichtquelle bis zum Verlassen der Leuchtenbegrenzung (s. Abschnitt 2.4) verfolgt.

Die daraus berechnete Lichtverteilung wird dann mit einer vorgegebenen Lichtverteilung verglichen, um anhand der Abweichung die Güte der aktuellen Konfiguration zu beurteilen (s. Abschnitt 2.6). Da es jedoch nicht einfach möglich ist, die Auswirkungen lokaler Änderungen am Reflektor in der Lichtverteilung erfolgreich vorherzusagen, muss nach jeder Veränderung der Reflektorkonfiguration die aktuelle Lichtverteilung wieder komplett neu mit Hilfe des Raytracers berechnet werden.

Da weiterhin die Berechnung des Schnittpunktes einer Geraden (des Lichtstrahls) mit einer Spline-Fläche (dem Reflektor) oder auch die Berechnung der Flächennormalen an einem Punkt auf einer Spline-Fläche recht aufwendig ist, wurde versucht, diesen Teil des Programms deutlich zu beschleunigen. Dies wurde durch eine Approximation der Spline-Fläche mit Hilfe von Dreiecken erreicht, da dort die Normalen-Bestimmung sowie der Schnitt mit einer Geraden sehr einfach und damit schnell zu berechnen sind.

3.1 Splines

Splinefunktionen – oder kurz Splines – haben sich vor allem in der geometrischen Datenverarbeitung zur Berechnung und Darstellung der geometrischen Primitiven am Rechner durchgesetzt. Da Splines stückweise polynomiale Funktionen sind, besitzen sie lokal auch deren Approximationskraft, wie sie z. B. im Satz von Taylor zum Ausdruck kommt. Durch die Aufteilung des Grundintervalls in Teilintervalle mit gewissen Stetigkeitsanforderungen an den Übergängen setzen sich andererseits lokale Änderungen nicht wie bei Polynomen unbeschränkt global fort, sondern nur im entsprechenden Teilintervall. Die Unterteilung des Grundintervalls und die Stetigkeitsanforderungen werden dabei mit Hilfe eines Knotenvektors beschrieben.

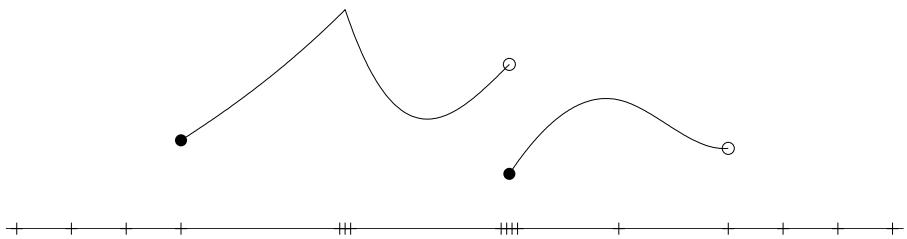


Abbildung 3.1: Spline vom Grad 3 mit angedeuteten Knotenvielfachheiten

Definition 3.1 (Knotenvektor) *Der Vektor*

$$T = [t_{-n}, \ldots, t_0, \ldots, t_m, \ldots, t_{m+n}], \quad m, n \in \mathbb{N}, \quad t_0 < t_m, \tag{3.1}$$

heißt Knotenvektor *vom Grad n der Länge m zum Intervall $[t_0, t_m)$. Die rellen Zahlen t_j mit $t_j \leq t_{j+1}$ werden dabei als Knoten bezeichnet und die Mächtigkeit der Menge*

$$v_j = |\{k : t_k = t_j\}| \tag{3.2}$$

als Vielfachheit des Knotens t_j.

Eine Knotenfolge wird als uniform *bezeichnet, wenn die Folge der Knoten äquidistant*

$$t_{j+1} - t_j = const. > 0 \tag{3.3}$$

ist.

Die Knoten geben dabei nicht nur die Aufteilung des Grundintervalls $[t_0, t_m)$, sondern durch evtl. mehrfaches Auftreten im Knotenvektor auch die Stetigkeitsübergänge an.

Definition 3.2 (Splinefunktion) *Eine Splinefunktion vom Grad n mit Knotenvektor T ist eine Funktion*

$$f : [t_0, t_m) \to \mathbb{R}\,, \tag{3.4}$$

die auf jedem Teilintervall $[t_j, t_{j+1})$ mit einem Polynom vom Grad $\leq n$ übereinstimmt und an den Knoten $t_j \in (t_0, t_m)$ $(n - v_j)$-mal stetig differenzierbar ist.

Die Menge aller Splinefunktionen vom Grad n mit Knotenvektor T wird als Spline-Raum $\mathcal{S}_T^n$ bezeichnet.

Die maximale sinnvolle Vielfachheit für einen Knoten ist $n+1$, da diese bereits einen unstetigen Übergang zwischen zwei Teilintervallen beschreibt, und im Rahmen dieser Arbeit werden daher nur höchstens $(n+1)$-fache Knoten-Vielfachheiten betrachtet. Die Verwendung von halboffenen Intervallen ist notwendig, um Inkonsistenzen an den Knoten zu vermeiden. Dies ist auch der Grund für die Wahl des halboffenen Grundintervalls $[t_0, t_m)$. Die Lage der äußeren Knoten t_j mit $-n \leq j < 0$ und $m < j < m+n$ beeinflußt den Spline innerhalb des Intervalls $[t_0, t_m)$ nicht, jedoch die unten vorgestellte Basis aus B-Splines.

Wählt man alle Knoten-Vielfachheiten n-fach (und die Randknoten $n+1$-fach), so spricht man von der sogenannten Bézier-Darstellung. Im Rahmen dieser Arbeit werden ausschließlich Splines mit Bézier-Rändern verwendet.

Eine Basis des Spline-Raums $\mathcal{S}_T^n$ bilden die sogenannten B-Splines, die rekursiv aus den charakteristischen Funktionen der Teilintervalle definiert werden.

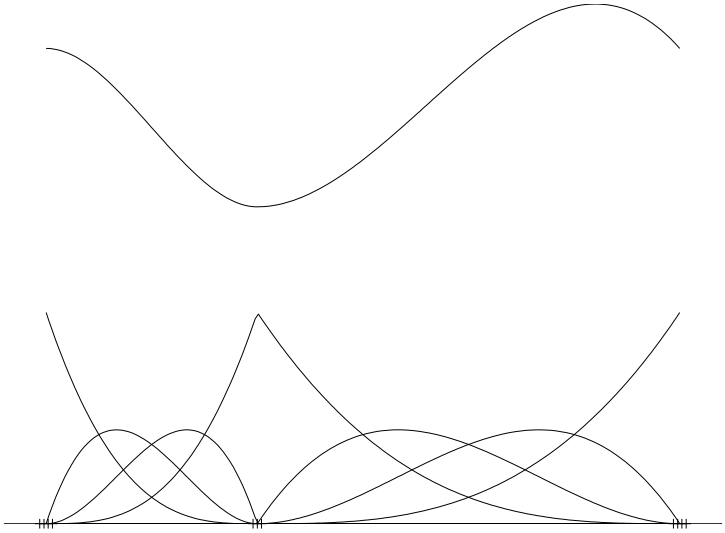


Abbildung 3.2: Spline vom Grad 3 in Bézier-Darstellung mit Bernstein-Polynomen

Definition 3.3 (B-Spline) *Die Funktionen*

$$b^0_{j,T}(x) = \begin{cases} 1, & x \in [t_j, t_{j+1}), \\ 0, & \textit{sonst}, \end{cases} \tag{3.5}$$

$$b^n_{j,T}(x) = w^n_{j,T}(x) b^{n-1}_{j,T}(x) + (1 - w^n_{j+1,T}(x)) b^{n-1}_{j+1,T}(x) \tag{3.6}$$

mit den Gewichten

$$w^n_{j,T}(x) = \begin{cases} \frac{x - t_j}{t_{j+n} - t_j}, & \textit{falls } t_j < t_{j+n}\,, \\ 0, & \textit{sonst}, \end{cases} \tag{3.7}$$

heißen B-Spline-Funktionen *mit Knotenvektor T oder kurz* B-Splines $b_j(x)$*. Hierbei ist T wie in Definition 3.1 und $j \in \{-n, \ldots, m-1\}$.*

Aus der Definition der Gewichte $w^n_{j,T}$ erkennt man, daß die $b^n_{j,T}$ positive Funktionen sind, die außerhalb von $[t_j, t_{j+n+1})$ verschwinden und auf den Intervallen $[t_k, t_{k+1})$ für $j \le k \le j+n$ ein Polynom vom Grad n sind. Durch Induktion nach n erhält man weiterhin, daß

$$\sum_j b_j(x) = 1 \tag{3.8}$$

für $x \in [t_0, t_m)$ gilt.

Im Spezialfall $[t_0, t_m) = [0, 1)$ mit $m = 1$ und Bézier-Rändern sind die B-Splines die sogenannten Bernstein-Polynome

$$b^n_j(x) = \binom{n}{j} x^j (1-x)^{n-j}, \quad 0 \le j \le n\,, \tag{3.9}$$

wobei hier die Indizierung bzgl. der Definition 3.3 um n verschoben wurde.

Satz 3.4 (B-Spline-Basis) *Die B-Splines $b^n_{j,T}$ $(-n \leq j < m-1)$ vom Grad n bilden eine Basis des Spline-Raums $\mathcal{S}^n_T$.*

D. h. man kann jede Spline-Funktion $f \in \mathcal{S}^n_T$ auf dem Intervall $[t_0, t_m)$ eindeutig als Linearkombination der B-Splines

$$f(x) = \sum_{j=-n}^{m-1} c_j b^n_{j,T}(x) \tag{3.10}$$

mit Gewichten c_j darstellen.

Aus den univariaten Splinefunktionen erhält man durch Multiplikation bivariate Spline-Funktionen. Dieses Vorgehen wird zunächst anhand der B-Spline-Basis demonstriert (vgl. [Höl03]).

Definition 3.5 (Tensorprodukt-B-Spline) *Sind*

$$T = [t_{-n}, \ldots, t_{m+n}]$$

und

$$U = [u_{-r}, \ldots, u_{r+s}]$$

zwei Knotenvektoren, so erhält man aus den univariaten B-Splines $b^n_{j,T}(x)$ und $b^s_{k,U}(y)$ die Tensorprodukt-B-Splines

$$b^{n,s}_{j,k,T,U}(x,y) = b^n_{j,T}(x) b^s_{k,U}(y)\,, \tag{3.11}$$

die in x- bzw. y-Richtung wieder die Stetigkeitsvorgaben der Knotenvektoren T bzw. U erfüllen.

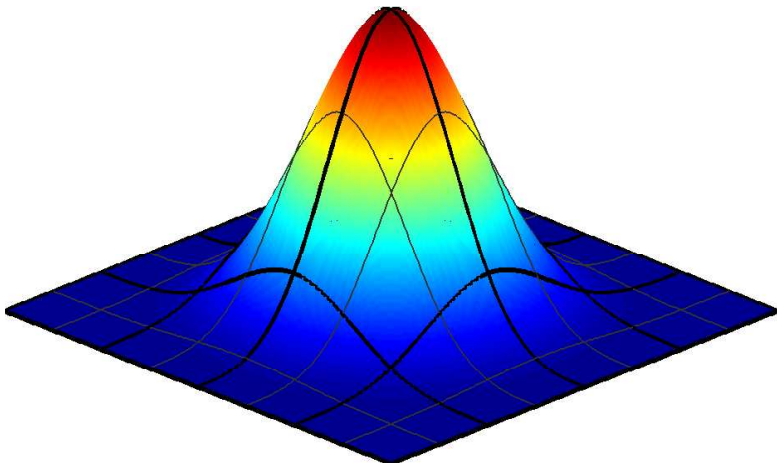


Abbildung 3.3: Tensorprodukt-B-Spline

Wie im univariaten Fall bilden diese $b^{n,s}_{j,k,T,U}$ wieder eine Basis für die bivariaten Tensorprodukt-Splines.

Definition 3.6 (Tensorprodukt-Spline) *Die auf dem Intervall* $[t_0, t_m) \times [u_0, u_r)$ *definierte Linearkombination*

$$f(x,y) = \sum_{j=-n}^{m-1} \sum_{k=-s}^{r-1} c_{j,k} b^{n,s}_{j,k,T,U}(x,y) \tag{3.12}$$

heißt Tensorprodukt-Spline.

Der Raum aller Tensorprodukt-Splines wird mit $\mathcal{S}^{n,s}_{T,U}$ *bezeichnet.*

Es ist klar, wie man durch Forsetzung dieses Vorgehens auch Tensorprodukt-Splines mit mehr als zwei Variablen konstruieren kann.

3.2 NURBS

Manche in der Computergrafik verwendeten Basiselemente wie z. B. Kegelschnitte lassen sich im allgemeinen nicht exakt durch Splines beschreiben. Um diesen Mangel zu beheben, kann man – wie beim Konstruieren von rationalen Funktionen aus Polynomen – aus Splines durch Quotientenbildung sogenannte NURBS (*n*icht-*u*niforme, *r*ationale *B*-*S*plines) erhalten.

Definition 3.7 (NURBS) *Sind* $b^n_{j,T}$ *wie oben die B-Splines vom Grad* n *mit Knotenvektor* T*, so heißt die auf* $[t_0, t_m)$ *definierte Funktion*

$$f(x) = \frac{\sum_{j=-n}^{m-1} \varrho_j \, c_j b^n_{j,T}(x)}{\sum_{j=-n}^{m-1} \varrho_j b^n_{j,T}(x)} \tag{3.13}$$

NURBS-*Funktion vom Grad* n*.*

Üblicherweise wird vorausgesetzt, daß der Nenner auf $[t_0, t_m)$ ungleich 0 ist, und daß die Gewichte ϱ_j positiv sind, so daß f eine Konvexkombination der Punkte c_j bildet.

Wie bei Splinefunktionen kann man auch die Tensorproduktbildung (s. Definitionen 3.5 und 3.6) auf NURBS anwenden und erhält so aus den Tensorprodukt-B-Splines die Tensorprodukt-NURBS-Funktionen.

Definition 3.8 (Tensorprodukt-NURBS) *Die auf dem Intervall* $[t_0, t_m) \times [u_0, u_r)$ *definierte Funktion*

$$f(x,y) = \frac{\sum_{j=-n}^{m-1} \sum_{k=-s}^{r-1} \varrho_{j,k} \, c_{j,k} b^{n,s}_{j,k,T,U}(x,y)}{\sum_{j=-n}^{m-1} \sum_{k=-s}^{r-1} \varrho_{j,k} b^{n,s}_{j,k,T,U}(x,y)} \tag{3.14}$$

heißt Tensorprodukt-NURBS.

Alle bisher betrachteten Funktionen – sowohl die NURBS- als auch die Spline-Funktionen – waren reellwertig. In der geometrischen Datenverarbeitung werden jedoch meist Kurven und Flächen betrachtet, aber auch diese lassen sich nun in natürlicher Weise als Splines bzw. NURBS-Funktionen darstellen.

Definition 3.9 (NURBS-Kurve) *Eine Funktion f von $[t_0, t_m)$ nach $\mathbb{R}^3$ ($\mathbb{R}^2$) mit*

$$f(x) = \frac{\sum_{j=-n}^{m-1} \varrho_j \, C_j b_{j,T}^n(x)}{\sum_{j=-n}^{m-1} \varrho_j b_{j,T}^n(x)} \tag{3.15}$$

und Kontrollpunkten $C_j \in \mathbb{R}^3$ ($\mathbb{R}^2$) heißt (ebene) NURBS-Kurve.

D. h. eine NURBS-Kurve besteht in jeder Koordinate aus einer NURBS-Funktion.

Im Spezialfall $\varrho_j = 1$ für alle j ist die Nennerfunktion identisch eins und man erhält eine sogenannte *Spline-Funktion*.

Wie bei der Konstruktion von NURBS-Kurven aus NURBS-Funktionen kann man auch aus Tensorprodukt-NURBS-Funktionen entsprechend NURBS-Flächen bilden.

Definition 3.10 (NURBS-Fläche) *Eine Funktion f von $[t_0, t_m) \times [u_0, u_r)$ nach $\mathbb{R}^3$ mit*

$$f(x,y) = \frac{\sum_{j=-n}^{m-1} \sum_{k=-s}^{r-1} \varrho_{j,k} \, C_{j,k} b_{j,k,T,U}^{n,s}(x,y)}{\sum_{j=-n}^{m-1} \sum_{k=-s}^{r-1} \varrho_{j,k} b_{j,k,T,U}^{n,s}(x,y)} \tag{3.16}$$

und Kontrollpunkten $C_{j,k} \in \mathbb{R}^3$ heißt NURBS-Fläche.

Analog zu NURBS-Kurven besteht also eine NURBS-Fläche in jeder Koordinate aus einem Tensorprodukt-NURBS.

Wie bei NURBS-Kurven spricht man hier von Spline-Flächen, falls alle Gewichte $\varrho_{j,k} = 1$ sind. Auch hier wird klar, wie sich beliebige NURBS-Parametrisierungen von $\mathbb{R}^m$ nach $\mathbb{R}^n$ bilden lassen.

Alle Reflektorflächen im Rahmen dieses Projekts sind NURBS-Flächen mit Bézier-Rändern.

3.3 Lineare B-Splines über Triangulierungen

Alle bisher behandelten B-Splines sind über ein- oder mehrdimensionalen Intervallen definiert. Insbesondere für Tensorprodukt-Flächen hat dies jedoch den Nachteil, daß aufgrund

des Eulerschen Polyedersatzes geschlossene Flächen nur vom Geschlecht eins dargestellt werden können. Dieses Problem wird behoben, wenn man auf andere Definitionsgebiete, wie z. B. Dreiecke ausweicht. Dieses Vorgehen soll nun anhand von linearen B-Splines über Triangulierungen erläutert werden.

Definition 3.11 (Triangulierung) *Eine Triangulierung $\mathcal{T}$ eines Definitionsgebietes $\Omega = [x_{min}, x_{max}] \times [y_{min}, y_{max}]$ ist eine Menge von abgeschlossenen Dreiecken $\mathcal{T} = \{T_1, \ldots, T_n\}$, für die*

- $\bigcup_{k=1}^{n} T_k = \Omega$
- *zwei verschiedene Dreiecke T_j und T_k mit $T_j \cap T_k \neq \emptyset$ besitzen entweder genau einen gemeinsamen Eckpunkt oder genau eine gemeinsame Kante*

gilt.

Ähnlich wie eindimensionale, lineare B-Splines an genau einem Knoten den Wert eins aufweisen und an allen anderen Knoten den Wert null haben, bildet man über dieser Triangulierung die linearen B-Splines, in dem man fordert, daß es zu jedem Eckpunkt P in der Triangulierung genau einen B-Spline gibt, der an diesem Punkt den Wert eins hat und an allen anderen Eckpunkten den Wert null, und der linear auf jedem Dreieck ist.

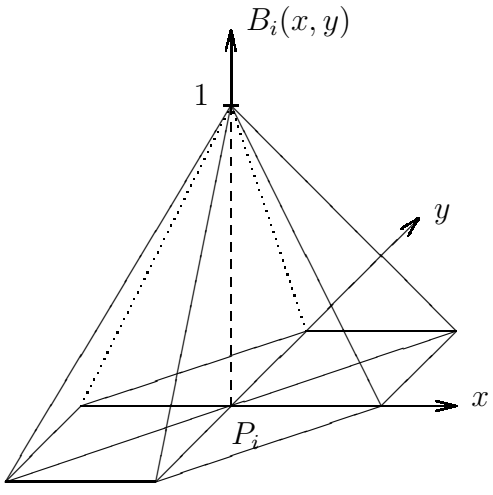


Abbildung 3.4: Hutfunktion

Sind $P_i = (x_i, y_i)$ mit i aus einer Indexmenge I alle Eckpunkte der Triangulierung wie oben beschrieben, so lassen sich diese linearen B-Splines, die wegen ihrer Form auch *Hutfunktionen* genannt werden, wie folgt formal definieren.

Definition 3.12 (Hutfunktion) *Eine Funktion*

$$B_i : \Omega = [x_{min}, x_{max}] \times [y_{min}, y_{max}] \to \mathbb{R} \tag{3.17}$$

heißt Hutfunktion oder linearer B-Spline zum Punkt P_i, wenn

- $B_i(x_i, y_i) = 1$,
- $B_i(x_j, y_j) = 0$ *für alle* $i \neq j$,
- B_i *ist linear auf allen Dreiecken*

gilt.

Genau wie bei den eindimensionalen Splines lassen sich auch hier über die Hutfunktionen die linearen Splinefunktionen über Dreiecken definieren.

Definition 3.13 (lineare Splinefunktion über einer Triangulierung) *Eine Funktion*

$$f(x, y) = \sum_{i \in I} \gamma_i B_i(x, y) \tag{3.18}$$

mit Koeffizienten $\gamma_i \in \mathbb{R}$ ist ein linearer Spline über einer Triangulierung gemäß Definition 3.11.

3.4 Approximative Triangulierungen von Flächen

Wählt man die Koeffizienten aus (3.18) als Auswertungspunkte $\gamma_i = h(x_i, y_i)$ einer vorgegebenen Funktion h, so ergibt sich für f die lineare Standard-Interpolation des Graphen $S = \{(x, y, h(x, y) : (x, y) \in \Omega\}$ über obiger Triangulierung.

Der maximale Fehler in der L_∞-Norm

$$\|f - g\|_\infty = \sup_{(x,y) \in \Omega} |f(x, y) - g(x, y)| \,, \tag{3.19}$$

oder auch der totale Fehler in der L_1-Norm

$$\|f - g\|_1 = \int_\Omega |f(x, y) - g(x, y)| \, d(x, y) \tag{3.20}$$

dieser Approximierung ist jedoch relativ groß, insbesondere wenn es sich um konvexe Flächen handelt, wie sie typischerweise bei Reflektoren auftreten, da die approximative Triangulierung immer nur auf einer Seite der Fläche verläuft.

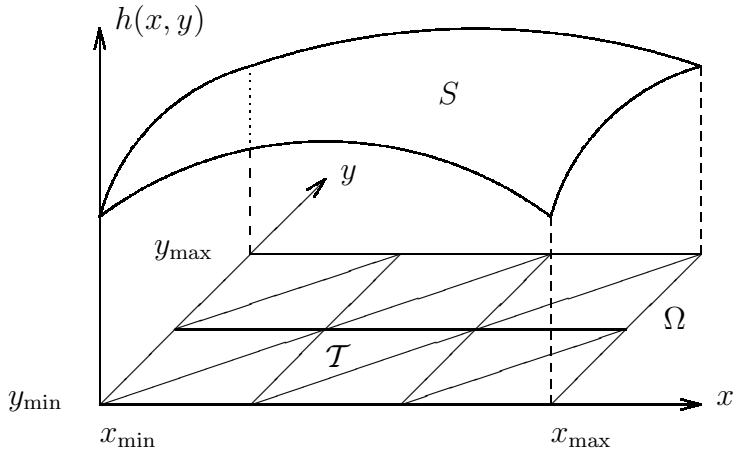


Abbildung 3.5: Graph S über der Triangulierung $\mathcal{T}$ im Definitionsgebiet Ω

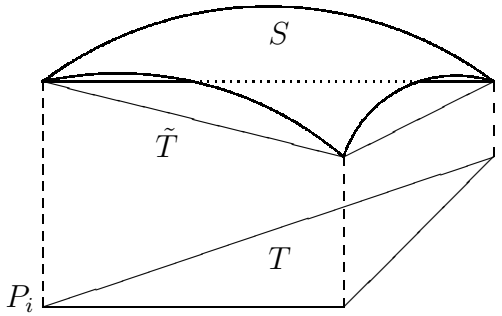


Abbildung 3.6: Lineare Standard-Interpolation über einem Dreieck

Will man den Fehler verbessern, so kann man versuchen die optimale Approximation bzgl. der L_2-Norm zu finden, die den quadratischen oder Gaußschen Fehler mißt

$$\|f-g\|_2 = \sqrt{\int\limits_{\Omega} |f(x,y)-g(x,y)|^2 \, d(x,y)}\,. \tag{3.21}$$

Diese Norm ist einfach zu handhaben und hat auch den Vorteil, daß sie von einem Skalarprodukt

$$\langle f,g\rangle_2 = \int\limits_{\Omega} f(x,y)g(x,y)\, d(x,y) \tag{3.22}$$

mittels

$$\|f\|_2 = \sqrt{\langle f,f\rangle_2} = \sqrt{\int\limits_{\Omega} f(x,y)^2\, d(x,y)} \tag{3.23}$$

induziert wird. Die Berechnung dieser Approximation führt jedoch auf ein meist großes lineares Gleichungssystem, das nur aufwendig zu lösen ist. Im folgenden wird deshalb ein

Verfahren vorgestellt, mit dessen Hilfe eine deutlich bessere Approximation als bei der linearen Standard-Interpolation erreicht wird, aber mit deutlich weniger Aufwand als bei der direkten Optimierung des L_2-Fehlers benötigt wird.

Im Hinblick auf die Approximation von Reflektorflächen ist jedoch nicht nur die Differenz der Funktionswerte von f und g von Interesse, sondern auch die Abweichung der Flächennormalen bzw. des Gradienten, d. h. es liegt nahe, auch

$$\nabla f = (f_x, f_y)^{\mathrm{t}} = \left(\frac{\partial}{\partial x} f, \frac{\partial}{\partial y} f \right) \tag{3.24}$$

zu berücksichtigen.

Die Standard-Norm, die dies erfüllt, ist die Sobolev-Norm $\| \cdot \|_{1,2}$ mit dem zugehörigen Skalarprodukt

$$\begin{aligned}
\langle f, g \rangle_{1,2} &= \langle f, g \rangle_2 + \langle f_x, g_x \rangle_2 + \langle f_y, g_y \rangle_2 && (3.25)\\
&= \int_\Omega f(x,y) g(x,y)\, d(x,y) + \\
&\quad \int_\Omega f_x(x,y) g_x(x,y)\, d(x,y) + \int_\Omega f_y(x,y) g_y(x,y)\, d(x,y) && (3.26)\\
&= \int_\Omega f(x,y) g(x,y)\, d(x,y) + \int_\Omega \nabla f(x,y)^{\mathrm{t}} M \nabla g(x,y)\, d(x,y) && (3.27)
\end{aligned}$$

mit

$$M = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}. \tag{3.28}$$

Es wird nun ein leicht modifiziertes Sobolev-Skalarprodukt (sowie die zugehörige Norm) gesucht,

$$\begin{aligned}
\langle f, g \rangle^o &= w_0 \int_\Omega f(x,y) g(x,y)\, d(x,y) + \\
&\quad w_{1,1} \int_\Omega f_x(x,y) g_x(x,y)\, d(x,y) + w_{1,2} \int_\Omega f_x(x,y) g_y(x,y)\, d(x,y) + \\
&\quad w_{2,1} \int_\Omega f_y(x,y) g_x(x,y)\, d(x,y) + w_{2,2} \int_\Omega f_y(x,y) g_y(x,y)\, d(x,y) && (3.29)\\
&= w_0 \int_\Omega f(x,y) g(x,y)\, d(x,y) + \int_\Omega \nabla f(x,y)^{\mathrm{t}} W \nabla g(x,y)\, d(x,y) && (3.30)
\end{aligned}$$

mit einem positiven Gewicht w_0 und einer (symmetrischen) 2×2-Matrix

$$W = \begin{pmatrix} w_{1,1} & w_{1,2} \\ w_{2,1} & w_{2,2} \end{pmatrix}, \tag{3.31}$$

die für dieses Problem geeignet ist.

3.5 Orthogonale B-Splines

Ähnlich wie in [Rei97], wo das Prinzip für univariate B-Splines mit äquidistanten Knoten erläutert wird, werden nun die Gewichte w_0 und w_1 – w_4 so bestimmt, daß die Hutfunktionen B_i paarweise orthogonal bzgl. dem Skalarprodukt $\langle \cdot, \cdot \rangle^o$ sind, denn dann liefert der Projektionssatz die Gewichte γ_i für die *optimale* Approximation (bzgl. der zugehörigen Norm) als Skalarprodukte der zu approximierenden Funktion f mit den Hutfunktionen B_i selbst,

$$h(x,y) = \sum_{i \in I} \gamma_i B_i(x,y) = \sum_{i \in I} \frac{\langle f, B_i \rangle^o}{\langle B_i, B_i \rangle^o} B_i(x,y)\,. \tag{3.32}$$

Da die Orthogonalitäts- oder sogar die Orthonormalitätsbedingungen

$$\langle B_i, B_j \rangle^o = \delta_{i,j} = \begin{cases} 1, & i = j \\ 0, & i \neq j \end{cases} \tag{3.33}$$

schwierig zu handhaben sind, wird stattdessen die etwas stärkere Bedingung der Orthonormalität auf jedem Dreieck $\triangle_k$ gefordert,

$$\begin{aligned} \langle B_i, B_j \rangle^o_{\triangle_k} &= w_0 \int\limits_{\triangle_k} B_i(x,y) B_j(x,y)\, d(x,y) + \int\limits_{\triangle_k} \nabla B_i(x,y)^{\mathrm{t}} W \nabla B_j(x,y)\, d(x,y) \\ &= \delta_{i,j}\,, \end{aligned} \tag{3.34}$$

die die Orthogonalität der Hutfunktionen auf dem gesamten Gebiet Ω zur Folge hat. D. h. das Skalarprodukt wird eingeschränkt und auf jedem Dreieck $\triangle_k$ gesondert betrachtet.

Um die Berechnungen zu vereinfachen, verwendet man die lineare Transformation

$$T : \vec{x} \mapsto \hat{\vec{x}} = A\vec{x} + \vec{b} \tag{3.35}$$

mit der Umkehrung

$$T^{-1} : \hat{\vec{x}} \mapsto \vec{x} = A^{-1}(\hat{\vec{x}} - \vec{b}) \tag{3.36}$$

sowie

$$\hat{f}(\hat{\vec{x}}) = f(A^{-1}(\hat{\vec{x}} - \vec{b})) = f(\vec{x}) \tag{3.37}$$

zur Transformation vom Einheitsdreieck $\triangle = \triangle((0,0);(0,1);(1,0))$ auf ein beliebiges Dreieck $\widehat{\triangle}$. Sind $\widehat{P}_i = (x_i, y_i), i = 1,2,3$ die Eckpunkte des Dreiecks $\widehat{\triangle}$, so ist offensichtlich

$$\vec{b} = \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} \tag{3.38}$$

und

$$A = \begin{pmatrix} x_3 - x_1 & x_2 - x_1 \\ y_3 - y_1 & y_2 - y_1 \end{pmatrix}. \tag{3.39}$$

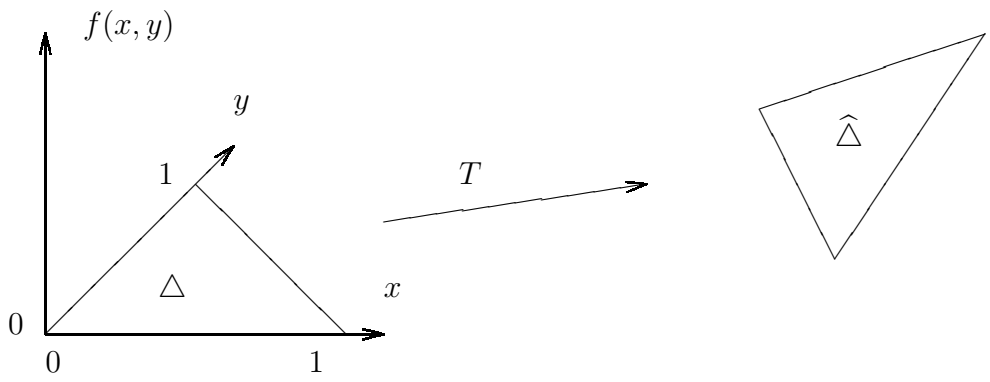


Abbildung 3.7: Transformation des Einheitsdreiecks

Für das Skalarprodukt auf einem beliebigen Dreieck $\widehat{\triangle}$ ergibt sich damit

$$\begin{aligned}
\langle \hat{f}, \hat{g} \rangle^{o}_{\widehat{\triangle}} &= \hat{w}_0 \int_{\widehat{\triangle}} \hat{f}(\vec{\hat{x}}) \hat{g}(\vec{\hat{x}})\, d\vec{\hat{x}} + \int_{\widehat{\triangle}} \left(\nabla \hat{f}(\vec{\hat{x}})\right)^{\mathrm{t}} \widehat{W} \left(\nabla \hat{g}(\vec{\hat{x}})\right) d\vec{\hat{x}} \\
&= \hat{w}_0 \int_{\widehat{\triangle}} f(A^{-1}(\vec{\hat{x}} - \vec{b})) g(A^{-1}(\vec{\hat{x}} - \vec{b}))\, d\vec{\hat{x}} \\
&\quad + \int_{\widehat{\triangle}} \left(\nabla f(A^{-1}(\vec{\hat{x}} - \vec{b}))\right)^{\mathrm{t}} \widehat{W} \left(\nabla g(A^{-1}(\vec{\hat{x}} - \vec{b}))\right) d\vec{\hat{x}} \\
&= \hat{w}_0 \int_{\triangle} f(\vec{x}) g(\vec{x}) |\det(A)|\, d\vec{x} \\
&\quad + \int_{\triangle} \left((A^{-1})^{\mathrm{t}} \nabla f(\vec{x})\right)^{\mathrm{t}} \widehat{W} \left((A^{-1})^{\mathrm{t}} \nabla g(\vec{x})\right) |\det(A)|\, d\vec{x}\,. \qquad (3.40)
\end{aligned}$$

Das heißt, die Gewichte transformieren sich mit

$$w_0 = |\det(A)| \hat{w}_0, \qquad \hat{w}_0 = \frac{1}{|\det(A)|} w_0 \qquad (3.41)$$

$$W = |\det(A)| A^{-1} \widehat{W} (A^{-1})^{\mathrm{t}}, \qquad \widehat{W} = \frac{1}{|\det(A)|} A W A^{\mathrm{t}} \qquad (3.42)$$

zwischen dem Einheitsdreieck $\triangle$ und einem beliebigen Dreieck $\widehat{\triangle}$.

3.6 n-dimensionale orthogonale B-Splines

Man sieht leicht, daß sich alle Formeln des vorigen Abschnitts statt nur im zweidimensionalen ohne weiteres auch im n-dimensionalen Raum anwenden lassen. Statt dem Ein-

heitsdreieck wird nun das Einheitssimplex $\triangle$ mit den Eckpunkten

$$P_0 = \begin{pmatrix} 0 \\ \\ \vdots \\ \\ 0 \end{pmatrix}; \quad P_i = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}, \quad (i = 1, \ldots, n) \tag{3.43}$$

betrachtet. Die zugehörigen n-variaten linearen B-Splines lauten

$$B_0(\vec{x}) = 1 - \sum_{j=1}^{n} x_j\,; \quad B_i(\vec{x}) = x_i, \quad (i = 1, \ldots, n)\,. \tag{3.44}$$

D. h. auf dem Einheitssimplex gelten die Bedingungen

$$\langle B_i, B_j \rangle_{\triangle}^{o} = w_0 \int_{\triangle} B_i(\vec{x}) B_j(\vec{x})\, d\vec{x} + \sum_{k,l=1}^{n} w_{k,l} \int_{\triangle} \frac{\partial B_i}{\partial x_k}(\vec{x}) \frac{\partial B_j}{\partial x_l}(\vec{x})\, d\vec{x} = \delta_{i,j} \tag{3.45}$$

für $0 \leq i, j \leq n$. Da die partiellen Ableitungen

$$\frac{\partial B_0}{\partial x_k}(\vec{x}) = -1, \quad (k = 1, \ldots, n) \tag{3.46}$$

und

$$\frac{\partial B_i}{\partial x_k}(\vec{x}) = \delta_{i,k}, \quad (i, k = 1, \ldots, n) \tag{3.47}$$

sind, und da aufgrund der Symmetrie des Skalarprodukts $w_{i,j} = w_{j,i}$ gilt, erhält man damit das überbestimmte lineare Gleichungssystem

$$\begin{aligned} w_0 \int_{\triangle} B_i(\vec{x}) B_j(\vec{x})\, d\vec{x} + w_{i,j}\, \mathrm{Vol}_{\triangle} &= \delta_{i,j} \\ w_0 \int_{\triangle} B_0(\vec{x}) B_i(\vec{x})\, d\vec{x} - \sum_{k=1}^{n} w_{i,k}\, \mathrm{Vol}_{\triangle} &= 0 \\ w_0 \int_{\triangle} B_0(\vec{x})^2\, d\vec{x} + \sum_{k,l=1}^{n} w_{k,l}\, \mathrm{Vol}_{\triangle} &= 1 \end{aligned} \tag{3.48}$$

für $1 \leq i < j \leq n$, wobei $\mathrm{Vol}_{\triangle}$ das Volumen des n-dimensionalen Einheitssimplex $\triangle$ ist.

Lemma 3.14 (Volumen des Einheitssimplex) *Für das Volumen des n-dimensionalen Einheitssimplex gilt*

$$\mathrm{Vol}_{\triangle} = \frac{1}{n!}\,. \tag{3.49}$$

Beweis: Setzt man das Volumen als Mehrfachintegral

$$\mathrm{Vol}_{\triangle} = \int_{\triangle} 1\, d\vec{x} = \int_{x_n=0}^{1} \dots \int_{x_i=0}^{1-\sum_{j=i+1}^{n} x_j} \dots \int_{x_1=0}^{1-\sum_{j=2}^{n} x_j} 1\, dx_1 \dots dx_i \dots dx_n \tag{3.50}$$

an und verwendet die Transformation

$$\begin{aligned}
x_1 &= y_1 \\
x_2 &= y_2(1-y_1) = y_2(1-x_1) \\
&\vdots \\
x_i &= y_i(1-y_1)\cdots(1-y_{i-1}) = y_i(1-x_1-\cdots-x_{i-1}) \\
&\vdots \\
x_n &= y_n(1-y_{n-1})\cdots(1-y_1) = y_n(1-x_x-\cdots-x_{n-1})\,,
\end{aligned} \tag{3.51}$$

so ergibt sich für die Transformation der Grenzen

$$0 \le y_i \le 1, \quad (i = 1, \dots, n) \tag{3.52}$$

und als Jacobi-Determinante der Transformation

$$J = (1-y_1)^{n-1}(1-y_2)^{n-2}\cdots(1-y_{n-1})\,. \tag{3.53}$$

Daraus folgt

$$\begin{aligned}
\mathrm{Vol}_{\triangle} &= \int_{y_n=0}^{1} \dots \int_{y_i=0}^{1} \dots \int_{y_1=0}^{1} (1-y_1)^{n-1}\cdots(1-y_i)^{n-i}\cdots(1-y_{n-1})\, dy_1 \dots dy_i \dots dy_n \\
&= \left[-\frac{1}{n}(1-y_1)^n\right]_0^1 \cdots \left[-\frac{1}{n-i+1}(1-y_i)^{n-i+1}\right]_0^1 \cdots \left[-\frac{1}{2}(1-y_{n-1})^2\right]_0^1 \\
&= \frac{1}{n}\cdots\frac{1}{n-i+1}\cdots\frac{1}{2} = \frac{1}{n!}\,.
\end{aligned} \tag{3.54}$$

□

Mit der gleichen Transformation lassen sich auch die anderen Integrale der Gleichungen (3.48) bestimmen.

Lemma 3.15 *Für $0 \le i < j \le n$ gilt*

$$\int_{\triangle} B_i(\vec{x}) B_j(\vec{x})\, d\vec{x} = \frac{1}{(n+2)!}\,. \tag{3.55}$$

Beweis: Man sieht zunächst aus Symmetriegründen, daß $\int_{\triangle} B_i(\vec{x}) B_j(\vec{x})\, d\vec{x}$ für alle $0 < i < j \le n$ denselben Wert hat. Es genügt deshalb $\int_{\triangle} B_1(\vec{x}) B_2(\vec{x})\, d\vec{x}$ zu berechnen. Wie im obigen Beweis verwendet man die Transformation (3.51) und erhält damit

$$\begin{aligned}
\int_{\triangle} B_1(\vec{x}) B_2(\vec{x})\, d\vec{x} &= \int_{\triangle} x_1 x_2\, d\vec{x} = \int_{x_n=0}^{1} \cdots \int_{x_2=0}^{1-\sum_{k=3}^{n} x_k} \int_{x_1=0}^{1-\sum_{k=2}^{n} x_k} x_1 x_2\, dx_1 dx_2 \ldots dx_n \\
&= \int_{y_n=0}^{1} \cdots \int_{y_1=0}^{1} y_1 y_2 (1-y_1)^n (1-y_2)^{n-2} \cdots (1-y_{n-1})\, dy_1 \ldots dy_n \\
&= \int_{y_1=0}^{1} y_1 (1-y_1)^n\, dy_1 \int_{y_2=0}^{1} y_2 (1-y_2)^{n-2}\, dy_2 \left[-\frac{1}{n-2} (1-y_3)^{n-2} \right]_0^1 \cdots \\
&\qquad \cdots \left[-\frac{1}{2} (1-y_{n-1})^2 \right]_0^1 \\
&= \left(\frac{1}{(n+2)(n+1)} \right) \left(\frac{1}{n(n-1)} \right) \frac{1}{n-2} \cdots \frac{1}{2} = \frac{1}{(n+2)!}\,,
\end{aligned} \tag{3.56}$$

wenn man berücksichtigt, daß mit partieller Integration

$$\begin{aligned}
\int_0^1 t(1-t)^m\, dt &= \left[-\frac{t}{m+1} (1-t)^{m+1} \right]_0^1 + \int_0^1 \frac{1}{m+1} (1-t)^m\, dt \\
&= \frac{1}{m+1} \left[-\frac{1}{m+2} (1-t)^{m+2} \right]_0^1 = \frac{1}{(m+1)(m+2)}
\end{aligned} \tag{3.57}$$

gilt. Da ebenfalls aus Symmetriegründen $\int_{\triangle} B_0(\vec{x}) B_i(\vec{x})\, d\vec{x}$ den selben Wert für alle $1 \le i \le n$ besitzt, genügt es noch $\int_{\triangle} B_0(\vec{x}) B_2(\vec{x})\, d\vec{x} = \int_{\triangle} B_1(\vec{x}) B_2(\vec{x})\, d\vec{x}$ zu zeigen, was aus

$$\int_{\triangle} B_0(\vec{x}) B_2(\vec{x})\, d\vec{x} = \int_{x_n=0}^{1} \cdots \int_{x_2=0}^{1-\sum_{k=3}^{n} x_k} \int_{x_1=0}^{1-\sum_{k=2}^{n} x_k} \left(1 - \sum_{k=1}^{n} x_k \right) x_2\, dx_1 dx_2 \ldots dx_n$$

und mit der linearen Substitution $\tilde{x}_1 = 1 - \sum_{k=1}^{n} x_k$ folgt

$$
\begin{aligned}
&= \int_{x_n=0}^{1} \dots \int_{x_2=0}^{1-\sum_{k=3}^{n} x_k} \int_{\tilde{x}_1=1-\sum_{k=2}^{n} x_k}^{0} -\tilde{x}_1 x_2 \, d\tilde{x}_1 dx_2 \dots dx_n \\
&= \int_{\triangle} B_1(\vec{x}) B_2(\vec{x}) \, d\vec{x} \,. \qquad (3.58)
\end{aligned}
$$

□

Analog verläuft auch der nächste Beweis zum Berechnen der letzten fehlenden Werte aus (3.48).

Lemma 3.16 *Für $0 \le i \le n$ gilt*

$$\int_{\triangle} B_i(\vec{x})^2 \, d\vec{x} = \frac{2}{(n+2)!} \,. \qquad (3.59)$$

Beweis: Wiederum hat $\int_{\triangle} B_i(\vec{x})^2 \, d\vec{x}$ für alle $1 \le i \le n$ denselben Wert, und mit der Transformation (3.51) erhält man

$$
\begin{aligned}
\int_{\triangle} B_1(\vec{x})^2 \, d\vec{x} = \int_{\triangle} x_1^2 \, d\vec{x} &= \int_{x_n=0}^{1} \dots \int_{x_2=0}^{1-\sum_{k=3}^{n} x_k} \int_{x_1=0}^{1-\sum_{k=2}^{n} x_k} x_1^2 \, dx_1 dx_2 \dots dx_n \\
&= \int_{y_n=0}^{1} \dots \int_{y_1=0}^{1} y_1^2 (1-y_1)^{n-1} \cdots (1-y_{n-1}) \, dy_1 \dots dy_n \\
&= \left(\frac{2}{(n+2)(n+1)n} \right) \frac{1}{n-1} \cdots \frac{1}{2} = \frac{2}{(n+2)!} \,, \qquad (3.60)
\end{aligned}
$$

wenn man berücksichtigt, daß mit partieller Integration

$$\begin{aligned}\int_0^1 t^2(1-t)^m\,dt &= \left[-\frac{t^2}{m+1}(1-t)^{m+1}\right]_0^1 + \int_0^1 \frac{2t}{m+1}(1-t)^m\,dt \\ &= \frac{2}{m+1}\left(\left[-\frac{t}{m+2}(1-t)^{m+2}\right]_0^1 + \int_0^1 \frac{1}{m+2}(1-t)^{m+2}\,dt\right) \\ &= \frac{2}{(m+1)(m+2)}\left[-\frac{1}{m+3}(1-t)^{m+3}\right]_0^1 \\ &= \frac{2}{(m+1)(m+2)(m+3)} \qquad (3.61)\end{aligned}$$

gilt. Damit bleibt nur noch $\int_\triangle B_0(\vec{x})^2\,d\vec{x} = \int_\triangle B_1(\vec{x})^2\,d\vec{x}$ zu zeigen, was aus

$$\int_\triangle B_0(\vec{x})^2\,d\vec{x} = \int_{x_n=0}^{1} \dots \int_{x_2=0}^{1-\sum_{k=3}^{n} x_k} \int_{x_1=0}^{1-\sum_{k=2}^{n} x_k} \left(1-\sum_{k=1}^{n} x_k\right)^2 dx_1 dx_2 \dots dx_n$$

und wie oben mit der linearen Substitution $\tilde{x}_1 = 1 - \sum_{k=1}^{n} x_k$ folgt

$$\begin{aligned}&= \int_{x_n=0}^{1} \dots \int_{x_2=0}^{1-\sum_{k=3}^{n} x_k} \int_{\tilde{x}_1=1-\sum_{k=2}^{n} x_k}^{0} -\tilde{x}_1^2\,d\tilde{x}_1 dx_2 \dots dx_n \\ &= \int_\triangle B_1(\vec{x})^2\,d\vec{x}\,. \qquad (3.62)\end{aligned}$$

□

Mit Hilfe dieser drei Lemmata kann man nun (3.48) zu

$$w_0\,\frac{1}{(n+2)!} + w_{i,j}\,\frac{1}{n!} = 0 \qquad (3.63)$$

$$w_0\,\frac{2}{(n+2)!} + w_{i,i}\,\frac{1}{n!} = 1 \qquad (3.64)$$

$$w_0\,\frac{1}{(n+2)!} - \sum_{k=1}^{n} w_{i,k}\,\frac{1}{n!} = 0 \qquad (3.65)$$

$$w_0\,\frac{2}{(n+2)!} + \sum_{k,l=1}^{n} w_{k,l}\,\frac{1}{n!} = 1 \qquad (3.66)$$

für $1 \leq i < j \leq n$ umschreiben.

Anhand der Gleichungen (3.63) erhält man

$$w_{i,j} = -\frac{1}{(n+1)(n+2)} w_0 \,, \tag{3.67}$$

also haben für $i \neq j$ alle Gewichte $w_{i,j}$ den gleichen Wert. Die Gleichungen (3.64) liefern entsprechend

$$w_{i,i} = n! - \frac{2}{(n+1)(n+2)} w_0 \,, \tag{3.68}$$

d. h. auch alle Gewichte $w_{i,i}$ haben den gleichen Wert. Einsetzen dieser Beziehungen in (3.66) liefert

$$w_0 = (n+1)! \tag{3.69}$$

und es läßt sich leicht nachrechnen, daß auch die Gleichungen (3.65) damit erfüllt sind. Dieses Ergebnis läßt sich in dem folgenden Satz zusammenfassen.

Satz 3.17 *Das lineare Gleichungssystem (3.48) hat die eindeutige Lösung*

$$w_0 = (n+1)! \tag{3.70}$$

$$w_{i,i} = \frac{n \cdot n!}{n+2}, \quad (1 \leq i \leq n) \tag{3.71}$$

$$w_{i,j} = -\frac{n!}{n+2}, \quad (1 \leq i, j \leq n, i \neq j) \,. \tag{3.72}$$

Insbesondere erhält man für die ersten Werte von n die Gewichte

$$w_0 = 6, \qquad W = \begin{pmatrix} 1 & -1/2 \\ -1/2 & 1 \end{pmatrix} \tag{3.73}$$

für $n = 2$, sowie für $n = 3$

$$w_0 = 24, \qquad W = \begin{pmatrix} 18/5 & -6/5 & -6/5 \\ -6/5 & 18/5 & -6/5 \\ -6/5 & -6/5 & 18/5 \end{pmatrix} \tag{3.74}$$

und schließlich für $n = 4$

$$w_0 = 120, \qquad W = \begin{pmatrix} 16 & -4 & -4 & -4 \\ -4 & 16 & -4 & -4 \\ -4 & -4 & 16 & -4 \\ -4 & -4 & -4 & 16 \end{pmatrix} . \tag{3.75}$$

Mit Satz 3.17 läßt sich leicht nachrechnen, daß die Matrix W im n-Dimensionalen die Eigenpaare

$$(\lambda_1, \vec{e}_1) = \left(\frac{n!}{n+2}, \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix} \right) \tag{3.76}$$

und

$$(\lambda_j, \vec{e}_j) = \left(\frac{(n+1)!}{n+2}, \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \\ -1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \right), \quad (j = 2, \ldots, n) \tag{3.77}$$

besitzt und damit positiv definit ist. Das heißt, daß $\langle \cdot, \cdot \rangle^o_\triangle$ wirklich ein Skalarprodukt auf dem Einheitssimplex $\triangle$ darstellt und durch Summation über alle Simplices auch $\langle \cdot, \cdot \rangle^o$ auf gesamt Ω. Damit erhält man die Abschätzung

$$\sqrt{\frac{n!}{n+2}} \, \|f\|_{1,2,\triangle} \leq \|f\|^o_\triangle \leq \sqrt{(n+1)!} \, \|f\|_{1,2,\triangle} \tag{3.78}$$

auf $\triangle$ bzw. mit geeigneten positiven Konstanten C_1 und C_2 auch

$$C_1 \, \|f\|_{1,2} \leq \|f\|^o \leq C_2 \, \|f\|_{1,2} \tag{3.79}$$

auf Ω, wenn $\|\cdot\|^o$ die durch $\langle \cdot, \cdot \rangle^o$ induzierte Norm und $\|\cdot\|^1_2$ die durch (3.25) induzierte Standard-Norm im Sobolev-Raum $H^{1,2}(\Omega)$ beschreibt.

Schließlich kann man in (3.32) sehr leicht nachrechnen, daß stückweise lineare Funktionen f, da sie sich als Linearkombinationen der B_i schreiben lassen, exakt approximiert werden. Damit läßt sich wie in [Rei97] die Approximationsordnung des Quasi-Interpolationsverfahrens mit Orthogonalen B-Splines bestimmen.

Satz 3.18 (Approximationsordnung) *Das Approximationsverfahren Q, das in (3.32) definiert wird, hat die Ordnung $O(h^2)$, wenn h die maximale Kantenlänge eines Simplex in $\Omega \subset \mathbb{R}^n$ ist.*

Der Beweis wird nur auf dem mit h skalierten Einheitssimplex $h\triangle$ geführt.

Beweis: Man betrachtet zunächst die Taylorentwicklung

$$f(x) = f(0) + (\nabla f(0))^{\mathrm{t}} \vec{x} + \frac{1}{2} \vec{x}^{\mathrm{t}} H f(0) \vec{x} + O(\|x\|^3_\infty) \tag{3.80}$$

einer glatten Funktion f um den Ursprung für $\|x\|_\infty \leq h/\sqrt{n}$, wobei Hf die Hesse-Matrix von f bezeichnet. Damit erhält man für das lineare Taylorpolynom p zu f im Ursprung die Abschätzungen

$$\|f - p\|_\infty \leq C_4 \, h^2 |f|_{2,\infty} \tag{3.81}$$

und

$$\|\partial_j(f-p)\|_\infty \le C_4\, h |f|_{2,\infty} \tag{3.82}$$

mit einer geeigneten Konstante C_4, wenn $|\cdot|_{k,\infty}$ die Halbnorm

$$|f|_{k,\infty} = \max_{|\alpha|=k} \|\partial^\alpha f\|_\infty \text{ mit Multiindex } \partial^\alpha = \prod_\nu \partial_\nu^{\alpha_\nu} \tag{3.83}$$

bezeichnet. Da p ebenfalls eine stückweise lineare Funktion ist, gilt $Qp = p$. Daraus folgt

$$\|f - Qf\|_\infty = \|(f-p) - Q(f-p)\|_\infty \le C_4\, h^2 |f|_{2,\infty} + \|Q(f-p)\|_\infty \tag{3.84}$$

und der letzte Term läßt sich nach (3.32) durch

$$\begin{aligned} \|Q(f-p)\|_\infty &= \left\| \sum_{i\in I} \frac{\langle f-p, B_i\rangle^o_{h\triangle}}{\langle B_i, B_i\rangle^o_{h\triangle}} B_i \right\|_\infty \\ &\le \sup_{i\in I} |\langle f-p, B_i\rangle^o_{h\triangle}| \end{aligned} \tag{3.85}$$

aufgrund der Konvexen-Hülle-Eigenschaft und der Orthogonalität der B_i auf $h\triangle$ abschätzen. Mit

$$\langle f-p, B_i\rangle^o_{h\triangle} = \hat{w}_0 \int_{h\triangle} (f-p) B_i \, d\vec{x} + \sum_{j,k=1}^{n} \hat{w}_{j,k} \int_{h\triangle} \partial_j (f-p) \partial_k B_i \, d\vec{x} \tag{3.86}$$

und $A = hE$ in (3.41) und (3.42) erhält man

$$\begin{aligned} \|Q(f-p)\|_\infty \le \sup_{i\in I} \Bigg| & w_0 h^{-n} \|f-p\|_\infty \int_{h\triangle} B_i \, d\vec{x} \\ & + \sum_{j,k=1}^{n} w_{j,k} h^{n-2} \|\partial_j(f-p)\|_\infty \int_{h\triangle} \partial_k B_i \, d\vec{x} \Bigg| . \end{aligned} \tag{3.87}$$

Da die linearen Hutfunktionen auf $h\triangle$ die Form

$$B_0 = 1 - \sum_{j=1}^{n} x_j/h, \quad B_i = x_i/h \text{ für } 1 \le i \le n \tag{3.88}$$

mit den Ableitungen

$$\partial_k B_0 = -1/h, \quad \partial_k B_i = \delta_{i,k}/h \text{ für } 1 \le i \le n \tag{3.89}$$

haben, folgt weiter

$$\|Q(f-p)\|_\infty \le C_5 \|f-p\|_\infty + C_6\, h \|\partial_j(f-p)\|_\infty \tag{3.90}$$

mit geeigneten Konstanten C_5 und C_6 und die Gleichungen (3.81) und (3.82) liefern schließlich die Approximationsordnung. □

Es ist weiterhin erwähnenswert, daß zusätzlich zur Orthogonalität der B-Splines bzgl. $\langle\cdot,\cdot\rangle^o$, die die schnelle Berechnung der Koeffizienten in (3.32) ermöglicht, dieses Skalarprodukt nicht nur die Differenz der Funktionswerte berücksichtigt, sondern auch die Abweichung der Normalen und damit insbesondere für Berechnungen mit Reflektoren geeignet ist.

3.7 Diskrete Approximation des Skalarprodukts

Betrachtet man für die Anwendung wieder das Skalarprodukt $\langle\cdot,\cdot\rangle^o_\triangle$ im Zweidimensionalen, so stellt man fest, daß zur Berechnung der Koeffizienten aus (3.32) die auftretenden Integrale

$$\langle h, B_i\rangle^o = w_0 \int_\Omega h(x,y)B_i(x,y)\,d(x,y) + \int_\Omega \nabla h^{\mathrm{t}}(x,y)W\nabla B_i(x,y) \tag{3.91}$$

nur schwer exakt zu lösen sind. Deshalb wird nun $\langle f, g\rangle^o_\triangle$ durch eine gewichtete Summe von Funktionswerten approximiert. Die Funktion wird dabei an den Ecken sowie den Kantenmittelpunkten eines Dreiecks ausgewertet.

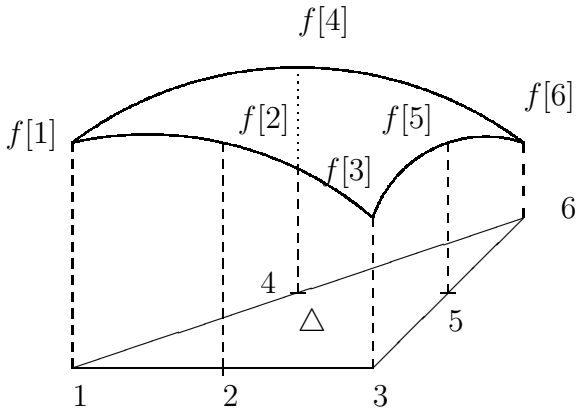


Abbildung 3.8: Auswertung von f an sechs Punkten über einem Dreieck

Das Ziel ist dabei eine 6×6-Matrix V mit

$$\langle f, g\rangle^o_\triangle \approx f[1,\ldots,6]^{\mathrm{t}} V g[1,\ldots,6]\,. \tag{3.92}$$

Da sechs Funktionswerte zur Verfügung stehen, kann man sogar Gleichheit erreichen, falls f und g polynomiale Funktionen vom Grad zwei sind. Setzt man der Reihe nach für f

und g die Polynome $1-x-y, x, y, x^2, y^2, (1-x-y)^2$ an, die eine Basis für den Raum der Polynome vom Grad zwei bilden, und verlangt Gleichheit in Gleichung (3.92), so erhält man ein LGS mit der Lösung

$$V = \begin{pmatrix} \frac{3}{5} & -\frac{1}{3} & \frac{1}{15} & -\frac{1}{3} & -\frac{1}{15} & \frac{1}{15} \\ -\frac{1}{3} & \frac{38}{15} & -\frac{1}{3} & -\frac{2}{5} & -\frac{2}{5} & -\frac{1}{15} \\ \frac{1}{15} & -\frac{1}{3} & \frac{3}{5} & -\frac{1}{15} & -\frac{1}{3} & \frac{1}{15} \\ -\frac{1}{3} & -\frac{2}{5} & -\frac{1}{15} & \frac{38}{15} & -\frac{2}{5} & -\frac{1}{3} \\ -\frac{1}{15} & -\frac{2}{5} & -\frac{1}{3} & -\frac{2}{5} & \frac{38}{15} & -\frac{1}{3} \\ \frac{1}{15} & -\frac{1}{15} & \frac{1}{15} & -\frac{1}{3} & -\frac{1}{3} & \frac{3}{5} \end{pmatrix}. \tag{3.93}$$

Zu beachten ist, daß diese Gewichte unabhängig von der Geometrie des einzelnen Dreiecks sind.

Da zur Berechnung der Gewichte γ_i aus (3.32) immer nur die Skalarprodukte mit den Basis-Hutfunktionen

$$B_0 = 1-x-y, \quad B_1 = x, \quad B_2 = y \tag{3.94}$$

benötigt werden, kann man VB_i bereits vorab berechnen. An den obigen Auswertungstellen gilt für diese Hutfunktionen

$$\begin{aligned} B_0[1,\ldots,6] &= \begin{pmatrix} 1 & 1/2 & 0 & 1/2 & 0 & 0 \end{pmatrix}^{\mathrm{t}} \\ B_1[1,\ldots,6] &= \begin{pmatrix} 0 & 1/2 & 1 & 0 & 1/2 & 0 \end{pmatrix}^{\mathrm{t}} \\ B_2[1,\ldots,6] &= \begin{pmatrix} 0 & 0 & 0 & 1/2 & 1/2 & 1 \end{pmatrix}^{\mathrm{t}} \end{aligned} \tag{3.95}$$

und damit

$$\begin{aligned} VB_0[1,\ldots,6] &= 1/15 \begin{pmatrix} 4 & 11 & -2 & 11 & -7 & -2 \end{pmatrix}^{\mathrm{t}} \\ VB_1[1,\ldots,6] &= 1/15 \begin{pmatrix} -2 & 11 & 4 & -7 & 11 & -2 \end{pmatrix}^{\mathrm{t}} \\ VB_2[1,\ldots,6] &= 1/15 \begin{pmatrix} -2 & -7 & -2 & 11 & 11 & 4 \end{pmatrix}^{\mathrm{t}}. \end{aligned} \tag{3.96}$$

Die lineare Standard-Interpolation, die lediglich die Funktionswerte an den Eckpunkten der Dreiecke exakt wiedergibt, verwendet zum Vergleich

$$\begin{aligned} &\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}^{\mathrm{t}} \\ &\begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix}^{\mathrm{t}} \\ &\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}^{\mathrm{t}}. \end{aligned} \tag{3.97}$$

Mit Hilfe dieser Werte ist es nun möglich, die Gewichte γ_i aus (3.32) durch einfache Vektor-Vektor-Multiplikation schnell approximativ zu berechnen.

Mit dem bisher vorgestellten Verfahren ist es also möglich, den Graph einer gegeben n-dimensionalen, skalaren Funktion schnell durch eine Triangulierung zu approximieren.

Dabei berücksichtigt das verwendete Skalarprodukt $\langle \cdot, \cdot \rangle^o$ wie erwähnt nicht nur die Differenz der Funktionswerte, sondern auch die Abweichung der Normalen, was insbesondere bei Reflektorflächen, wie sie bei diesem Projekt verwendet werden, von Bedeutung ist.

Um dieses Verfahren in CALMIR2 verwenden zu können, muß noch beachtet werden, daß die verwendeten Reflektorflächen jedoch nicht durch skalare Tensorprodukt-NURBS-Funktionen sondern durch NURBS-Flächen dargestellt werden, die in jeder Koordinate eine Tensorprodukt-NURBS-Funktion enthalten. Um auch diese Flächen zu approximieren, wird das vorgestellte Verfahren einfach für jede Koordinate getrennt angewendet.

Vergleicht man schließlich noch den Aufwand dieses Verfahrens mit der linearen Standard-Interpolation, so erkennt man, daß das CALMIR2-Verfahren bei einer regulären Triangulierung im Durchschnitt zwei Funktionsauswertungen pro Dreieck benötigt, während die linerare Interpolation im Durchschnitt 1.5 Funktionsauswertungen pro Dreieck benötigt, wenn zusätzlich der Fehler in der Mitte des Dreiecks berechnet wird, wie es für die in diesem Programm verwendeten adaptiven Unterteilungen notwendig ist.

Auch bei dieser diskreten Approximation läßt sich wieder nachrechnen, daß – wie in Abschnitt 3.6 – stückweise lineare Funktionen exakt approximiert werden. D. h. auch bei der diskreten Approximation läßt sich zeigen, daß sie die Ordnung $O(h^2)$ besitzt, wenn wieder h die maximale Kantenlänge eines Dreiecks bezeichnet.

3.8 Adaptive Triangulierung

Geht man von einem rechteckigen Definitionsgebiet $\Omega = [x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}]$ aus, über dem eine Reflektorfläche als Graph einer NURBS-Funktion gegeben ist, so wird zunächst die reguläre Initial-Triangulierung wie in untenstehender Abbildung berechnet, wobei die Abschnitte in x- und y-Richtung vom Benutzer festgelegt werden.

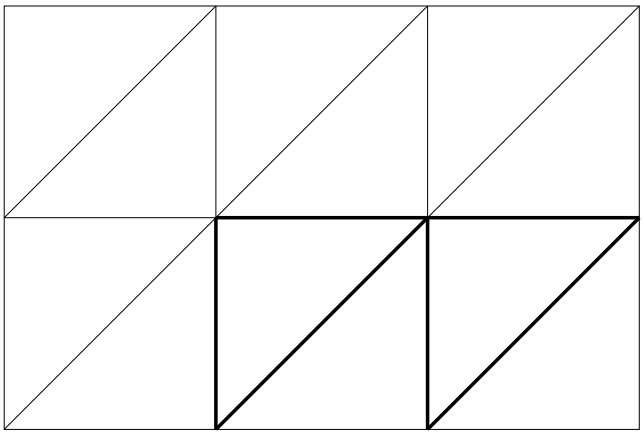


Abbildung 3.9: Initial-Triangulierung und zu unterteilende Dreiecke

Danach wird die zugehörige approximative Triangulierung des NURBS-Graphen wie im vorigen Abschnitt beschrieben berechnet. Sollte die parametrische Differenz zwischen der Triangulierung und dem Funktionswert an einer der Stützstellen größer als eine vom Benutzer vorgegebene Fehlergrenze sein, so werden die entsprechenden Dreiecke im Definitionsgebiet lokal weiter verfeinert. Um dabei hängende Knoten in der Triangulierung zu verhindern, gibt es mehrere Verfahren, wobei sich zwei in der Praxis durchgesetzt haben.

Das erste ist die sogenannte „longest edge bisection“ von Rivara [Riv84]. Bei diesem Verfahren werden Dreiecke mit zu schlechter Approximation durch Einfügen einer neuen Strecke zwischen dem Mittelpunkt der längsten Kante und dem gegenüberliegenden Eckpunkt halbiert. Solange noch Dreiecke mit hängenden Knoten vorhanden sind, werden diese durch Bisektion der längsten Kante ebenfalls unterteilt. Man kann zeigen, daß dieser Algorithmus nach endlich vielen Schritten terminiert. Durch Unterteilung der längsten Kante ist bei diesem Verfahren außerdem gewährleistet, daß die Winkel eines Dreieckes nicht beliebig klein werden [RS75].

Das zweite Verfahren ist die sogenannte „rot-grün-Verfeinerung“, die auf Bank, Sherman und Weiser zurückgeht [BSW83]. Dieser Algorithmus läuft dabei wie folgt ab.

- Alle Dreiecke mit zu schlechter Approximationsgüte werden regulär in vier ähnliche Dreiecke unterteilt („rote Verfeinerung“).

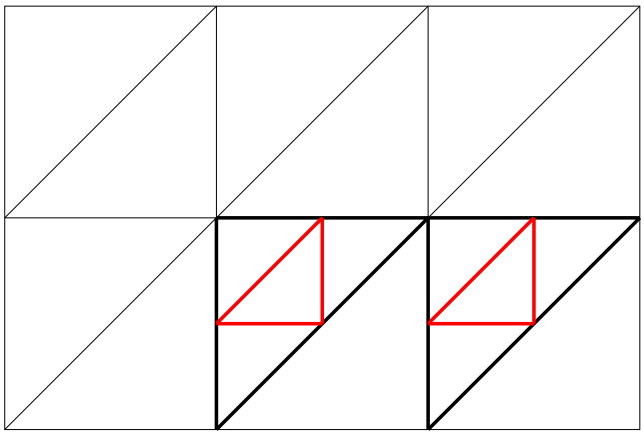


Abbildung 3.10: Rote Verfeinerung

- Solange noch Dreiecke mit zwei oder mehr hängenden Knoten vorhanden sind, werden diese ebenfalls regulär unterteilt.

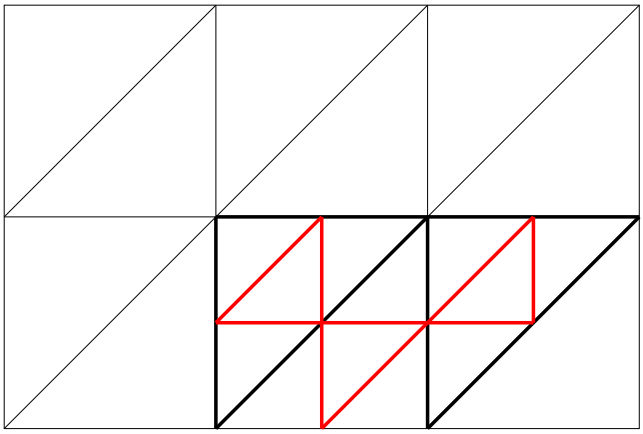


Abbildung 3.11: Weitere Dreiecke mit 2 hängenden Knoten werden unterteilt

- Alle Dreiecke, die nur einen hängenden Knoten haben, werden an diesem Knoten und dem gegenüberliegenden Eckpunkt durch eine neue Kante halbiert („grüne Verfeinerung“).

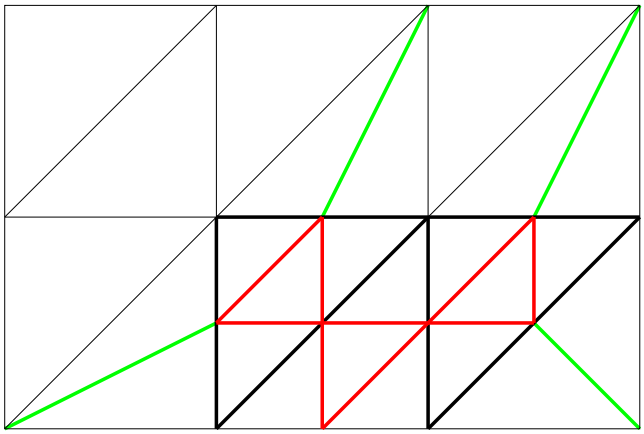


Abbildung 3.12: Grüne Verfeinerung

Es ist leicht zu sehen, daß die Kantenverhältnisse durch $2\sqrt{2}$ beschränkt sind, und damit auch die Innenwinkel eines Dreiecks nicht beliebig klein werden.

Für dieses Projekt wurde das letztgenannte Verfahren implementiert, wobei das Verfahren iteriert wird bis die gewünschte parametrische Genauigkeit erreicht wird.

3.9 Raytracer

Der letzte Baustein zur Bewertung einer gegebenen Reflektorkonfiguration ist der Raytracer in CALMIR2. Dieser verfolgt einen von der Lichtquelle ausgehenden Lichtstrahl –

wie in Abschnitt 2.1 beschrieben – über eventuelle Reflexionen an den Reflektorflächen bis zum Verlassen der Leuchte. Der Raytracer verwendet dabei den sogenannten *Bricklayer*-Algorithmus, wie er in [Gre99] beschrieben ist.

Dieser Algorithmus basiert auf einer gleichmäßigen Unterteilung des Anfangs-Quaders Q aus (2.27) in kleinere Quader (*bricks*, zu deutsch *Ziegelsteine*) und einem einfachen Mechanismus, der garantiert, daß leere Quader schnell durchlaufen werden. Während der Initialisierung des Algorithmus wird die maximale Ausdehnung der Dreiecke, die den Reflektor approximieren, in x-, y- und z-Richtung bestimmt, und Q in $n_x \times n_y \times n_z$ kleinere Quader gleichmäßig unterteilt, wobei der Benutzer die Anzahl der Unterteilungen in die einzelnen Koordinatenrichtungen vorgibt. Anschließend wird für jeden Quader eine Liste der Dreiecke erstellt, die mit ihm einen nichtleeren Schnitt haben. Um nun einen Lichtstrahl zu verfolgen, wird zunächst der Quader bestimmt, in dem der Startpunkt des Strahls liegt. Danach werden wie unten beschrieben die Quader bestimmt, durch die der Strahl verläuft. Dieser Prozess wird gestoppt, wenn der Strahl auf ein Dreieck trifft oder Q verläßt.

3.9.1 Initialisierung

Sei n die Anzahl der Dreiecke in der Triangulierung, wobei ein Dreieck

$$\triangle_i = \triangle(P_{i,1}; P_{i,2}; P_{i,3}), \quad (1 \leq i, \leq n) \tag{3.98}$$

die Eckpunkte mit den Koordinaten

$$P_{i,j} = (x_{i,j}, y_{i,j}, z_{i,j}), \quad (1 \leq j \leq 3) \tag{3.99}$$

besitzt. Um die räumliche Ausdehnung der Dreiecke zu bestimmen werden zunächst die Werte

$$\begin{aligned} x_{i,\min} &= \min\{x_{i,j} : 1 \leq j \leq 3\}, & x_{i,\max} &= \max\{x_{i,j} : 1 \leq j \leq 3\}, \\ y_{i,\min} &= \min\{y_{i,j} : 1 \leq j \leq 3\}, & y_{i,\max} &= \max\{y_{i,j} : 1 \leq j \leq 3\}, \\ z_{i,\min} &= \min\{z_{i,j} : 1 \leq j \leq 3\}, & z_{i,\max} &= \max\{z_{i,j} : 1 \leq j \leq 3\}, \end{aligned} \tag{3.100}$$

für $1 \leq i \leq n$, sowie

$$\begin{aligned} x_{\min} &= \min\{x_{i,\min} : 1 \leq i \leq n\}, & x_{\max} &= \max\{x_{i,\max} : 1 \leq i \leq n\}, \\ y_{\min} &= \min\{y_{i,\min} : 1 \leq i \leq n\}, & y_{\max} &= \max\{y_{i,\max} : 1 \leq i \leq n\}, \\ z_{\min} &= \min\{z_{i,\min} : 1 \leq i \leq n\}, & z_{\max} &= \max\{z_{i,\max} : 1 \leq i \leq n\}, \end{aligned} \tag{3.101}$$

berechnet. Mit den Anzahlen n_x, n_y und n_z der Quader erhält man die Ausdehnung jedes Quaders

$$\Delta x = \frac{x_{\max} - x_{\min}}{n_x}, \quad \Delta y = \frac{y_{\max} - y_{\min}}{n_y}, \quad \Delta z = \frac{z_{\max} - z_{\min}}{n_z} \tag{3.102}$$

in die entsprechende Koordinatenrichtung. Ein einzelner Quader $Q_{k,l,m}$ ist dann durch

$$\begin{aligned} Q_{k,l,m} = \{(x,y,z) :& (k-1)\Delta x \le x - x_{\min} \le k\Delta x, \\ & (l-1)\Delta y \le y - y_{\min} \le y\Delta y, \\ & (m-1)\Delta z \le z - z_{\min} \le z\Delta z\} \end{aligned} \tag{3.103}$$

für $1 \le k \le n_x, 1 \le l \le n_y, 1 \le m \le n_z$ gegeben.

Anschließend wird für jeden Quader $Q_{k,l,m}$ die Menge $T_{k,l,m}$ von Dreiecken bestimmt, die den Quader schneiden. Dabei wird ein einfacheres Kriterium verwendet, weil es nicht schadet, wenn zu viele Dreiecke in der Menge $T_{k,l,m}$ auftauchen. Es ist klar, daß ein Dreieck $\triangle_j$ den Quader $Q_{k,l,m}$ nicht schneidet, wenn eine der folgenden Bedingungen

$$\begin{array}{llll} x_{i,\max} \le (k-1)\Delta x + x_{\min} & \text{oder} & x_{i,\min} \ge k\Delta x + x_{\min} & \text{oder} \\ y_{i,\max} \le (l-1)\Delta y + y_{\min} & \text{oder} & y_{i,\min} \ge l\Delta y + y_{\min} & \text{oder} \\ z_{i,\max} \le (m-1)\Delta z + z_{\min} & \text{oder} & z_{i,\min} \ge m\Delta z + z_{\min} & \end{array} \tag{3.104}$$

gilt. Alle Dreiecke, die keine der obigen Bedingungen erfüllen, werden in die Menge $T_{k,l,m}$ übernommen.Natürlich kann man hier auch andere Tests anwenden, die ein besseres Ergebnis liefern, aber dafür auch aufwendiger sind.

3.9.2 Der Raytracing-Algorithmus

Es wird ein Lichtstrahl verfolgt, der den Startpunkt (x_0, y_0, z_0) und die normierte Richtung (r_x, r_y, r_z) hat. Dies kann entweder ein an der Lichtquelle neu generierter oder ein reflektierter Strahl sein. Als erstes wird der Quader $Q_{k,l,m}$ bestimmt, in dem der Startpunkt des Strahls liegt. Dazu werden die Indizes

$$k = \left\lceil \frac{x_0 - x_{\min}}{\Delta x} \right\rceil + 1, \quad l = \left\lceil \frac{y_0 - y_{\min}}{\Delta y} \right\rceil + 1, \quad m = \left\lceil \frac{z_0 - z_{\min}}{\Delta z} \right\rceil + 1 \tag{3.105}$$

berechnet, wobei $\lceil x \rceil$ die kleinste ganze Zahl angibt, die kleiner oder gleich x ist, und an der jeweils oberen Grenze, d. h. zum Beispiel für $k = n_x + 1$ zu $k = n_x$ korrigiert wird.

Als nächstes wird der lokale Versatz innerhalb eines Quaders

$$t_x = \begin{cases} ((k-1)\Delta x + x_{\min} - x_0)/r_x \\ \infty \\ (k\Delta x + x_{\min} - x_0)/r_x \end{cases} \quad \Delta t_x = \begin{cases} -\Delta x/r_x, & \text{falls } r_x < 0 \\ \infty, & \text{falls } r_x = 0 \\ \Delta x/r_x, & \text{falls } r_x > 0 \end{cases} \tag{3.106}$$

$$t_y = \begin{cases} ((l-1)\Delta y + y_{\min} - y_0)/r_y \\ \infty \\ (l\Delta y + y_{\min} - y_0)/r_y \end{cases} \quad \Delta t_y = \begin{cases} -\Delta y/r_y, & \text{falls } r_y < 0 \\ \infty, & \text{falls } r_y = 0 \\ \Delta y/r_y, & \text{falls } r_y > 0 \end{cases} \tag{3.107}$$

$$t_z = \begin{cases} ((m-1)\Delta z + z_{\min} - z_0)/r_z \\ \infty \\ (m\Delta z + z_{\min} - z_0)/r_z \end{cases} \quad \Delta t_z = \begin{cases} -\Delta z/r_z, & \text{falls } r_z < 0 \\ \infty, & \text{falls } r_z = 0 \\ \Delta z/r_z, & \text{falls } r_z > 0 \end{cases} \tag{3.108}$$

in die entsprechende Koordinatenrichtung berechnet. Für alle Dreiecksindizes $1 \leq i \leq n$ wird der Abstand zum Schnittpunkt t_i auf -1 initialisiert.

Nun werden alle Dreiecke $\triangle_i$ in $T_{k,l,m}$ auf einen Schnittpunkt mit dem Strahl untersucht, und falls vorhanden der Abstand von (x_0, y_0, z_0) zu diesem Schnittpunkt in t_i festgehalten. Falls es nun ein t_i gibt, das die Bedingung

$$0 \leq t_i \leq \min\{t_x, t_y, t_z\} \tag{3.109}$$

erfüllt, wird das kleinste von diesen ausgewählt. Der zugehörige Index i liefert das Dreieck mit dem ein Schnittpunkt gefunden wurde und der Algorithmus wird mit dem reflektierten Strahl von Beginn dieses Abschnitts an fortgesetzt.

Falls kein Schnittpunkt gefunden wurde, wird zu einem benachbarten Quader gewechselt, wobei der neue Index mittels

$$\begin{aligned}
&\text{falls } t_x \leq t_y \text{ und } t_x \leq t_z \text{ und } r_x < 0 : k \leftarrow k-1, t_x \leftarrow t_x + \Delta t_x \\
&\text{falls } t_x \leq t_y \text{ und } t_x \leq t_z \text{ und } r_x > 0 : k \leftarrow k+1, t_x \leftarrow t_x + \Delta t_x \\
&\text{falls } t_y < t_x \text{ und } t_y \leq t_z \text{ und } r_y < 0 : l \leftarrow l-1, t_y \leftarrow t_y + \Delta t_y \\
&\text{falls } t_y < t_x \text{ und } t_y \leq t_z \text{ und } r_y > 0 : l \leftarrow l+1, t_y \leftarrow t_y + \Delta t_y \\
&\text{falls } t_z < t_x \text{ und } t_z < t_y \text{ und } r_z < 0 : m \leftarrow m-1, t_z \leftarrow t_z + \Delta t_z \\
&\text{falls } t_z < t_x \text{ und } t_z < t_y \text{ und } r_z > 0 : m \leftarrow m+1, t_z \leftarrow t_z + \Delta t_z
\end{aligned} \tag{3.110}$$

berechnet wird. Falls nun ein Index außerhalb des zugehörigen Bereichs ist, d. h. falls z. B. k außerhalb von $1, \ldots, n_x$ liegt, wird der Algorithmus beendet und es wurde keine weitere Reflexion gefunden. Ansonsten wird wieder an der Stelle eingestiegen, an der alle Dreiecke des Quaders auf einen Schnittpunkt mit dem Strahl untersucht werden.

Das bemerkenswerte an diesem Algorithmus ist, daß zum Durchlaufen eines leeren Quaders nur bis zu fünf Vergleiche und zwei Additionen ausgeführt werden müssen. Dadurch kann ein Lichstrahl über weite Strecken ohne Reflektionen sehr schnell verfolgt werden.

4 Optimierung

Um von einer gegebenen Reflektorkonfiguration zu einer noch besseren zu gelangen, war bisher der Ansatz der Lichttechnikingenieure bei Philips einzelne Kontrollpunkte der NURBS-Flächen von Hand zu verschieben, und die modifizierte Fläche wieder mit einem Raytracerlauf neu zu bewerten. Die ursprüngliche Idee war deshalb, diesen Ablauf mit Hilfe von CALMIR2 einfach zu automatisieren.

Es hat sich jedoch schnell herausgestellt, daß das Verschieben einzelner Kontrollpunkte oft zu unerwünschten Ausbuchtungen (sogenannter Artefakte) der NURBS-Flächen führt - insbesondere weil die originalen Reflektorflächen oft mit einem sehr hohen Spline-Grad konstruiert waren. Eine Approximierung der Fläche mit niedrigerem Grad konnte diesen Effekt zwar abschwächen, aber nicht völlig eliminieren.

Gesucht war deshalb ein Verfahren zur „glatten“ Änderung der Fläche, das eine überschaubare Anzahl von Freiheitsgraden aufweist, um eine Optimierung in annehmbarer Zeit zu ermöglichen. Die im Folgenden beschriebene Bézier-Raum-Deformation entspricht genau diesen Anforderungen. Sie wird im Zweidimensionalen auch in der Computer-Grafik zum sogenannten „morphen“ von Bildern verwendet.

4.1 Bézier-Raum-Deformation

Zur Vereinfachung wird im Folgenden davon ausgegangen, daß der Quader Q aus (2.27) der Einheitswürfel

$$Q = [0,1] \times [0,1] \times [0,1] \tag{4.1}$$

ist, was durch eine einfache lineare Transformation erreicht werden kann. Als Basis für die Bézier-Raum-Deformation werden die Bernstein-Polynome aus (3.9) verwendet, die eine Basis für die Spline-Funktionen auf dem Intervall $[0,1)$ mit Bézier-Rändern bilden. Die Bézier-Raum-Deformation hat dann die Form

$$(x,y,z) \mapsto B(x,y,z) = \sum_{i=0}^{n_x} \sum_{j=0}^{n_y} \sum_{k=0}^{n_z} P_{i,j,k} b_i^{n_x}(x) b_j^{n_y}(y) b_k^{n_z}(z), \quad x,y,z \in [0,1], \tag{4.2}$$

wobei n_x, n_y und n_z den Grad der Bernstein-Polynome in die entsprechende Koordinatenrichtung angeben, und $P_{i,j,k} \in \mathbb{R}^3$ die Kontrollpunkte der Deformation sind.

Geht man von

$$1 = (1 - x + x)^{n-1} = \sum_{j=0}^{n-1} \binom{n-1}{j} x^j (1-x)^{n-j-1} \tag{4.3}$$

aus, so erhält man nach Multiplikation mit x

$$x = \sum_{j=0}^{n-1} \binom{n-1}{j} x^{j+1} (1-x)^{n-j-1}$$

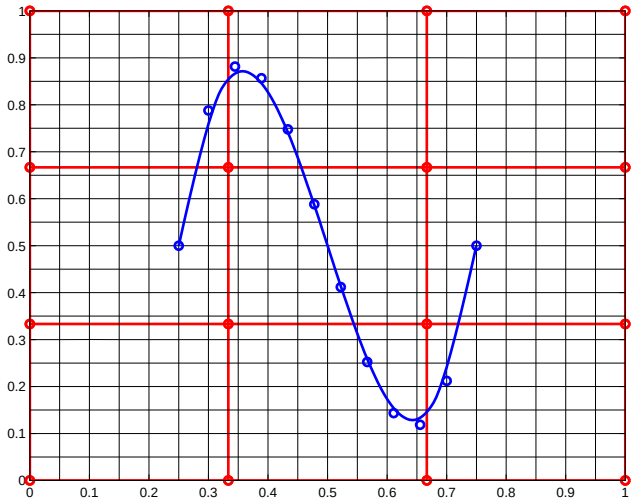


Abbildung 4.1: Initial-Transformation

und Indexverschiebung

$$
\begin{aligned}
&= \sum_{j=1}^{n} \binom{n-1}{j-1} x^j (1-x)^{n-j} \\
&= \sum_{j=1}^{n} c_j b_j^n(x)
\end{aligned}
\tag{4.4}
$$

mit

$$
c_j = \frac{j}{n} \tag{4.5}
$$

für $0 \leq j \leq n$. Für ein regelmäßiges Gitter von Kontrollpunkten

$$
P_{i,j,k} = \begin{pmatrix} i/n_x \\ j/n_y \\ k/n_z \end{pmatrix}, \begin{cases} 0 \leq i \leq n_x, \\ 0 \leq j \leq n_y, \\ 0 \leq k \leq n_z, \end{cases} \tag{4.6}
$$

erhält man also die identische Abbildung des Quaders Q auf sich selbst, und dies ist auch die Initial-Transformation, mit der die Optimierung in CALMIR2 gestartet wird. Anschaulich für den zweidimensionalen Raum wird diese Konfiguration in der nachstehenden Abbildung gezeigt. Es sind sowohl die regelmäßigen Kontrollpunkte der Bézier-Raum-Deformation als auch die Kontrollpunkte der NURBS-Kurve sowie die Kurve selbst eingezeichnet.

Während der Optimierung werden die Kontrollpunkte $P_{i,j,k} \rightarrow \tilde{P}_{i,j,k}$ innerhalb (bzw. in positiver z-Richtung auch außerhalb) des Quaders Q verschoben, wobei die Kontrollpunkte in der unteren z-Ebene in dieser Ebene verbleiben, um die Austrittsöffnung der Leuchte

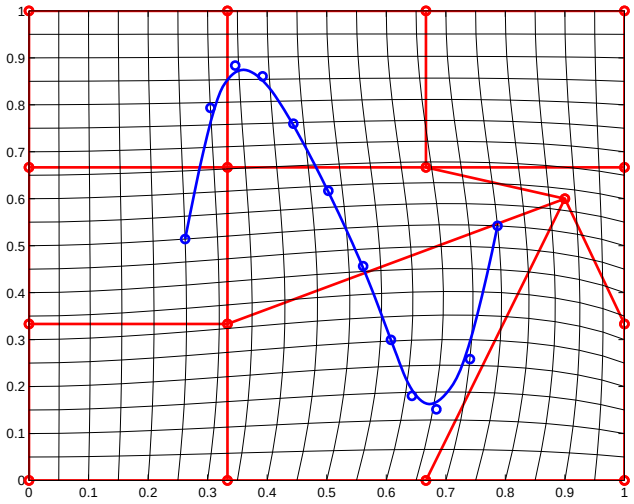


Abbildung 4.2: Bézier-Raum-Deformation

in dieser Ebene zu fixieren. Durch die Konvexe-Hülle-Eigenschaft der Bézier-Splines ist damit gewährleistet, daß sich die Kontrollpunkte der NURBS-Fläche und damit auch die Fläche selbst wieder innerhalb von Q befindet (mit Ausnahme der positiven z-Richtung), die Vorgaben aus Abschnitt 2.4 also erfüllt sind.

Die Transformation B für die verschobenen Kontrollpunkte $\tilde{P}_{i,j,k}$ wird lediglich an den Kontrollpunkten R_l der NURBS-Fläche ausgewertet,

$$\tilde{Q}_l = B(Q_l) \tag{4.7}$$

und aus diesen modifizierten Kontrollpunkten wird die modifizierte NURBS-Fläche neu konstruiert. Es ist zu beachten, daß dadurch im allgemeinen eine andere Fläche entsteht, als wenn die NURBS-Fläche direkt durch B transformiert wird. Dieses Vorgehen wird wieder für den zweidimensionalen Fall anschaulich in der nächsten Abbildung illustriert.

Die Bézier-Raum-Deformation liefert durch ihre globalen Veränderungen in Q eine „glatte“ Deformation der NURBS-Fläche. Die Auswirkungen sind jedoch in der Nähe von verschobenen Kontrollpunkten $P_{i,j,k}$ größer, womit sich auch lokale Effekte modellieren lassen.

Der große Vorteil der Bézier-Raum-Deformation gegenüber der direkten Manipulation individueller Kontrollpunkte Q_l der NURBS-Fläche liegt darin, daß die Modifikationen unabhängig von der vorgegeben Fläche sind, insbesondere unabhängig von der Anzahl der Kontrollpunkte Q_l.

In der Praxis hat sich gezeigt, daß schon mit relativ niedrigem Grad der Bernstein-Polynome sehr gute Ergebnisse erzielt werden können. So ist bereits Grad drei für die Optimierung ausreichend; Grad vier bringt nur marginal bessere Ergebnisse, benötigt dafür aber viel mehr Freiheitsgrade und demzufolge längere Programmlaufzeiten.

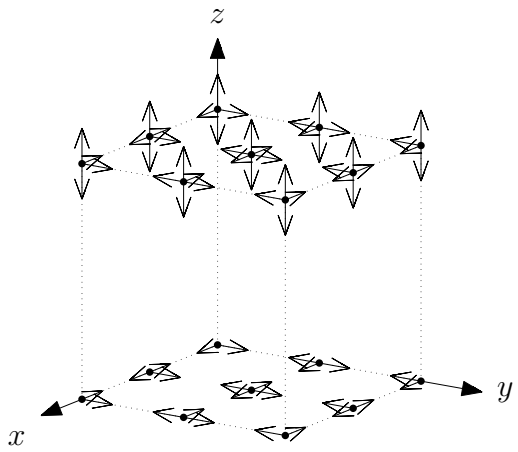


Abbildung 4.3: Freiheitsgrade für die Bézier-Raum-Deformation

In der Praxis hat sich ebenfalls gezeigt, daß für die gegebenen Reflektor-Flächen die Deformationen in z-Richtung gröber ausfallen dürfen als in die anderen Koordinatenrichtungen. In CALMIR2 wird deshalb immer $n_z = 1$ verwendet, und für die anderen Richtungen beginnend mit $n_x = n_y = 1$ der Grad schrittweise erhöht bis $n_x = n_y = 3$.

Die Anzahl der Freiheitsgrade pro Kontrollpunkt $P_{i,j,k}$ ist für den Grad $n_x = n_y = 2$ der Bernstein-Polynome exemplarisch in Abbildung 4.3 - unter Berücksichtigung der Einschränkungen aus Abschnitt 2.4 - angegeben. Die Gesamtzahl der Freiheitsgrade für die Optimierung beträgt für den Spline-Grad eins 20, für Grad zwei 45 und für Grad drei 80. Gibt man vor, daß die Reflektorfläche in x-Richtung symmetrisch modifiziert werden soll, reduziert dies die Freiheitsgrade auf 10 bei Spline-Grad eins, auf 24 bei Grad zwei und schließlich auf 40 bei Grad drei.

4.2 Optimierungs-Algorithmen

Für die Lösung von Optimierungsproblemen stehen mittlerweile eine Vielzahl von Algorithmen zur Verfügung. Die Seite [MS05] gibt einen kurzen Online-Überblick über Algorithmen und bereits implementierte Programm-Versionen und hilft, die geeigneten für das jeweilige Optimierungsproblem zu finden. Eine weitere Übersicht über Optimierungs-Algorithmen findet man z. B. in [Sch95]. Dabei lassen sich die klassischen Algorithmen grob in zwei Klassen einteilen.

Die erste Klasse sind Algorithmen, die die erste oder auch höhere Ableitungen der Zielfunktion miteinbeziehen. Typische Vertreter dieser Klasse sind das *Newton-Verfahren*, die *Methode der konjugierten Gradienten* oder Varianten davon. Während diese Verfahren meist sehr schnell und effektiv arbeiten, sind sie für dieses Projekt ungeeignet, da hier die Zielfunktion aus dem Abschnitt 2.6 nur numerisch ausgewertet wird und keine Ableitungen direkt zur Verfügung stehen.

Die zweite Klasse besteht aus direkten Suchmethoden, die ohne Ableitungen auskommen. Beispiele für diese Klasse sind der *Simplex-Algorithmus* mit Varianten wie dem *Nelder-Mead-Algorithmus*, das *Simulated Annealing*, das sogar eine globale Optimierungsmethode darstellt, oder die *Tabu-Suche*. Weitere Vertreter sind der *Hooke-Jeeves-Algorithmus*, die *Methode von Rosenbrock* und ihre Varianten oder die *Methoden von Powell*.

Bei dem hier betrachteten Optimierungsproblem handelt es sich um ein nichtlineares Problem mit Ungleichungsnebenbedingungen. Die Nebenbedingungen geben einerseits die Restriktionen durch den Begrenzungsquader aus Abschnitt 2.4 wieder. Andererseits muß man bei der Bézier-Raum-Deformation für hohe Grade beachten, daß sich die Kontrollpunkte nicht überkreuzen dürfen, damit die Modifikation des Raumes injektiv und damit die Moifikation der Fläche ohne Selbstdurchdringung bleibt.

Es ist zu beachten, daß manche der oben erwähnten Optimierungs-Verfahren nur für eine relativ kleine Anzahl von Variablen gut funktionieren, wie beispielsweise das *Simulated Annealing* oder die *Tabu-Suche*. Dieses Problem konnte jedoch durch den Einsatz der Bézier-Raum-Deformation, die eine deutlich geringere Anzahl an Freiheitsgraden als die direkte Manipulation der Kontrollpunkte der NURBS-Fläche aufweist, deutlich entschärft werden. Prinzipiell sind damit alle Algorithmen der zweiten Gruppe für die Optimierung im Rahmen dieser Arbeit geeignet. Es wurde schließlich der *Hooke-Jeeves-Algorithmus*, der sich für die hier typischerweise auftretenden 20 bis 80 Freiheitsgrade bewährt hat, als Optimierungs-Verfahren für dieses Projekt ausgewählt und implementiert.

4.3 Hooke-Jeeves-Algorithmus

Der Hooke-Jeeves-Algorithmus dient zur direkten Minimierung einer skalaren Funktion f durch Variation der n-dimensionalen Funktionsvariable x, d. h. zum Finden von

$$\min_x \{ f(x) | x \in \mathbb{R} \} . \tag{4.8}$$

Für CALMIR2 entspricht f dabei der Zielfunktion (2.33) aus 2.6, und die Variable x faßt die n frei veränderbaren Koordinaten der Kontrollpunkte aus 4.1 zusammen. Der Algorithmus, wie er z. B. in [Sch95] oder [HH71] beschrieben ist, besteht dabei aus folgenden Schritten.

- Schritt 0 (Initialisierung)
  Es werden ein Startpunkt $x^{(0,0)} = x^{(-1,n)}$, eine Genauigkeitsgrenze $\varepsilon > 0$ und die Anfangs-Schrittlängen $s_i^{(0)} \neq 0$ für $1 \leq i \leq n$ gewählt (z. B. $s_i^{(0)} = 1$ falls keine plausibleren Startwerte zur Verfügung stehen). Des weiteren wird $k = 0$ und $i = 1$ gesetzt.

- Schritt 1 (Erkundungs-Schritt)
  Es wird $\tilde{x} = x^{(k,i-1)} + s_i^{(k)} e_i$ gebildet, wobei e_i der i-te Einheitsvektor ist (Schritt in die positive Richtung).

Falls $f(\tilde{x}) < f(x^{(k,i-1)})$ wird mit Schritt 2 fortgesetzt (erfolgreicher erster Versuch);
ansonsten wird $\tilde{x} \leftarrow \tilde{x} - 2s_i^{(k)} e_i$ gesetzt (Schritt in die negative Richtung).
Falls $f(\tilde{x}) < f(x^{(k,i-1)})$ wird mit Schritt 2 fortgesetzt (Erfolg);
ansonsten wird $\tilde{x} \leftarrow \tilde{x} + s_i^{(k)} e_i$ gesetzt (Ausgangssituation).

- Schritt 2 (Übergang zur nächsten Koordinate)
  Es wird $x^{(k,i)} = \tilde{x}$ gesetzt.
  Falls $i < n$ wird $i \leftarrow i + 1$ gesetzt und mit Schritt 1 fortgesetzt.

- Schritt 3 (Test auf Fehlschlag in alle Richtungen)
  Falls $f(x^{(k,n)}) \geq f(x^{(k,0)})$, wird $x^{(k+1,0)} = x^{(k,0)}$ gesetzt und mit Schritt 9 fortgesetzt.

- Schritt 4 (Mustersuche)
  Es wird $x^{(k+1,0)} = 2x^{(k,n)} - x^{(k-1,n)}$ (Extrapolation),
  und $s_i^{(k+1)} = s_i^{(k)} \operatorname{sign}(x_i^{(k,n)} - x_i^{(k-1,n)})$ für alle $1 \leq i \leq n$ gesetzt. (Hierdurch werden eventuell die positiven und negativen Richtungen im nächsten Erkundungs-Schritt vertauscht.)
  Es wird $k \leftarrow k + 1$ erhöht und $i = 1$ gesetzt. (Das Ergebnis der Mustersuche wird zunächst nicht überprüft!)

- Schritt 5 (Erkundung nach Extrapolation)
  Es wird $\tilde{x} = x^{(k,i-1)} + s_i^{(k)} e_i$ gebildet.
  Falls $f(\tilde{x}) < F(x^{(k,i-1)})$ wird mit Schritt 6 fortgesetzt;
  ansonsten wird $\tilde{x} = \tilde{x} - 2s_i^{(k)} e_i$ gesetzt.
  Falls $f(\tilde{x}) < F(x^{(k,i-1)})$ wird mit Schritt 6 fortgesetzt;
  ansonsten wird $\tilde{x} = \tilde{x} + s_i^{(k)} e_i$ gesetzt.

- Schritt 6 (Innere Schleife über die Koordinatenrichtungen)
  Es wird $x^{(k,i)} = \tilde{x}$ gesetzt.
  Falls $i < n$ wird $i \leftarrow i + 1$ gesetzt und mit Schritt 5 fortgesetzt.

- Schritt 7 (Test auf Fehlschlag der Mustersuche)
  Falls $f(x^{(k,n)}) \geq f(x^{(k-1,n)})$ (zurück zur Position vor der Mustersuche)
  werden $x^{(k+1,0)} = x^{(k-1,n)}$ und $s_i^{(k+1)} = s_i^{(k)}$ für alle $1 \leq i \leq n$ gesetzt und mit Schritt 10 fortgesetzt.

- Schritt 8 (Abbruchtest nach erfolgreicher Mustersuche)
  Falls $|x_i^{(k,n)} - x_i^{(k-1,n)}| \leq \frac{1}{2}|s_i^{(k)}|$ für alle $1 \leq i \leq n$ wird $x^{(k+1,0)} = x^{(k-1,n)}$ gesetzt und mit Schritt 9 fortgesetzt;
  ansonsten wird mit Schritt 4 fortgesetzt (für eine weitere Mustersuche).

- Schritt 9 (Reduzierung der Schrittlänge und Abbruchtest)
  Falls $|s_i^{(k)}| \leq \varepsilon$ für alle $1 \leq i \leq n$ wird die Suche mit dem Ergebnis $x^{(k,0)}$ beendet;
  ansonsten wird $s_i^{(k+1)} = \frac{1}{2} s_i^{(k)}$ für alle $1 \leq i \leq n$ gesetzt.

- Schritt 10 (Iteration) Es wird $k \leftarrow k + 1$ erhöht und $i = 1$ gesetzt und mit Schritt 1 fortgesetzt.

In CALMIR2 ist, wie bereits erwähnt, die Startkonfiguration die identische Bézier-Deformation des Quaders Q auf sich selbst und die Anfangsschrittlängen werden auf ein Drittel der Abstände zwischen zwei Kontrollpunkten in die entsprechende Koordinatenrichtung gesetzt. Damit der Benutzer die Optimierung jederzeit vorzeitig abbrechen kann, werden auch die bisher beste NURBS-Fläche und der zugehörige Wert der Zielfunktion (in den Variablen `last_min_merit` und `min_merit`) gespeichert.

Des weiteren wurde die Endbedingung in Schritt 9 geändert. So wird nicht mehr ein absoluter Test der Schrittlängen durchgeführt, sonder mittels

$$|s_i^{(k)}| \leq \varepsilon |s_i^{(0)}| \tag{4.9}$$

für alle $1 \leq i \leq n$ ein relativer Vergleich zu den Anfangs-Schrittlängen durchgeführt. Außerdem wird die Extrapolation abgebrochen, falls die Verbesserung der Zielfunktion zu schwach ist (im Vergleich zum Anfangswert in der Variablen `start_merit`)

```
(last_min_merit - min_merit)/start_merit < accuracy,
```
(4.10)

wobei die Abbruch-Genauigkeit (`accuracy`) auch vom Benutzer vorgegeben wird. Dies war notwendig, weil bei ersten Versuchen der Optimierer immer noch in der Größenordnung von 10^{-4} unterhalb des Wertes der Zielfunktion verbesserte.

5 Aufbau des Programms

In diesem Kapitel wird die C++-Implementierung des Algorithmus zusammen mit den verwendeten Programmpaketen vorgestellt.

Für die Implementierung wurde die Programmiersprache C++ aus mehreren Gründen gewählt. Zunächst einmal war es damit möglich, bereits bestehende Programmpakete, die in C oder C++ geschrieben waren, weiter zu nutzen. Des weiteren boten sich insbesondere die objektorientierten Sprachmittel (s. Abschnitt 5.1) zur Vereinfachung eines so komplexen Algorithmus geradezu an. Und schließlich ist C/C++ auch eine der am weitesten verbreiteten Programmiersprachen, was die Portierung des Programms auf andere Architekturen erleichtert.

Für die grundlegenden mathematischen Strukturen werden die Standard-Bibliotheken *BLAS* und *LAPACK* (s. Abschnitt 5.2) benötigt, die im Original als Fortran-Bibliotheken implementiert sind, aber von J. Koch im Rahmen der Matrix-Klasse nach C/C++ portiert wurden (s. Abschnitt 5.3).

Ebenfalls von J. Koch stammt die Spline-Bibliothek, die grundlegende Routinen zum Rechnen mit Splines bereitstellt (s. Abschnitt 5.4).

Im Rahmen der grafischen Oberfläche wird das *Display*-Programmpaket [Kra97] zur schrittweisen Visualisierung des Algorithmus und zur möglichen interaktiven Steuerung wichtiger Designparameter verwendet (s. Abschnitt 5.5).

Zur Realisierung des Octree und der Triangulierung wird in der Implementierung die Bibliothek *NetzDatStrukt* verwendet, die von D. Nowottny und A. Fuchs zur Verwaltung zwei- und dreidimensionaler Netze entworfen wurde [NF] (s. Abschnitt 5.6).

Für die Visualisierung der Splineflächen wird schließlich noch die *bx*-Bibliothek von H. Bohl benötigt (s. 5.7).

5.1 Objektorientierte Programmierung in C++

Der Begriff „objektorientiert“ ist inzwischen zum Gütesiegel geworden, ohne daß immer klar ist, was darunter zu verstehen ist. Es zeigt sich sogar, daß es verschiedene Interpretationen des Begriffs gibt und der Begriff mitunter auch mißbraucht wird, um Software mit diesem Gütesiegel zu versehen.

Genauso umstritten wie der Begriff ist auch die Frage, ob und wann etwas objektorientiert ist. Es gibt zwar verschiedene Kennzeichen objektorientierter Sprachen, aber es ist nicht eindeutig festgelegt, ob alle Kennzeichen oder nur eine Auswahl davon vorhanden sein müssen.

C++ unterstützt alle Sprachmittel, die nach vorwiegender Meinung objektorientierte Sprachen kennzeichnen. Trotzdem gibt es zum Teil Diskussionen darüber, ob C++ eine echte objektorientierte Sprache ist. Dies liegt daran, daß C++ im Gegensatz zu Sprachen wie Smalltalk oder Eiffel keine rein objektorientierte Sprache ist. C++ unterstützt zwar objektorientierte Programmierung, durch die Abwärtskompatibilität zu C kann man aber auch C++-Programme schreiben, die die objektorientierten Sprachmittel nicht nutzen.

Bevor diese Frage in einen Glaubenskrieg ausartet, sollen nun aber die wesentlichen Kennzeichen von objektorientierter Programmierung vorgestellt werden. Diese Kennzeichen lassen sich im Grunde auf vier Begriffe zurückführen [Jos96]:

- Klassen (abstrakte Datentypen)
- Datenkapselung
- Vererbung
- Polymorphie

5.1.1 Objekte und Klassen

Im Mittelpunkt der objektorientierten Programmierung steht das *Objekt*. Ein Objekt ist „Etwas“, das betrachtet wird, das verwendet wird, das eine Rolle spielt. Wenn man objektorientiert programmiert, versucht man, die Objekte, die im Problemfeld des Programms eine Rolle spielen, zu ermitteln und zu implementieren.

Wenn Objekte betrachtet werden, interessiert allerdings nicht nur, wie sie aufgebaut sind bzw. was sie repräsentieren, sondern genauso wichtig ist das, was man mit ihnen machen kann, also die Operationen, die für ein Objekt aufgerufen werden können.

Und hier kommt der Begriff *abstrakter Datentyp* ins Spiel: Ein abstrakter Datentyp beschreibt nicht nur die Eigenschaften (*Attribute*) eines Objekts, aus denen es sich zusammensetzt, sondern auch dessen Verhalten (die *Methoden*).

Abbildung 5.1: Objekt mit Attributen und Methoden

Zur Realisierung von abstrakten Datentypen hat man in der objektorientierten Programmierung den Begriff *Klasse* eingeführt. Eine Klasse ist die Implementierung eines abstrakten Datentyps.

5.1.2 Datenkapselung

Ein Problem von C-Strukturen ist, daß jederzeit auf alle Komponenten einer solchen Struktur zugegriffen werden kann. Jedes Programmstück bietet also für alle Anwender die Möglichkeit, den Zustand eines Objekts beliebig zu verändern. Damit besteht die Gefahr, bei Anwendung einer Klasse Fehler und Inkonsistenzen zu produzieren.

Daraus entstand die Idee der *Datenkapselung*: Um sicherzustellen, daß nicht jeder Anwender mit einem Objekt machen kann, was er will, wird einfach der Zugriff auf ein solches Objekt auf eine wohldefinierte *Schnittstelle* reduziert.

Jeder Anwender, der mit einem solchen Objekt umgeht, soll nur die Operationen durchführen können, die der Designer des entsprechenden Datentyps für einen öffentlichen Zugriff vorgesehen hat. Auf Interna, die zur internen Verwaltung des Objekts im Programm dienen, besteht kein Zugriff.

5.1.3 Vererbung

Die objektorientierte Programmierung erweitert die abstrakten Datentypen durch die Einführung von Typ-/Untertyp-Beziehungen zwischen einzelnen Klassen. Dies wird mit einem Mechanismus erreicht, der als *Vererbung* bezeichnet wird. Statt gemeinsame Eigenschaften einer Klasse immer wieder neu zu implementieren, kann eine Klasse ausgewählte Attribute und Methoden an andere Klassen vererben.

In C++ ist die Vererbung in Form der *Klassenableitung* implementiert. Verschiedene Klassen können in einer hierarchischen Beziehung zueinander stehen. Dabei gilt, daß eine abgeleitete Klasse immer eine Spezialisierung ihrer *Basisklasse* ist. Umgekehrt ist die Basisklasse die *Generalisierung* der abgeleiteten Klasse. Wenn eine abgeleitete Klasse mehrere Basisklassen besitzt, spricht man von *Mehrfachvererbung*.

5.1.4 Polymorphie

Vererbung ist eine wichtige Grundvoraussetzung, um *Polymorphie* zu ermöglichen. Die Arbeit mit Oberbegriffen wird nämlich dann erst sinnvoll, wenn sich Objekte zeitweise auf ihren Oberbegriff reduzieren lassen.

Polymorphie bedeutet letztlich die Fähigkeit, daß eine Operation vom Objekt selbst interpretiert wird, und zwar unter Umständen erst zur Laufzeit, da beim Compilieren nicht unbedingt bekannt ist, welche Operation gemeint ist. Dieser Sachverhalt wird auch als *dynamisches Binden* bezeichnet. Die dynamische Bindung wird in C++ durch *virtuelle* Klassenfunktionen erreicht.

Nach diesem Ausflug in die Grundbegriffe der objektorientierten Programmierung werden nun die verwendeten Programmpakete näher beschrieben.

5.2 BLAS und LAPACK

BLAS (***B**asic **L**inear **A**lgebra **S**ubprograms*) ist eine Fortran-Bibliothek, die Routinen für elementare Vektor- und Matrix-Operationen zur Verfügung stellt. Level 1 BLAS sind dabei Vektor-Vektor-Operationen, Level 2 BLAS sind Vektor-Matrix-Operationen und Level 3 BLAS schließlich Matrix-Matrix-Operationen. Die Sourcen und eine Kurzdokumentation findet man unter [Neta]. Eine ausführlichere Dokumentation bilden [L+79], [D+86] und [D+88].

Die Fortran-Bibliothek LAPACK (***L**inear **A**lgebra **PACK**age*) ist eine Weiterentwicklung der EISPACK- und LINPACK-Programmpakete und basiert wie diese auf BLAS. LAPACK stellt Routinen zum Lösen von linearen Gleichungssystemen, Auffinden von *least-squares*-Lösungen und zur Berechnung von Eigenwert-Problemen zur Verfügung. Des weiteren umfaßt LAPACK Matrix-Zerlegungen wie z. B. LR-, QR- oder Cholesky-Zerlegung. Den Sourcecode sowie einen Überblick über LAPACK findet man unter [Netb].

5.3 Matrix-Paket

Das Ziel des von J. Koch geschriebenen C++-Matrix-Pakets [Koc93] ist es, ein Softwaretool zur Verfügung zu stellen, das in der Benutzung einfach ist und die wichtigsten Operationen zur Manipulation von Vektoren und Matrizen enthält. Die Syntax orientiert sich dabei an der Programmiersprache MATLAB. Die im Matrix-Paket implementierten Datentypen sind

- *indvec*:
  Ein Indexvektor als eindimensionales Feld von *integer*-Zahlen.
- *vector*:
  Ein eindimensionales Feld aus Zahlen vom Typ *double*.
- *matrix*:
  Eine Matrix ist ein zweidimensionales Feld mit *double*-Zahlen.

Zusätzlich zu diesen Neuerungen wurden im Rahmen des Matrix-Pakets die oben beschriebenen Bibliotheken BLAS und LAPACK mittels des Programms f2c von Fortran nach C konvertiert und stehen nun als `libmatrixblas` bzw. `libmatrixlapack` auch für Plattformen zur Verfügung, die keinen Fortran-Compiler besitzen.

5.4 Die Spline-Bibliothek

Die ebenfalls von J. Koch geschriebene Spline-Bibliothek basiert auf dem Matrix-Paket und stellt die wichtigsten Routinen zum Rechnen mit Spline-Funktionen zur Verfügung.

Dazu gehören z. B. die Auswertung von Splines, das Einfügen von Knoten oder die Umwandlung von B-Splines in Bézier-Form. Durch die Verwendung der Matrix-Klasse sind die Algorithmen sehr effizient, weil mehrere Berechnungen in einem Schritt parallel ausgeführt werden können, so z. B. die parallele Auswertung eines Splines an verschiedenen Punkten. Eine ausführliche Beschreibung sowie eine Installationsanleitung findet man in [Koc94].

5.5 Das Display-Programmpaket

Das Display-Programmpaket wurde zur grafischen Visualisierung geometrischer Objekte entworfen. Neben der verwendeten Display-Bibliothek existiert auch noch ein ausführbares Programm zur Abarbeitung von Skript-Files. Display verwendet als Ausgaberoutinen für den Bildschirm wahlweise die OpenGL-, die GL- oder die VOGL-Bibliothek.

Das herausragende Merkmal von Display ist das getrennte Steuerungsfenster zur interaktiven Kontrolle der wichtigsten Design-Parameter (s. Abbildung 5.2).

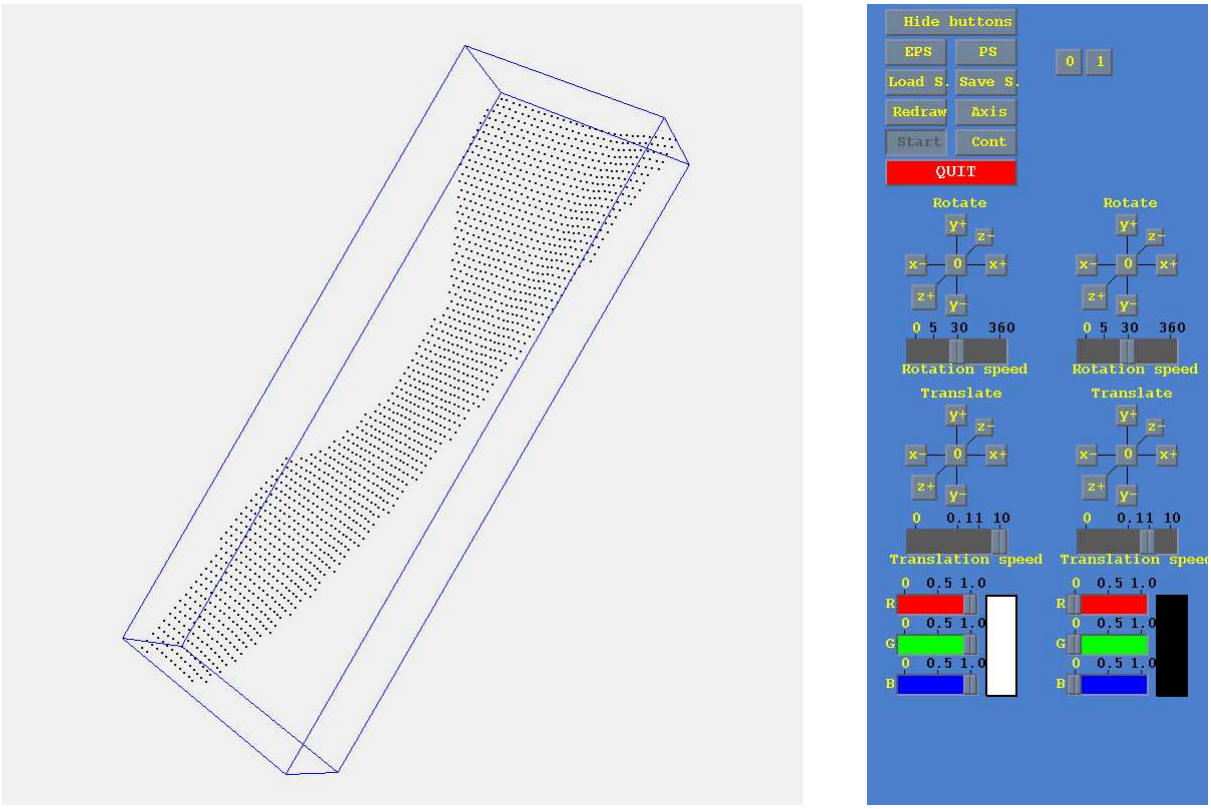


Abbildung 5.2: Display-Oberfläche mit Steuerungsfenster

So ist es am Bildschirm möglich, das dargestellte Objekt zu drehen, zu verschieben oder zu skalieren (*zoomen*). Des weiteren können auch die Lichtquellen interaktiv beeinflußt

werden, insbesondere kann deren Lage und Farbe verändert werden. Falls man die Szene im Grafikfenster weiterbearbeiten will, kann man über das Steuerungsfenster die aktuelle Darstellung in einer Datei speichern, wahlweise im EPS- oder PS-Format.

Und schließlich ist es noch möglich, das im Hintergrund ablaufende Programm, welches das dargestellte grafische Objekt berechnet und verwaltet — in diesem Fall die Implementierung von CALMIR2 — zu stoppen oder neu zu starten. Der aktuelle Stand des Programms wird dabei zur Information immer in der Kopfzeile des Grafikfensters eingeblendet.

5.6 Das NetzDatStrukt-Programmpaket

Die C++-Implementierung des Algorithmus greift wie schon erwähnt auf die C++-Datenstruktur *NetzDatStrukt* zurück, die zur Verwaltung zwei- und dreidimensionaler Netze dient. Eine Beschreibung der Fähigkeiten dieses Programmpakets sowie Anwendungsbeispiele findet man in [Fuc99] und [Now99].

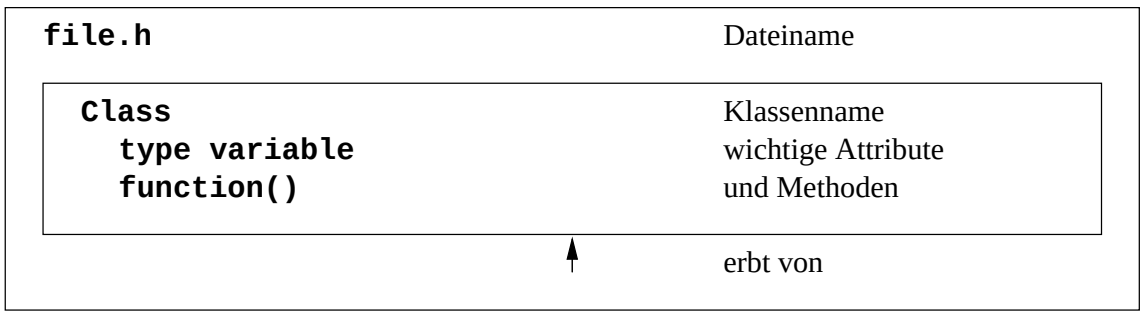


Abbildung 5.3: Legende

In diesem Abschnitt werden einige Klassen dieser Bibliothek aufgeführt, zusammen mit den davon abgeleiteten Klassen, die für die Implementierung des Algorithmus verwendet werden. Es sind nur die wichtigsten Klassen und einige ihrer Attribute bzw. Methoden angegeben, für eine komplette Übersicht wird auf [NF] verwiesen. Alle NetzDatStrukt-Klassen haben auch eine grafische Ausgabefunktion, die auf das Display-Paket zurückgreift, sowie eine Dateiausgabe.

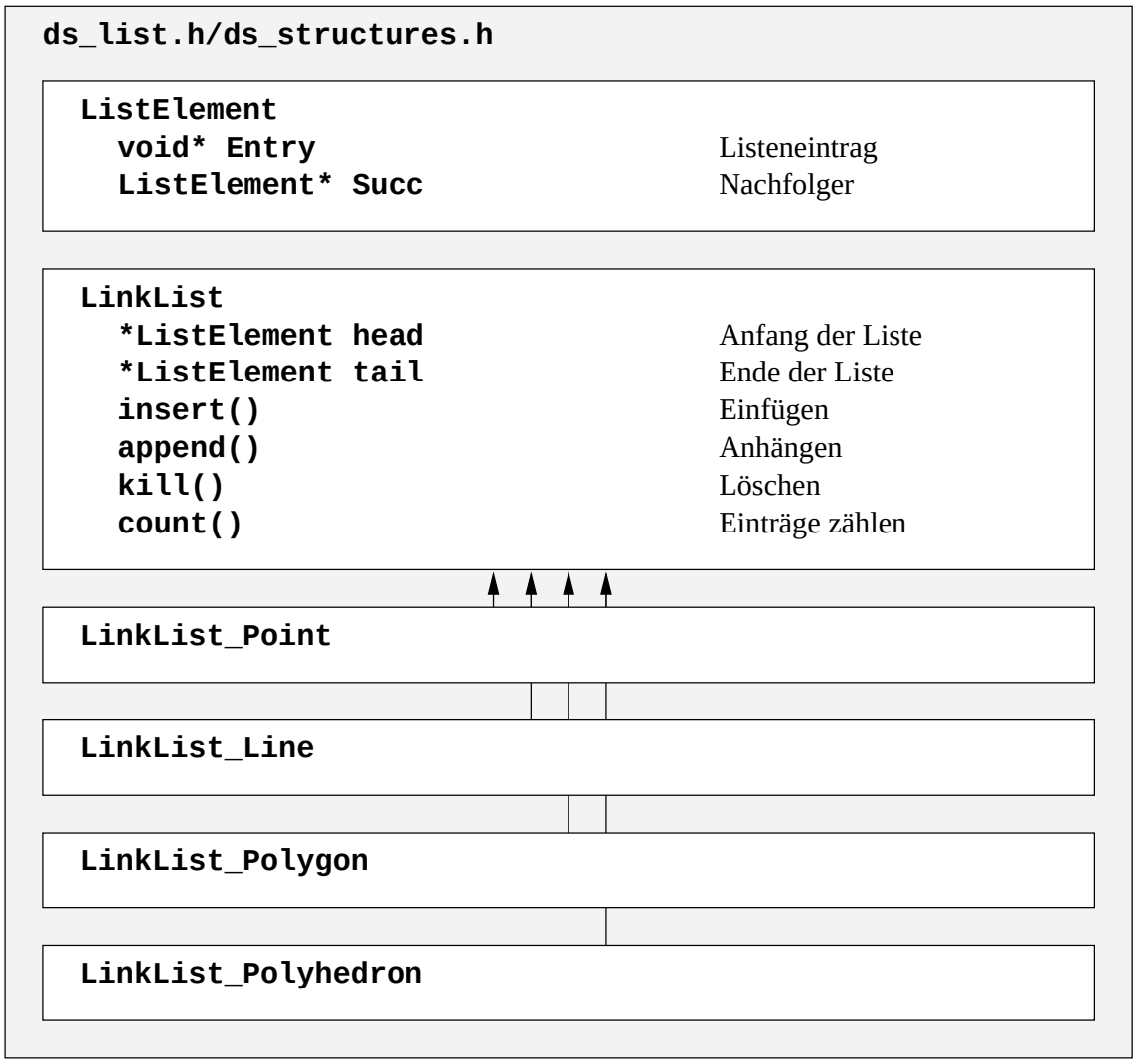


Abbildung 5.4: Die Module `ds_list.h` und `ds_structures.h`

Die Listen in `ds_list.h` und `ds_structures.h` sind durchweg als einfach verkettete, ungeordnete Listen implementiert (vgl. Abbildung 5.4). Die abgeleiteten Klassen `LinkList_*` sind Spezialisierungen für die entsprechenden geometrischen Objekte.

In `ds_elements.h` werden die Schnittstellen der wichtigsten geometrischen Objekte definiert. Die Klassen `Polyhedron`, `Line` und `Point` bilden dabei die Basis. Sie besitzen jeweils in dieser Richtung Verweise auf die nächste verwendete Klasse (vgl. Abbildung 5.5). Die Rückverweise sind erst in den abgeleiteten Klassen `Vertex`, `Edge` und `Element` implementiert.

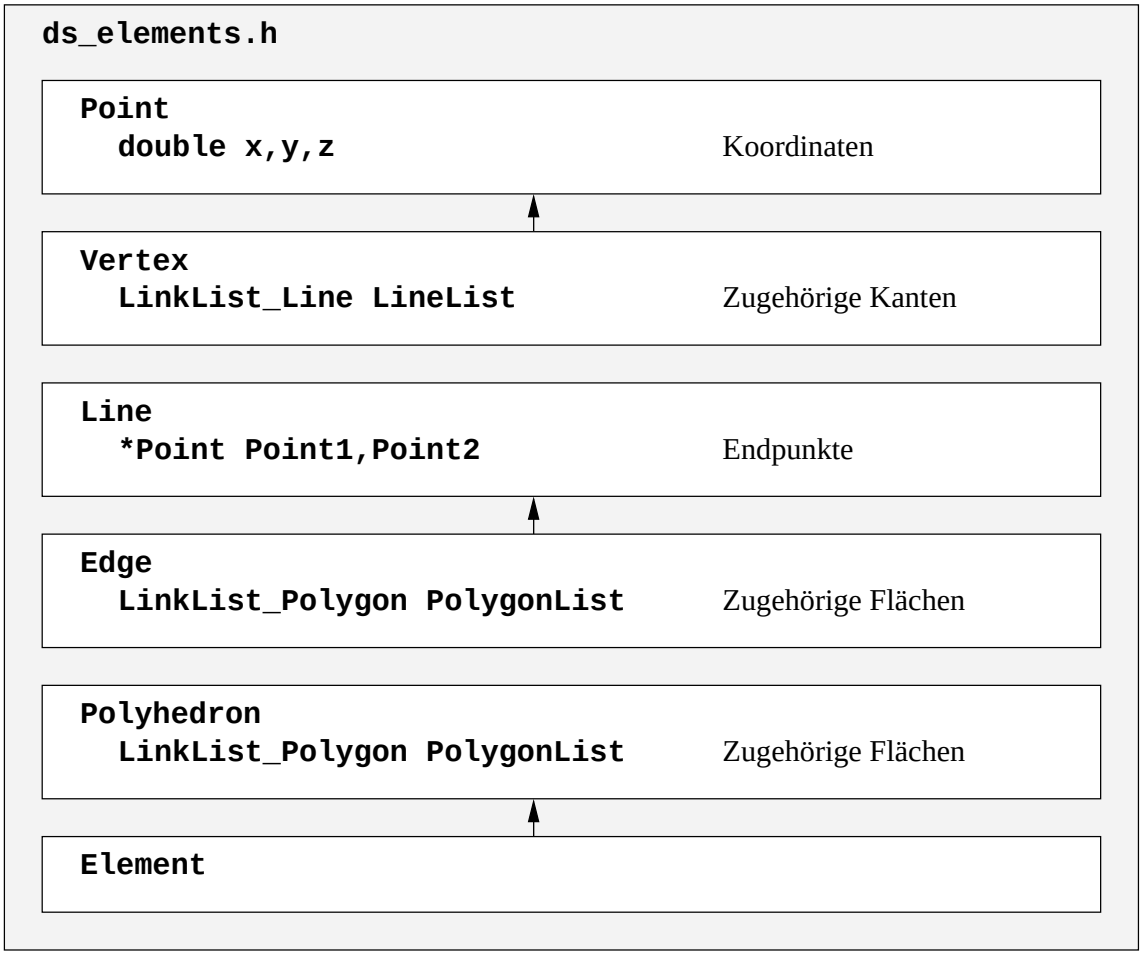


Abbildung 5.5: Die Module `ds_elements.h` und `box.h`

5.7 Die bx-Bibliothek

Das verwendete C++-Paket bx stellt eine Biliothek zur Verwaltung und Darstellung von Spline-Kurven und Flächen dar [Boh99]. Die wichtigsten verwendeten Strukturen sind dabei die Koordinaten für ebene und räumliche, euklidische und homogene Koordinaten

- `bxCoord2`
- `bxCoord3`
- `bxCoord2Hom`
- `bxCoord3Hom`

und darauf aufbauend die Klassen

- `bxSplineCurveP`
- `bxSplineCurveS`
- `bxNURBSCurveP`
- `bxNURBSCurveS`

für Spline-Kurven und NURBS, sowie die Klassen

- `bxSplineSurfaceA`
- `bxNURBSSurface`

für Spline- und NURBS-Flächen. Alle Klassen bieten komfortable Routinen zur Datei-Ein- und -Ausgabe. Für die Berechnungen wird intern auf die Matrix-Klasse zurückgegriffen und die Visualisierung verwendet die oben beschriebene Display-Bibliothek.

5.8 Besonderheiten der Windows- und Linux-Varianten

Prinizpiell läßt sich CALMIR2 für alle Architekturen compilieren, für die die obengenannten Bibliotheken sowie ein POSIX-kompatibler C++-Compiler zur Verfügung stehen. Momentan existieren lauffähige Varianten für Windows NT (CALMIR2 Version 3.1) und Linux mit Kernel 2.4.27 (CALMIR2 Version 4.1).

Beide Versionen zeigen neben den beiden Standard-Fenstern des Display-Programms (s. Abschnitt 5.5) zwei weitere Fenster. Das größere zeigt dabei den Verlauf der Optimierung

an, d. h. den Wert der Zielfunktion aus Gleichung 2.33 nach jedem Optimierungsschritt, während im kleineren die Fehler in der Intensitätsmatrix

$$w_{i,j}\left(\frac{\Phi_{i,j}}{\Phi}-\frac{\widetilde{\Phi}_{i,j}}{\widetilde{\Phi}}\right)^2 \tag{5.1}$$

angegeben werden.

Abbildung 5.6: Verlauf der Zielfunktion

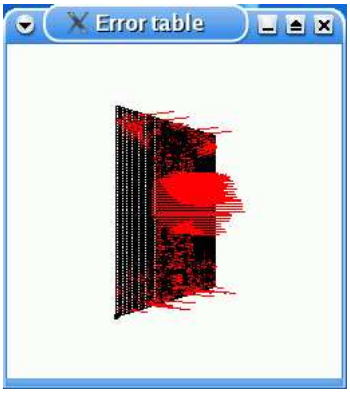


Abbildung 5.7: Aktueller Fehler in der Intensitätsmatrix

Für die Windows-Version gibt es ein getrenntes Startprogramm mit grafischer Benutzeroberfläche, über die alle Kommandozeilenargumente (vgl. Abschnitt 5.9) mittels Karteikarten und Formfelder eingegeben werden können. Das Startprogramm übergibt die Werte anschließend als Kommandozeilenparameter an CALMIR2 und startet dieses. Dies erlaubt eine benutzerfreundlichere Verwaltung der Parameter, aber selbstverständlich kann das Programm auch unter Windows direkt über die Kommandozeile mit Angabe aller Parameter gestartet werden.

Die Linux-Version wird immer über die Kommandozeile gestartet, sie verfügt jedoch als Besonderheit über eine Schnittstelle zu Matlab (s. [Mat05]), über die statt dem Hooke-Jeeves-Algorithmus ein in Matlab geschriebenes Optimierungsverfahren eingebunden werden kann. Hierfür können auch alle von Matlab bereits mitgelieferten Optimierungsverfahren – wie z. B. `fmincon` aus der Optimierungs-Toolbox – verwendet werden.

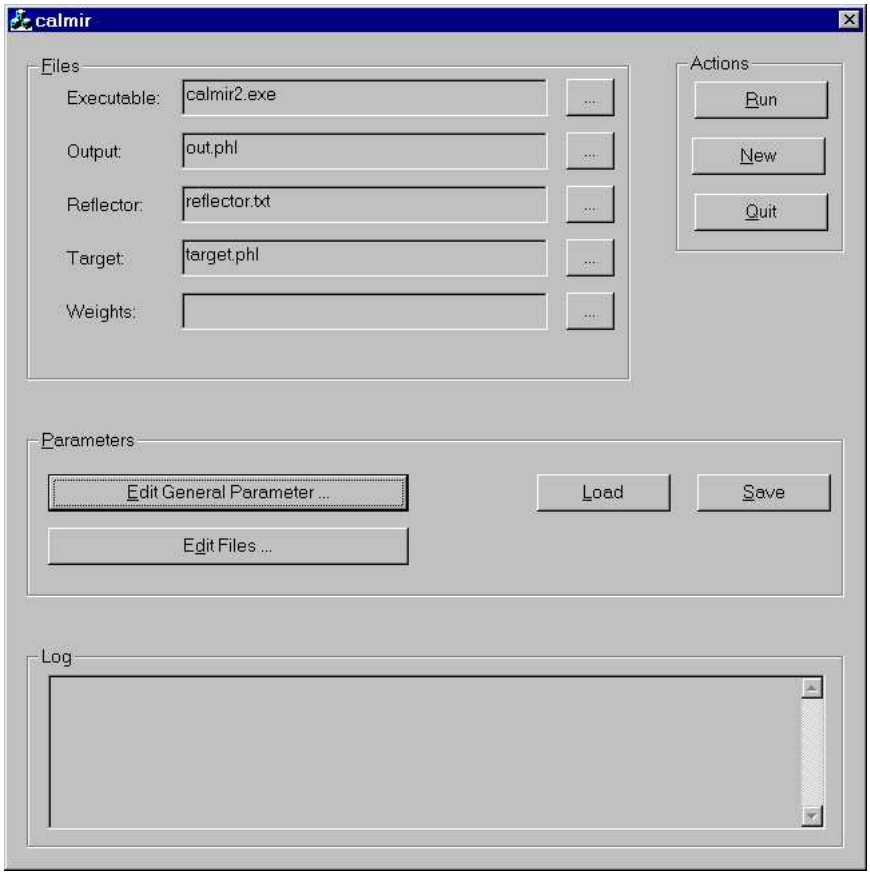


Abbildung 5.8: Startprogramm für CALMIR2 unter Windows

5.9 Kommandozeilenparameter

Der Aufruf des CALMIR2-Programms ist

`calmir2.exe [`*`OPTION...`*`] `*`FILE N r b t u v...`*

wobei *OPTION* eine oder mehrere der nachfolgenden Kommandozeilenoptionen sein muß. Die untenstehende Tabelle gibt an, welcher Buchstabe als Option anerkannt wird, welches Argument diesem Buchstaben evtl. folgen muß, welcher Default-Wert für diese Option bereits beim Start gesetzt ist, ob die Option ein Pflicht-Parameter ist und schließlich eine Kurzbeschreibung dieser Option.

Option	Default	Pflicht	Beschreibung
`-A #.#`		*	ΔA in Grad ($0° < \Delta A \leq 360°$)
`-a #.#`		*	$\Delta\alpha$ in Grad ($0° < \Delta\alpha \leq 180°$)
`-B #.#`		*	ΔB in Grad ($0° < \Delta B \leq 360°$)
`-b #.#`		*	$\Delta\beta$ in Grad ($0° < \Delta\beta \leq 180°$)
`-c #`	1000	nein	Anzahl der Iterationen bis zum Stop (> 0)
`-D #.#`		ja	Durchmesser des Befestigungszylinders in negativer Lampen-Richtung (≥ 0)
`-d #.#`		ja	Durchmesser des Befestigungszylinders in positiver Lampen-Richtung (≥ 0)
`-e #.#`	1.0	nein	Maximaler Fehler der Triangulierung (> 0)
`-G`		nein	Start ohne grafische Benutzeroberfläche
`-H "..."`		ja	Datei für die Ziel-Lichtverteilung
`-h`		nein	Ausgabe des Hilfe-Texts
`-I #`	0	nein	Initialisierung des Zufallsgenerators (≥ 0)
`-i #`		ja	Maximale Anzahl Reflexionen im Raytracer (≥ 1)
`-L #.#`		ja	Durchmesser der Lampe (> 0)
`-l #.#`		ja	Länge der Lampe (> 0)
`-m #`	0	**	Strahlgenerierung; 0: Zufall, 1: winkelabhängig
`-M "..."`		nein	Datei mit der Gewichts-Matrix
`-N`		nein	Normalen-Interpolation bei Reflexion
`-n #.#`		**	Phong-Exponent (≥ 1)
`-O "..."`		ja	Ausgabe-Datei
`-o #`	1	nein ***	Optimierungs-Modus; 0: direkt; 1, 2: Bézier-Raum-Deformation 1, 2
`-p #.#`		ja	Minimaler Lichtstrom in Prozent (> 0, < 100)
`-R "..."`		ja	Reflektor-Datei
`-r #`		ja	Anzahl der Lichtstrahlen (≥ 0)
`-S #`		**	Anzahl der Scheiben (≥ 1)
`-s #`		**	Anzahl der Sektoren (≥ 1)

Option	Default	Pflicht	Beschreibung
`-t`		nein	Standard-Triangulierung (nicht orth. B-Splines)
`-U #`		ja	Anzahl der Quader in x-Richtung (≥ 1)
`-V #`		ja	Anzahl der Quader in y-Richtung (≥ 1)
`-W #`		ja	Anzahl der Quader in z-Richtung (≥ 1)
`-w #.#`	0.9	nein	Gewicht w in der Zielfunktion ($\geq 0, \leq 1$)
`-X #.#`		ja	x-Koordinate der Lampenrichtung
`-Y #.#`		ja	y-Koordinate der Lampenrichtung
`-Z #.#`		ja	z-Koordinate der Lampenrichtung
`-x #.#`		ja	x-Koordinate der Lampenmitte
`-y #.#`		ja	y-Koordinate der Lampenmitte
`-z #.#`		ja	z-Koordinate der Lampenmitte

Dabei gilt:

- `"..."`: steht für einen Dateinamen,
- `#`: steht für eine ganze Zahl,
- `#.#`: steht für eine Fließkommazahl,
- Die Lampenrichtung (`-X`, `-Y`, `-Z`) darf nicht $(0, 0, 0)$ sein,
- *: entweder `-A` und `-a` oder `-B` und `-b` müssen angegeben werden,
- **: Falls die Strahlgenerierung (`-m`) 0 ist, muß `-n` angegeben werden und `-r` gibt die Gesamtanzahl der Strahlen an (vgl. Abschnitt 2.1.2); ansonsten müssen `-S` und `-s` angegeben werden und `-r` gibt die Anzahl der Strahlen pro Teilstück an (vgl. Abschnitt 2.1.1).
- ***: Die beiden implementierten Bézier-Raum-Deformationen haben den Spline-Grad und die Schrittweiten für Typ 1:

Grad			Schrittweite		
x	y	z	x	y	z
1	1	1	0.3	0.3	0.3
2	2	1	0.3	0.3	0.3
3	3	1	0.3	0.3	0.3

und für Typ 2:

Grad			Schrittweite		
x	y	z	x	y	z
1	1	1	0.0	0.0	0.3
2	2	1	0.0	0.0	0.3
3	3	1	0.0	0.0	0.3

Der Abschnitt *FILE N r b t u v...* steht für eine oder mehrere Dateien, die die NURBS-Daten einer Reflektorfläche enthalten. Die nachfolgenden Argumente, die in der untenstehenden Tabelle beschrieben werden, sind dabei alle zwingend erforderlich. Die Default-Orientierung der Normalen-Richtung der NURBS-Fläche ist $v \times u$, und für $N \neq 0$ wird $u \times v$ verwendet.

Option	Typ	Beschreibung
N	#	Umkehrung der Normalen-Richtung
r	#.#	Reflexions-Faktor (≥ 0, ≤ 1)
b	#.#	Phong-Exponent (≥ 1)
t	#.#	Maximaler θ-Winkel für enge Streuung in Grad ($\geq 0°$, $\leq 90°$)
u	#	Anzahl der Anfangs-Dreiecke in u-Richtung (≥ 1)
v	#	Anzahl der Anfangs-Dreiecke in v-Richtung (≥ 1)

Während des Programmlaufs wird eine Datei namens `display.head` im aktuellen Verzeichnis erstellt, die die Einstellungen und Parameter für die Display-Bibliothek enthält. Für das Programm CALMIR2 müssen also Leserechte auf die Datei mit der Ziel-Lichtverteilung (`-H`) und die Dateien mit den NURBS-Daten (*FILE...*) sowie Schreibrechte im aktuellen Verzeichnis und auf die Ausgabedateien gesetzt sein.

6 Testergebnisse

Dieses Kapitel stellt eine kurze Zusammenfassung von [App00b] dar. Dabei werden nur ausgewählte Tests aus diesem Bericht verwendet, weil die gesamte Fülle der Testdaten den Rahmen dieser Arbeit sprengen würde.

Da es aus Gründen des Firmengeheimnisses nicht möglich war, von Philips echte Testdaten zu bekommen, wurde für die Tests ein etwas geänderter Ansatz verfolgt. Zunächst wird für eine gegebene Reflektorfläche mit dem Raytracer die Lichtverteilung bestimmt. Sodann wird die originale Fläche modifiziert und zusammen mit der berechneten Lichtverteilung als Ziel dem Optimierungs-Algorithmus als Startwert übergeben. Bei einem Test wird dann überprüft, ob es mit CALMIR2 gelingt, während der Optimierung wieder eine Fläche zu finden, die der vorgegebenen Lichtverteilung entspricht. Der Vorteil dieses Setups ist, daß bereits bekannt ist, ob eine Lösung existiert – nämlich die originale, unveränderte Fläche.

Ein großes Problem stellt die Tatsache dar, daß – wie bereits in 2.4 erwähnt – die Lichtverteilung im C-γ-System übergeben wird, der Raytracer jedoch im A-α- oder B-β-System arbeitet. Die Routinen zur Umrechnung wurden von Philips aus Kompatibilitätsgründen vorgegeben und sind in [Phi97] beschrieben. Im Prinzip handelt es sich hierbei um einen Glättungsvorgang, der jedoch für $\Delta A = \Delta C = 5°$ bei den vorgegebenen Lichtverteilungen nach Umrechnung in das A-α-System (bzw. B-β-System) und erneuter Umrechnung zurück in das C-γ-System bereits Werte von bis zu 0.1 in der Zielfunktion erreicht, und für kleinere Werte von ΔA bzw. ΔC sogar noch höhere Werte. Es ist deshalb klar, daß nicht zu erwarten ist, daß die Optimierung diesen Wert unterschreiten kann.

6.1 Reflektorflächen und Modifikationen

Ein weiteres Probleme beim Testen des Programms CALMIR2 war die Tatsache, daß Philips nur sehr wenige „echte“ Reflektoren für die Tests zur Verfügung stellte. Die Basis für alle Tests stellt hierbei die Fläche `s5x8sm0` dar, die zusammen mit dem Lampenzylinder in der unten stehenden Abbildung zu sehen ist. Die Abkürzung `5x8` steht dabei für die Anzahl der Kontrollpunkte in u- und v-Richtung, das zweite `s` deutet an, daß die Fläche symmetrisch ist, und `m0` bedeutet, daß die Fläche keiner Modifikation unterworfen wurde.

Die beiden Modifikationen, denen die Fläche für die Testläufe unterzogen wurde, waren einerseits eine globale Modifikation, bei der der gesamte Reflektor orthogonal zur Öffnungsebene um den Faktor 0.9 gestaucht wurde. Die so modifizierten Flächen sind an der Endung `m1` erkenntlich, also z. B. `s5x8sm1.txt`.

Die zweite Modifikation war lediglich eine lokale. Bei ihr wurden die beiden mittleren Kontrollpunkte ebenfalls orthogonal um den Faktor 1.1 von der Öffnungsebene weg verschoben, wodurch eine kleine Ausbeulung entstand. Die so modifizierten Flächen haben die Endung `m2`, für die Tests hier also `s5x8sm2.txt`.

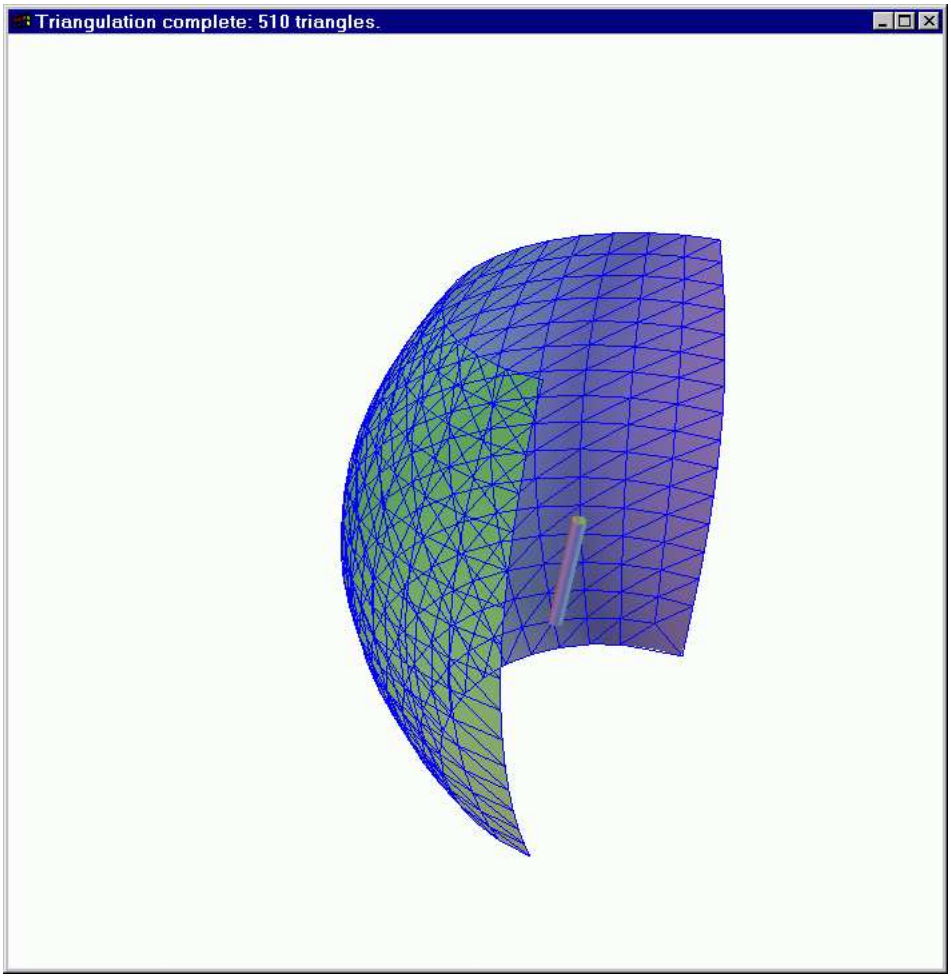


Abbildung 6.1: Reflektorfläche `s5x8sm0.txt`

6.2 Kommandozeilenparameter

Die gemeinsamen Kommandozeilenparameter, die für alle Tests verwendet wurden, sind in untenstehender Tabelle zusammengefaßt (vgl. dazu auch Abschnitt 5.9). Fehlende Optionen wurden je nach Test verändert und sind bei den einzelnen Tests beschrieben.

Option	Wert	Beschreibung
`-A`	5	ΔA in Grad
`-a`	2.5	$\Delta\alpha$ in Grad
`-D`	0	Durchmesser des Befestigungszylinders in negativer Lampen-Richtung
`-d`	0	Durchmesser des Befestigungszylinders in positiver Lampen-Richtung
`-e`	1.0	Maximaler Fehler der Triangulierung
`-I`	0	Initialisierung des Zufalls-Generators

Option	Wert	Beschreibung
`-i`	3	Maximale Anzahl Reflexionen im Raytracer
`-L`	6.5	Durchmesser der Lampe
`-l`	56	Länge der Lampe
`-m`	0	Zufällige Strahlgenerierung
`-N`		Normalen-Interpolation bei Reflexion
`-n`	1.0	Phong-Exponent
`-p`	10.0	Minimaler Lichtstrom in Prozent
`-r`	100000	Anzahl der Lichtstrahlen
`-U`	20	Anzahl der Quader in x-Richtung
`-V`	20	Anzahl der Quader in y-Richtung
`-W`	20	Anzahl der Quader in z-Richtung
`-X`	0	x-Koordinate der Lampenrichtung
`-Y`	0.98	y-Koordinate der Lampenrichtung
`-Z`	0.17	z-Koordinate der Lampenrichtung
`-x`	0	x-Koordinate der Lampenmitte
`-y`	0	y-Koordinate der Lampenmitte
`-z`	0	z-Koordinate der Lampenmitte

Zusätzlich galten folgende Einstellungen für die Reflektorfläche selber.

Option	Wert	Beschreibung
N	0	keine Umkehrung der Normalen-Richtung
r	0.85	Reflexions-Faktor
b	1	Phong-Exponent
t	0	Maximaler θ-Winkel für enge Streuung in Grad (keine Streuung)
u	4	Anzahl der Anfangs-Dreiecke in u-Richtung
v	4	Anzahl der Anfangs-Dreiecke in v-Richtung

6.3 Ziel-Lichtverteilung

Alle Ziel-Lichtverteilungen wurden durch einen Raytracer-Lauf aus der oben beschriebenen Fläche `s5x8sm0.txt` berechnet. Die Einstellungen für den Raytracer werden dabei auch wieder als Kürzel im Dateinamen wiedergegeben. Das Format für die Dateinamen ist dabei

$$u\mathtt{x}v A \Delta m r\mathtt{.phl}$$

Die Bedeutung der einzelnen Buchstaben ist der nächsten Tabelle zu entnehmen.

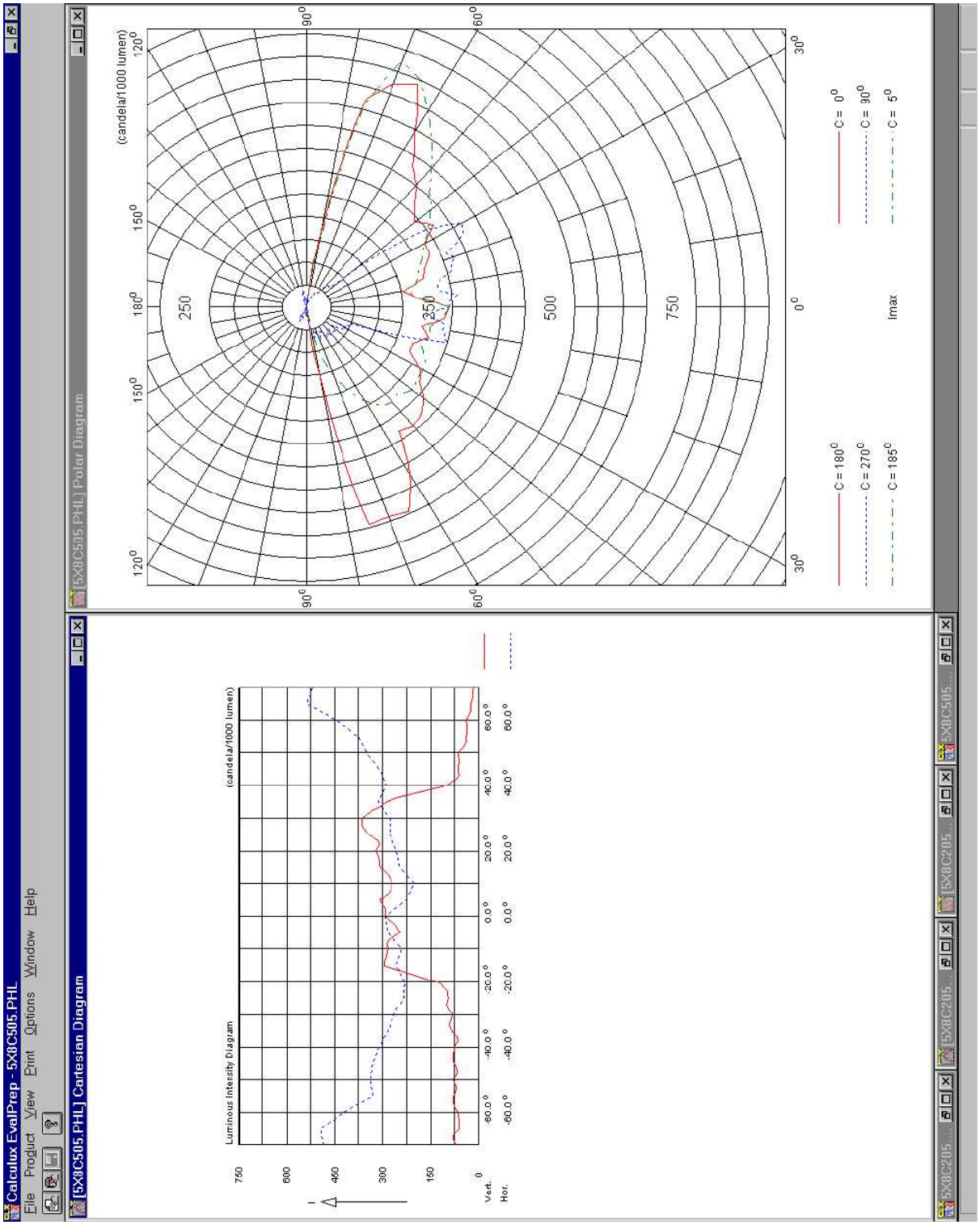


Abbildung 6.2: Lichtverteilung `5x8c505.phl`

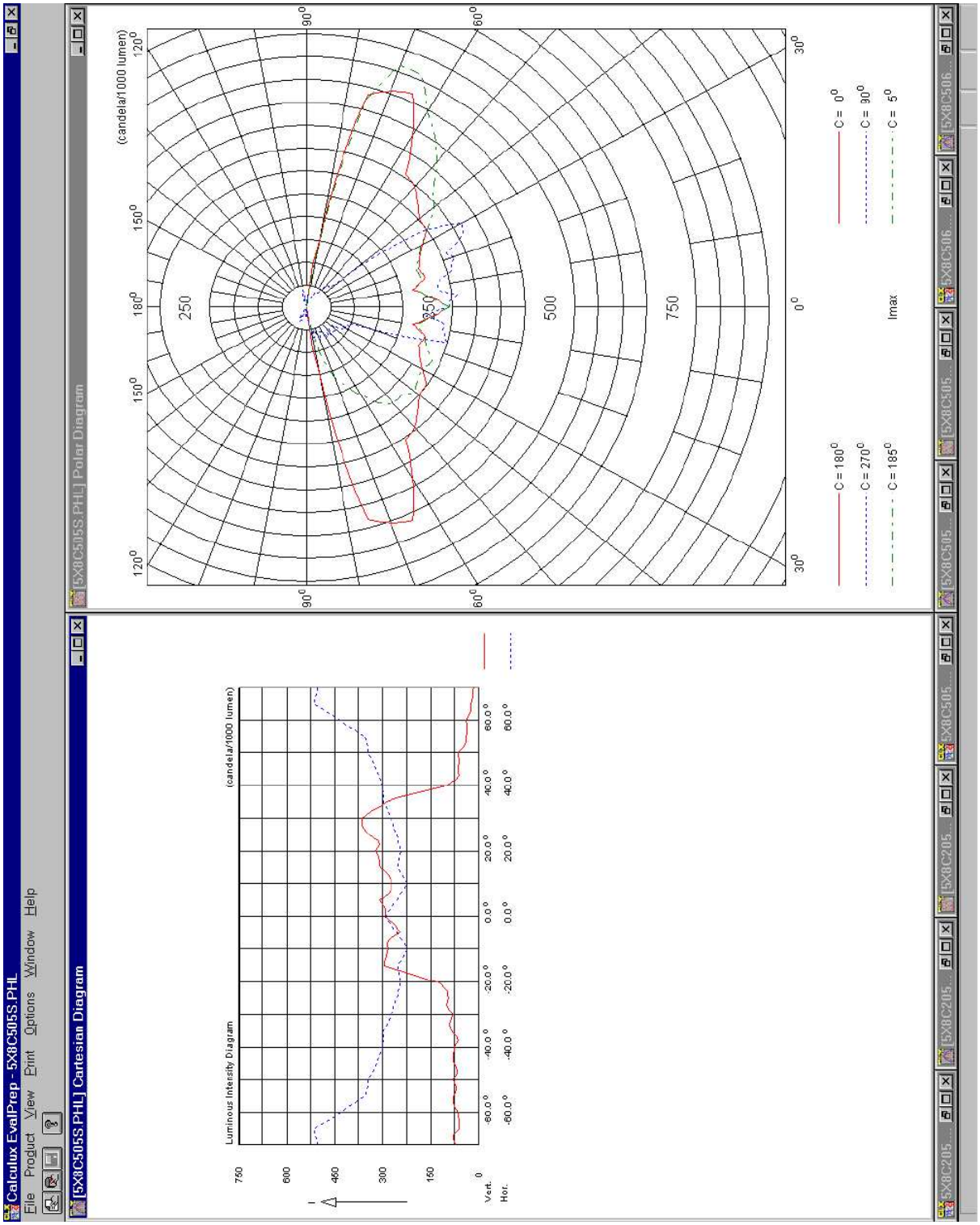


Abbildung 6.3: Lichtverteilung `5x8c505s.phl`

Variable	Beschreibung
`u`	Anzahl der Kontrollpunkte der Fläche in u-Richtung
`v`	Anzahl der Kontrollpunkte der Fläche in v-Richtung
`A`	Koordinatensystem (vgl. 2.4) a: Lichtverteilung ist im A-, α-System c: Lichtverteilung ist im C-, γ-System
Δ	ΔA oder ΔC in Grad $\Delta\alpha$ oder $\Delta\gamma$ sind immer halb so groß
`m`	0: Zufällige Strahlgenerierung 1: Winkelabhängige Strahlgenerierung
`r`	Anzahl der Lichstrahlen ist 10^r

Es hat sich gezeigt, daß bereits mit der relativ geringen Anzahl von 100000 Strahlen und dem recht groben Gitter von $\Delta A = 5°$ bzw. $\Delta C = 5°$ sehr gute Ergebnisse zu erzielen sind. So wird hier als Beispiel die Lichtverteilung `5x8c505.phl` gezeigt.

Die Bilder der Lichtverteilungen sind dabei mit dem Programm CalcuLuX EvalPrep erstellt, das von der Firma Philips zur Verfügung gestellt wurde. Dieses Programm bietet auch die Möglichkeit, vorhandene Lichtverteilungen zu symmetrisieren, wobei diese dann durch ein nachgestelltes `s` kenntlich gemacht sind. Die beiden Abbildungen 6.2 und 6.3 zeigen dieses Vorgehen.

Durch das Symmetrisieren können die Ungleichmäßigkeiten im Raytracer mit zufälliger Strahlgenerierung ausgeglichen werden. Die so entstehende symmetrische Lichtverteilung `5x8c505s` ist für die unten vorgestellten Tests immer die Ziel-Lichtverteilung des Optimierungs-Algorithmus.

6.4 Testläufe

Es werden hier stellvertretend vier Tests aus [App00b] vorgestellt, wobei die Namen der Tests beibehalten wurden. Zu jedem Test wird ein kurzer Auszug aus dem Testprotokoll (d. h. den Ausgabemeldungen von CALMIR2) und Bemerkungen über Besonderheiten gegeben.

In den Testprotokollen bezeichnet `start merit` den Anfangswert der Zielfunktion und `min_merit` den nach dem aktuellen Optimierungsschritt erreichten Endwert. Des weiteren gibt `dimension` die Anzahl der Freiheitsgrade an, abhängig vom aktuellen Grad (`degree`) der Bézier-Raum-Deformation in die einzelnen Koordinatenrichtungen, wobei beim Typ 2 mit Schrittweite 0 in x- und y-Richtung nicht alle Freiheitsgrade für die Optimierung verwendet werden.

Schließlich wird noch angezeigt, nach wievielen Iterationen die Optimierung beendet wurde. Dabei ist zu beachten, daß 100 Iterationen auf einem Pentium II mit 450 MHz ca. zehn Minuten benötigen. Außerdem wird zu jedem Test das Bild der entstandenen Licht-

verteilung gezeigt (vgl. mit 6.3), jedoch nicht die entstandene Reflektorfläche, da sie sich in allen Fällen visuell praktisch nicht mehr von `s5x8sm0.txt` unterscheiden läßt.

6.4.1 Test C13

Start-Design: `s5x8sm1.txt` (globale Modifikation)

Bézier-Raum-Deformation: Typ 1 (alle Freiheitsgrade)

Testprotokoll-Exzerpt:

```
start merit: 0.345168
dimension: 10
min_merit: 0.0834206
finished degree: 1, 1, 1 at iteration: 186
dimension: 24
min_merit: 0.0823129
finished degree: 2, 2, 1 at iteration: 691
dimension: 40
min_merit: 0.0820297
finished degree: 3, 3, 1 at iteration: 1316
```

Die untenstehende Abbildung gibt den Verlauf der Zielfunktion bei Abschluß der jeweiligen Bézier-Raum-Deformation grafisch wieder.

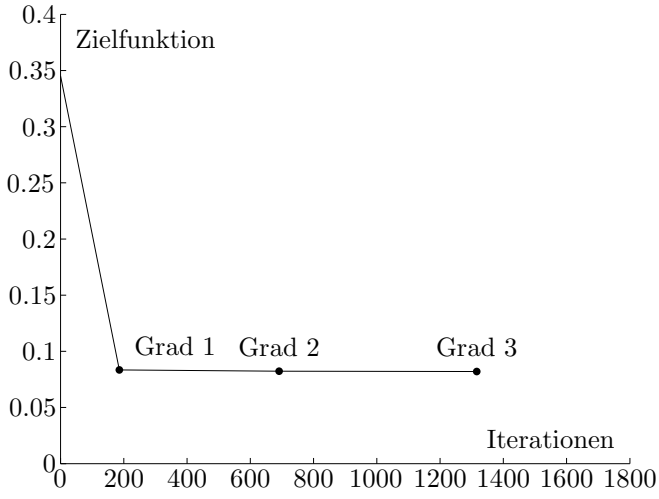


Abbildung 6.4: Verlauf der Zielfunktion bei Test C13

Auffallend bei diesem Test ist die geringe Verbesserung in der Zielfunktion für Grad zwei und drei der Bézier-Raum-Deformation.

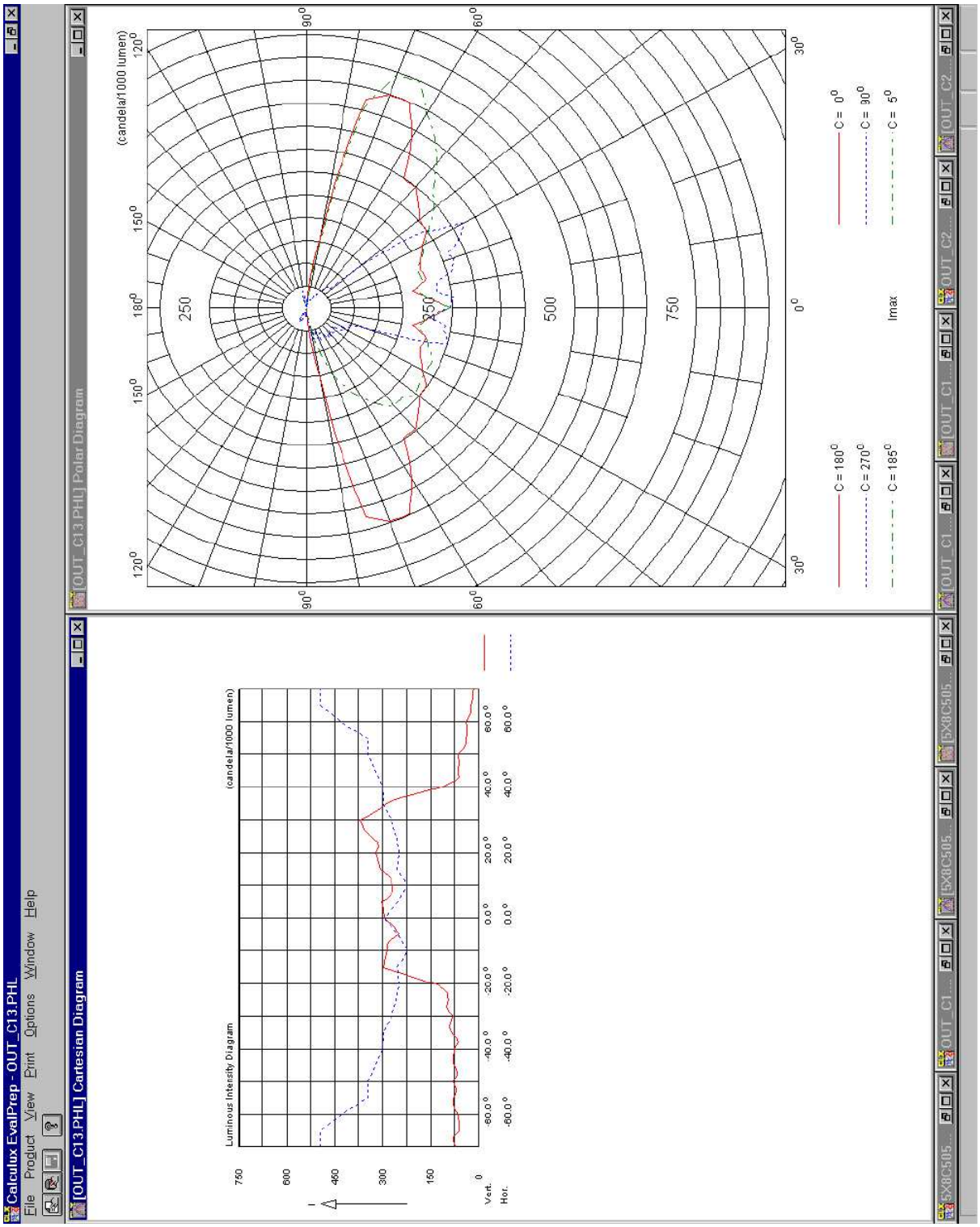


Abbildung 6.5: Lichtverteilung `out_c13.phl`

6.4.2 Test C14

Start-Design: `s5x8sm1.txt` (globale Modifikation)

Bézier-Raum-Deformation: Typ 2 (Freiheitsgrade nur in z-Richtung)

Testprotokoll-Exzerpt:

```
start merit: 0.345168
dimension: 10
min_merit: 0.0833337
finished degree: 1, 1, 1 at iteration: 74
dimension: 24
min_merit: 0.0825114
finished degree: 2, 2, 1 at iteration: 181
dimension: 40
min_merit: 0.082462
finished degree: 3, 3, 1 at iteration: 310
```

Die untenstehende Abbildung gibt den Verlauf der Zielfunktion bei Abschluß der jeweiligen Bézier-Raum-Deformation grafisch wieder.

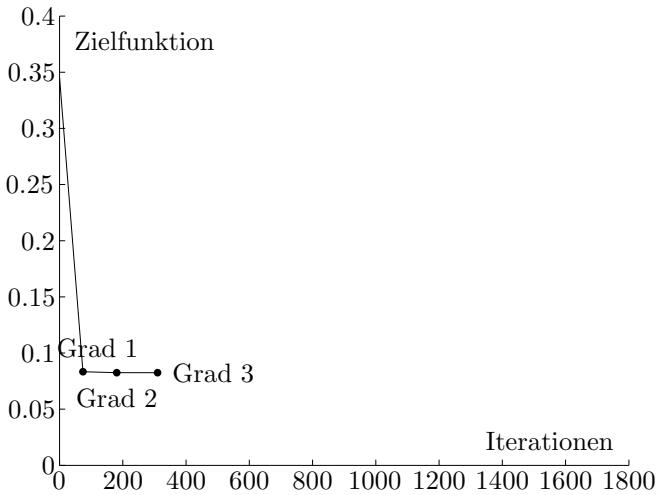


Abbildung 6.6: Verlauf der Zielfunktion bei Test C14

Auffallend bei diesem Test ist die geringe Anzahl der Iterationen und die geringe Verbesserung in der Zielfunktion für Grad zwei und drei der Bézier-Raum-Deformation.

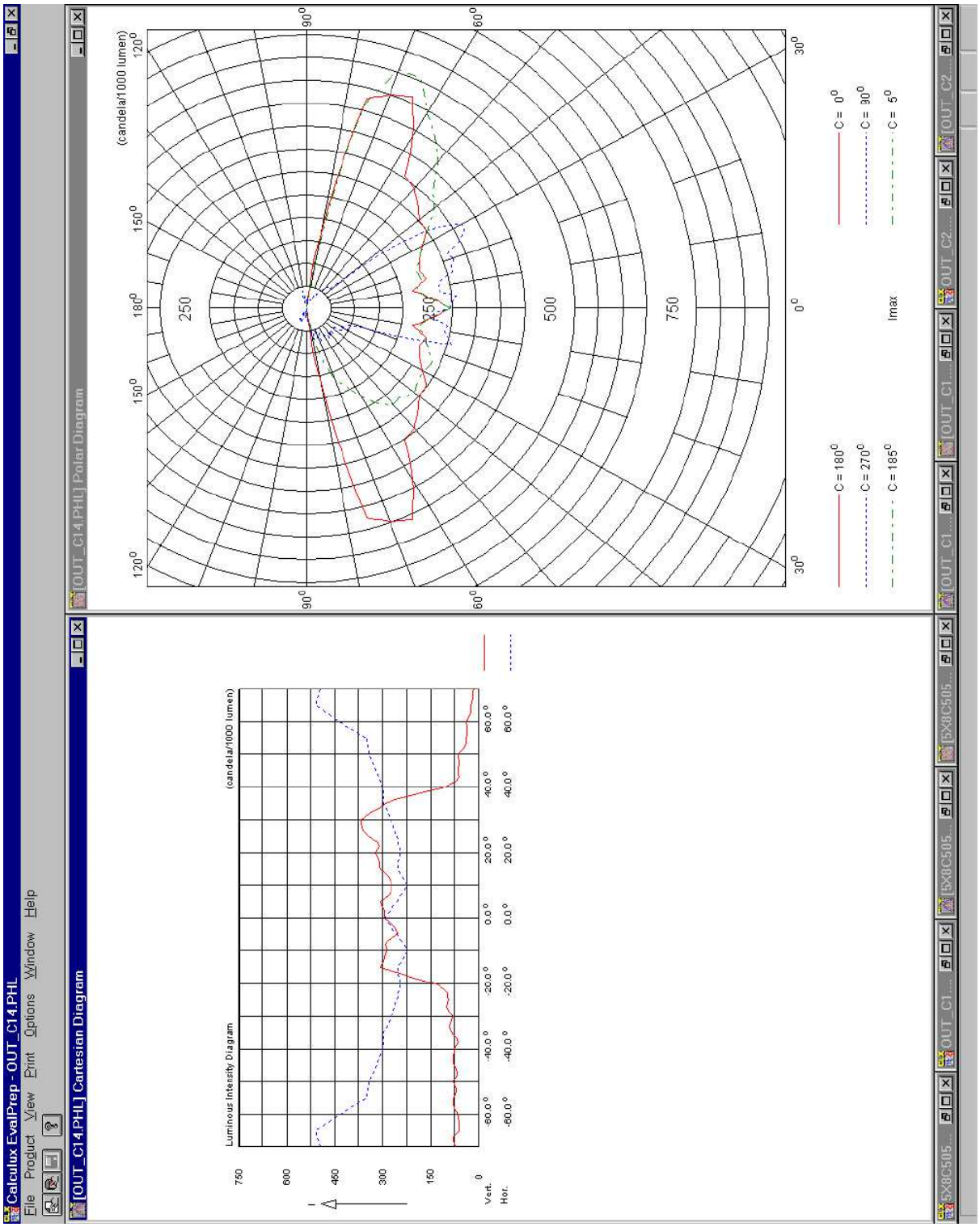


Abbildung 6.7: Lichtverteilung `out_c14.phl`

6.4.3 Test C16

Start-Design: `s5x8sm2.txt` (lokale Modifikation)

Bézier-Raum-Deformation: Typ 1 (alle Freiheitsgrade)

Testprotokoll-Exzerpt:

```
start merit: 0.220732
dimension: 10
min_merit: 0.121978
finished degree: 1, 1, 1 at iteration: 202
dimension: 24
min_merit: 0.0963846
finished degree: 2, 2, 1 at iteration: 638
dimension: 40
min_merit: 0.0859014
finished degree: 3, 3, 1 at iteration: 1630
```

Die untenstehende Abbildung gibt den Verlauf der Zielfunktion bei Abschluß der jeweiligen Bézier-Raum-Deformation grafisch wieder.

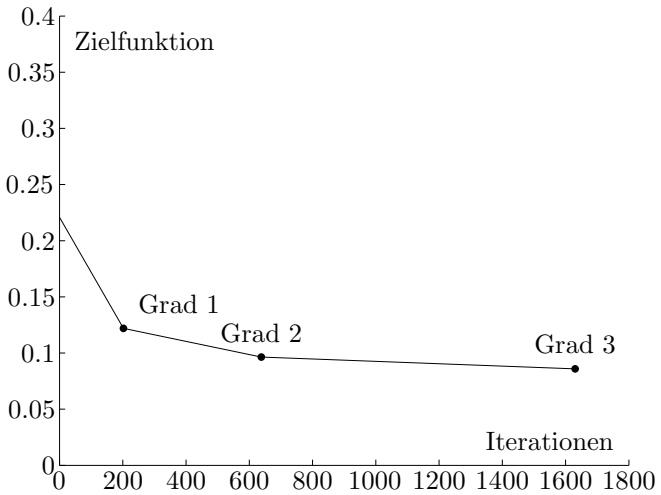


Abbildung 6.8: Verlauf der Zielfunktion bei Test C16

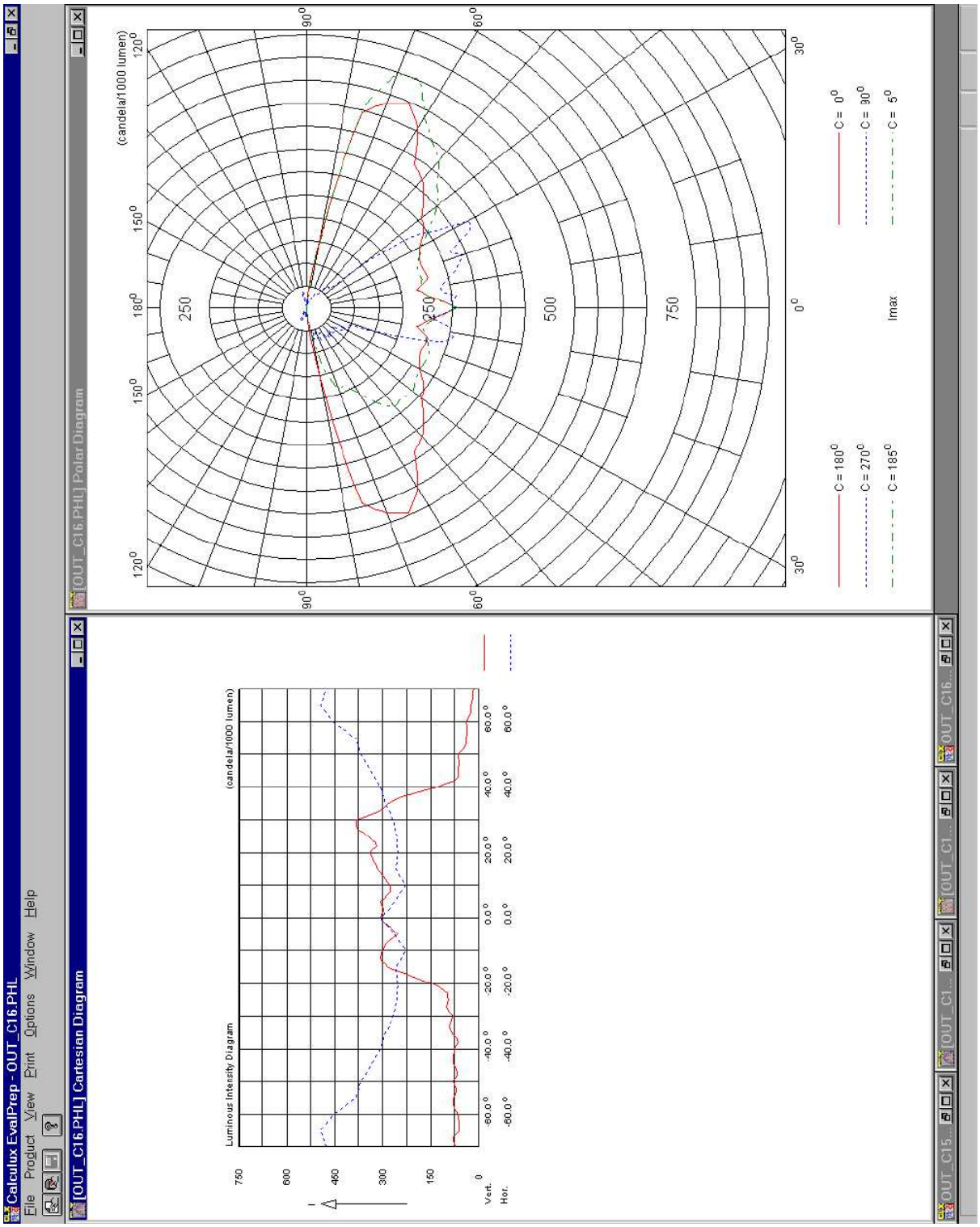


Abbildung 6.9: Lichtverteilung `out_c16.phl`

6.4.4 Test C17

Start-Design: `s5x8sm2.txt` (lokale Modifikation)

Bézier-Raum-Deformation: Typ 2 (Freiheitsgrade nur in z-Richtung)

Testprotokoll-Exzerpt:

```
start merit: 0.220732
dimension: 10
min_merit: 0.136023
finished degree: 1, 1, 1 at iteration: 46
dimension: 24
min_merit: 0.0917655
finished degree: 2, 2, 1 at iteration: 220
dimension: 40
min_merit: 0.0912141
finished degree: 3, 3, 1 at iteration: 364
```

Die untenstehende Abbildung gibt den Verlauf der Zielfunktion bei Abschluß der jeweiligen Bézier-Raum-Deformation grafisch wieder.

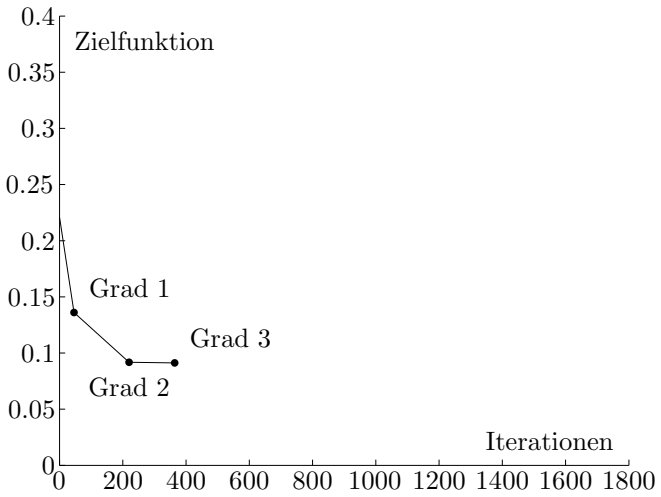


Abbildung 6.10: Verlauf der Zielfunktion bei Test C17

Auffallend bei diesem Test ist die geringe Anzahl der Iterationen und die geringe Verbesserung in der Zielfunktion für Grad drei der Bézier-Raum-Deformation.

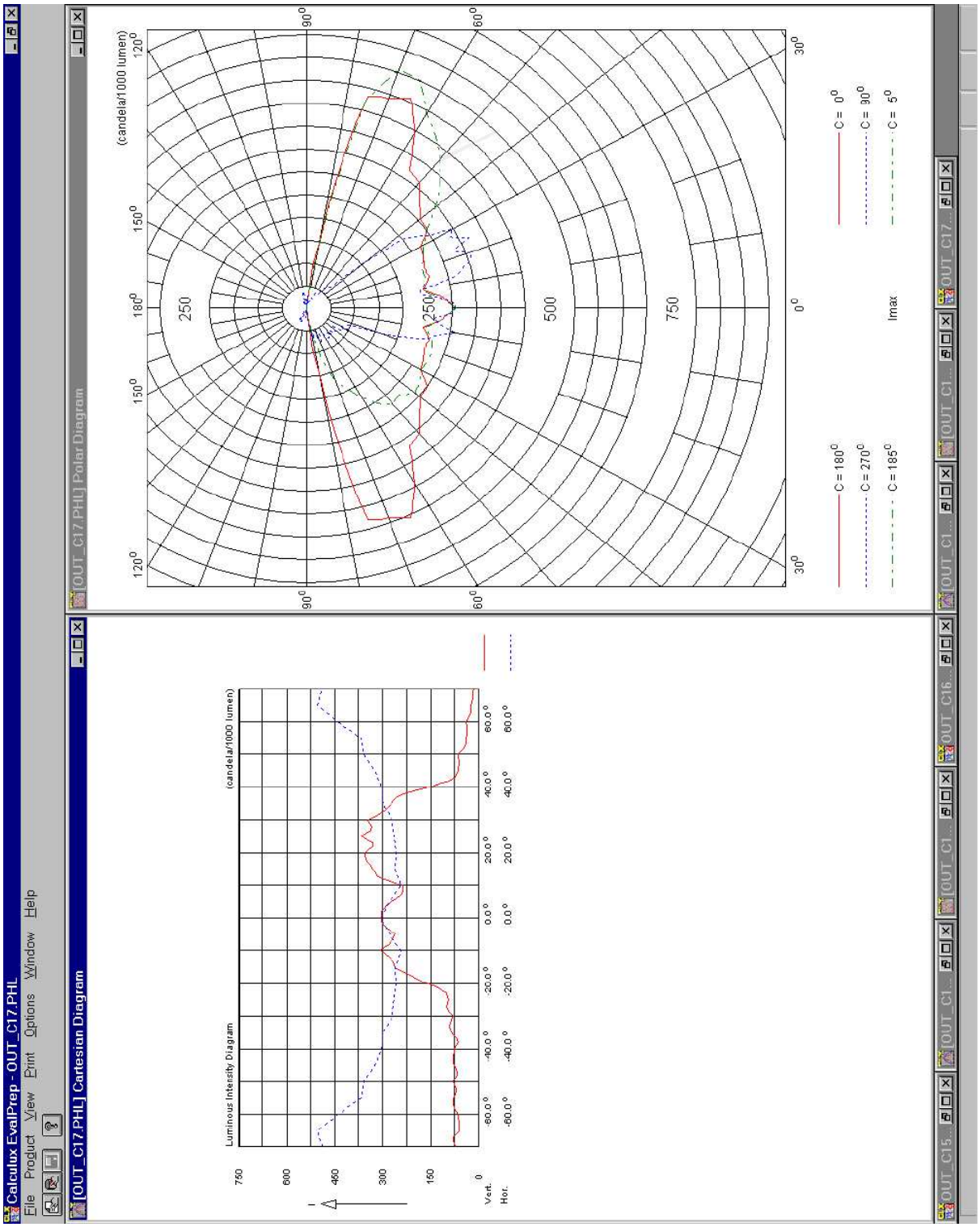


Abbildung 6.11: Lichtverteilung `out_c17.phl`

6.5 Zusammenfassung der Tests

In allen Tests erreichte die Optimierung den erwarteten Wert von knapp unter 0.1 in der Zielfunktion, der – wie ja bereits erwähnt – durch den Fehler in der Umrechnung der Lichtverteilung zwischen den Koordinatensystemen eine natürliche Grenze darstellt. Dieselben Tests wurden übrigens auch mit direkter Vorgabe der Lichtverteilung im A-α-System gemacht, wo die Optimierung die Modifikationen praktisch komplett ausgleichen konnte und Werte von rund 10^{-6} erreichte – obwohl diese Funktionalität eigentlich nicht zu den geforderten Spezifikationen für CALMIR2 gehört.

Es hat sich weiter gezeigt, daß die Optimierung sowohl mit der globalen als auch mit der lokalen Modifikation gleichermaßen zurecht kam. Die Verwendung der Bézier-Raum-Deformation zur Modifikation der Reflektorflächen während der Optimierung ist also der richtige Ansatz für diesen Anwendungsbereich.

Des weiteren ist auffallend, daß die zusätzlichen Freiheitsgrade für die Verschiebung der Kontrollpunkte der Bézier-Raum-Deformation in x- und y-Richtung nur unbedeutend bessere Ergebnisse bringen, jedoch durch die deutlich größere Anzahl an Freiheitsgraden eine deutlich längere Laufzeit bedingen. Daher ist im Normalfall die Bézier-Raum-Deformation vom Typ 2, die nur Freiheitsgrade in z-Richtung hat, derjenigen vom Typ 1, die Freiheitsgrade in jeder Koordinatenrichtung besitzt, vorzuziehen.

Die Tests haben weiterhin gezeigt, daß bereits mit Grad eins die Bézier-Raum-Deformation eine deutliche Verbesserung der Zielfunktion erreicht. Die Graderhöhung auf Grad zwei bringt nur bei der lokal veränderten Reflektorfläche eine merkliche Verbesserung der Zielfunktion, die aber längst nicht so stark ist wie für Grad eins, und aufgrund der größeren Anzahl der Freiheitsgrade auch aufwendiger ist. Bis auf Test C16 erreicht die Bézier-Raum-Deformation mit Grad drei nur geringfügige Verbesserungen in der Zielfunktion, benötigt aber aufgrund der nochmal gestiegenen Anzahl an Freiheitsgraden erheblich mehr Iterationen. Es bleibt die Frage, ob sich dieser zusätzliche Aufwand für Grad drei noch lohnt, oder ob man besser nach Grad zwei direkt abbricht.

Insgesamt haben aber alle Tests gezeigt, daß die gestellten Forderungen an die Optimierung mit CALMIR2 erfüllt werden. Es ist mit diesem Programm nun möglich, vollautomatisch in vernünftiger Zeit – d. h. mit vertretbarem Rechenaufwand – eine initiale Reflektorkonfiguration in Bezug auf eine vorgegebene Lichtverteilung vollautomatisch zu optimieren.

7 Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit sollte ein Programm erstellt werden, das es ermöglicht, vorhandene Reflektorflächen vollautomatisch in Bezug auf eine vorgegebene Lichtverteilung in vernünftiger Zeit zu optimieren. Mit dem Resultat CALMIR2 ist dieses Ziel voll und ganz erreicht worden.

Das Programm stellt eine große Erleichterung für die Lichttechnikingenieure dar, die bisher Modifikationen immer von Hand durch Veränderung einzelner Kontrollpunkte der NURBS-Fläche durchgeführt haben, und anschließend die neue Konfiguration durch einen von Hand gestarteten Raytracerlauf bewerteten, was immer einen großen Aufwand darstellte. Mit CALMIR2 ist es nun durch die Vollautomatisierung möglich, diesen Arbeitsaufwand drastisch zu reduzieren.

Ein Kernstück des Programms ist dabei der schnelle Brick-Layer-Raytracer, der die Bewertung der momentanen Reflektorkonfiguration ermöglicht und dessen Hauptidee die Unterteilung des urprunglichen Begrenzungsquaders in kleinere Quader ist. Wie erwähnt, wird aus Geschwindigkeitsgründen nicht die NURBS-Fläche selber, sondern eine approximative Triangulierung derselben verwendet, mit der Reflexionen von Lichtstrahlen deutlich einfacher zu berechnen sind.

Ein wichtiges Hilfsmittel stellen dabei die ortogonalen B-Splines aus Abschnitt 3.5 dar, mit denen es über den Projektionssatz sehr einfach möglich ist, eine gegebene NURBS-Fläche approximativ zu triangulieren. Der Aufwand gegenüber der linearen Standard-Interpolation ist dabei nur geringfügig größer, während die Approximation nicht nur der Funktionswerte sondern auch der Normalenrichtungen einen deutlichen Vorteil – insbesondere im Hinblick auf Reflexionen an der Fläche – liefert. Falls sich Philips zu einer Fortsetzung dieses Projekts entscheidet, wäre es auch interessant, andere Raum-Deformationen zu untersuchen, z. B. mit Spline- anstatt NURBS-Funktionen.

Das zweite Kernstück des Programms ist der Optimierer, der sich zur Modifikation der bestehenden Reflektor-Konfiguration der Bézier-Raum-Deformation bedient. Die Tests haben gezeigt, daß dieses Verfahren glatte Modifikationen der Flächen erlaubt, wobei sowohl globale als auch begrenzte lokale Änderungen der Flächen zustande kommen, womit also ein großer Teil der möglichen Veränderungen ausgeschöpft werden kann.

Die Tests haben aber auch ergeben, daß mit Grad drei der Bézier-Raum-Deformation zu einem relativ hohen Aufwand nur noch geringfügige Verbesserungen der Zielfunktion erreicht werden. Es bleibt die Frage, ob sich dieser Aufwand noch lohnt, oder ob die Optimierung besser direkt nach Grad zwei abgebrochen wird. In diesem Fall könnte man in Zukunft auch mehrere Iterationen der Bézier-Raum-Deformation mit Grad zwei hintereinander ausprobieren.

Ein Problem, das bei diesem Programm auftrat, war die Tatsache, daß die vorgegebene Ziel-Lichtverteilung in einem anderen Koordinatensystem beschrieben wird als dasjenige, in dem der Raytracer läuft, sowie die von Philips vorgegebenen Umrechnungsroutinen

zwischen diesen Koordinatensystemen. Bei einer Fortsetzung sollte unbedingt versucht werden, den Raytracer in dem Koordinatensystem zu betreiben, in dem auch die Ziel-Lichtverteilung gegeben ist. Die Tests haben gezeigt, daß in diesem Fall der Optimierer um Größenordungen besser arbeitet. Falls jedoch der bisherige Aufbau beibehalten wird, sollten zumindest andere Umrechnungsroutinen verwendet werden. Evtl. würde auch eine Vorbearbeitung der Ziel-Lichtverteilung, wie z. B. eine Glättung, das Problem entschärfen.

Wie in der Einleitung bereits erwähnt, wurde schon bald der ursprünglich vorgesehene Beamtracer, der Lichtstrahlen als Kegel betrachtet, durch das universellere Konzept eines Raytracers ersetzt, der Lichtstrahlen als Geraden behandelt. Ein Vorteil des Raytracers ist dabei, daß er auch Brechungen an Optiken berechnen kann, während dies mit einem Beamtracer praktisch nicht zu bewerkstelligen ist. Erweitert man noch das Modell der Lichtquelle von einem Lampenzylinder zu einer Punktlichtquelle, so würden sich mit diesen beiden Änderungen ganz neue Einsatzmöglichkeiten für CALMIR2 ergeben, wie z. B. die Optimierung von Scheinwerfern.

Schließlich wäre es auch noch interessant, andere Optimierungs-Algorithmen zu testen, die durch die Matlab-Schnittstelle sehr einfach eingebunden werden können. Vielversprechend für dieses Optimierungs-Problem sind hierbei die sogenannten genetischen Algorithmen, die in letzter Zeit immer populärer werden ([DC99], [Haj99], [PS97a], [PS97b]).

Die Implementierung von CALMIR2 als POSIX-kompatibles C++-Programm erleichtert einerseits die Einbindung solcher neuer Erweiterungen deutlich, andererseits sollte dadurch auch die Portierung auf andere und neuere Betriebssysteme gewährleistet sein. Damit kann das Programm auch auf neuerer Hardware, die ja naturgemäß immer leistungsfähiger wird, verwendet werden, was die Anwendungsmöglichkeiten dieses Programms weiter vergrößert.

A Programmcode

Insgesamt umfaßt CALMIR2 über 20000 Zeilen Quellcode, wobei die Implementierung jedoch dem Copyright von Philips unterliegt. Deshalb können hier nur die Schnittstellen, d. h. die Header-Dateien aufgeführt werden. Innerhalb dieser mussten allerdings auch die `private:`-Sektionen sowie die Implementierung der `inline`-Funktionen ausgeblendet werden. Da die Projekt-Sprache Englisch war, sind auch im Quellcode alle Kommentare in englisch gehalten.

Um den Source-Code von CALMIR2 leichter zu identifizieren, beginnen alle Dateien mit dem Prefix `c2`. Jede Klasse der Implementierung verwendet dabei ihre eigene `.cc`-Datei sowie die zugehörige gleichnamige Header-Datei. Diese umfassen:

- `c2box`
- `c2boxtree`
- `c2graphic`
- `c2hookejeeves`
- `c2inout`
- `c2lamp`
- `c2matlab`
- `c2optimizer`
- `c2ray`
- `c2raytracer`
- `c2triangle`
- `c2triangulation`
- `c2vertex`

Zusätzlich existiert die Datei `c2main.h` als globale Header-Datei und `c2main.cc` als Einsprungpunkt des Programms mit der Main-Loop.

A.1 c2main.h

Dies ist die einzige Header-Datei, die nicht zu einer Klasse gehört, sondern zum Einsprungpunkt des Programms selbst und die Default-Werte für globale Variablen enthält. Des weiteren werden hier der `start-` und `exitcallback` für die Display-Bibliothek deklariert.

```
// current version of the program
#define VERSION "4.1"

// maximum allowed error of the triangulation (default)
#define DEFAULT_MAX_ERROR (double)1.0

// reflection factor (default)
#define DEFAULT_REFLECTION_FACTOR (double)1.0

// the maximum length of the graphic window title string
#define TITLE_STR_LEN 1024

// entry point for the program
int main(int, char**);

// parse the arguments on the commandline
void c2args(int, char**);

// display start call back function
void c2startcallback(void*);

// display exit call back function
void c2exitcallback(void*);

// main loop without display
void c2mainloop(void);
```

A.2 c2box.h

In dieser Klasse ist ein Quader für den Brick-Layer-Algorithmus implementiert.

```
class c2box
{
 public:
  // default constructor
```

```
  inline c2box(void);
  // constructor with minimum and maximum corner
  inline c2box(bxCoord3, bxCoord3);
  // destructor
  virtual ~c2box();

  // get the index of this box
  inline void get_index(int&, int&, int&);
  // set the index of this box
  inline void set_index(int, int, int);

  // get the corners of the box
  inline void get_corners(bxCoord3&, bxCoord3&);
  // set the corners of the box
  inline void set_corners(bxCoord3, bxCoord3);

  // get the min/max bound in x/y/z-direction
  inline double get_bound(int, int);
  // set the min/max bound in x/y/z-direction
  inline void set_bound(int, int, double);

  // get the min and max bounds of the box
  // x_min, y_min, z_min, x_max, y_max, z_max
  inline void get_bounds(double&, double&, double&, double&, double&, double&);
  // set the min and max bounds of the box
  // x_min, y_min, z_min, x_max, y_max, z_max
  inline void set_bounds(double, double, double, double, double, double);

  // get the number of triangles in this box
  inline int get_num_triangles(void);
  // add the triangle to the list of triangles in this box
  inline void add_triangle(c2triangle*);
  // intersect this ray with all triangles in this box
  int intersect_ray(c2ray&, double);
};
```

A.3 c2boxtree.h

In dieser Klasse werden alle Quader des Brick-Layer-Algorithmus verwaltet, sowie die Lichtstrahlen auf ihrem Weg durch diese Quader verfolgt.

```
class c2boxtree
{
```

```
public:
 // default constructor
 inline c2boxtree(void);
 // constructor with lower and upper corner
 inline c2boxtree(bxCoord3, bxCoord3);
 // destructor
 virtual ~c2boxtree(void);

 // get the lower/upper corner
 inline bxCoord3 get_corner(int);
 // set the lower/upper corner
 inline void set_corner(int, bxCoord3);
 // get both corners at once
 inline void get_corners(bxCoord3&, bxCoord3&);
 // set both corners at once
 inline void set_corners(bxCoord3, bxCoord3);

 // get the number of boxes in int x/y/z-direction
 inline int get_num_boxes(int);
 // set the number of boxes in int x/y/z-direction to int
 inline void set_num_boxes(int, int);
 // get the number of boxes in all directions at once
 inline void get_num_boxes(int&, int&, int&);
 // set the number of boxes in all directions at once
 inline void set_num_boxes(int, int, int);

 // get the bounds of the boxtree (lower/upper)
 inline void get_bounds(bxCoord3&, bxCoord3&);
 // set the bounds of the boxtree (lower/upper)
 inline void set_bounds(bxCoord3, bxCoord3);

 // get the length of one box (x/y/z)
 inline void get_box_length(double&, double&, double&);
 // set the length of one box (x/y/z)
 inline void set_box_length(double, double, double);

 // compute the bounds of the boxtree
 void compute_boundingbox(void);

 // compute the subboxes
 void compute_subboxes(void);

 // add the triangles to the list in the appropriate boxes
 void compute_triangles(void);
```

```
  // trace a ray through the boxtree
  int trace_ray(c2ray&);
};
```

A.4 c2graphic.h

Diese Klasse dient zur grafischen Darstellung der geometrischen Primitiven und ist von der Klasse S_Graphic der NetzDatStrukt-Bibliothek abgeleitet.

```
class c2graphic: public S_Graphic
{
 public:
  // default constructor
  inline c2graphic(void);
  // destructor
  virtual ~c2graphic(void);

  // draw a bxNURBSSurface
  void draw(bxNURBSSurface *, int =10, float =1, float =1, float =1, int =0);
};
```

A.5 c2hookejeeves.h

In diesem Modul ist der Optimierungs-Algorithmus von Hooke-Jeeves implementiert.

```
class c2hookejeeves
{
 public:
  // default constructor
  c2hookejeeves(void);
  // destructor
  virtual ~c2hookejeeves(void);

  // set int lower and upper bounds
  void set_bounds(double*, double*, int);

  // get the actual minimum merit
  double get_min_merit(void);

  // the optimization routine itself with
```

```
  // int starting points/results and step lengths and accuracy
  void optimize(double*&, double*, int, double);
};
```

A.6 c2inout.h

Diese Klasse dient zur Verwaltung aller Datei-Ein- und -Ausgaben, zusammen mit den Umrechnungsroutinen für die Ein- und Ausgabe.

```
class c2inout
{
 public:
  //default constructor
  inline c2inout(void);
  // destructor
  virtual ~c2inout(void);

  // read the NURBS surfaces
  void read_surfaces(void);
  // smooth the surface, so that the control points are arranged on
  // a set of prallel plains
  void smooth_surfaces(void);
  // compute the bounding box of the surfaces
  void compute_surface_bounds(void);

  // write out the resulting surface
  void write_surface(void);

  // write the display header file
  void write_display_header(void);

  // read the optional weight matrix
  void read_weight_matrix(void);

  // read the input target light distribution
  void read_target_intensity(void);
  // convert the target intensity matrix from C-, gamma- to
  // A-, alpha- or B-, beta-system
  void convert_target_intensity(void);

  // convert the resulting intensity matrix from A-, alpha- or
  // B-, beta- to C-, gamma-system
  void convert_intensity(void);
```

```
  // write the intensity matrix to the output file
  void write_intensity(void);
};
```

Während des Programmablaufs werden diese Routinen in der folgenden Reihenfolge aufgerufen:

- `read_weight_matrix` von `main` (optional)
- `read_target_intensity` von `main`
- `convert_target_intensity` von `main`
- `read_surfaces` von `main`
- `smooth_surfaces` von `main`
- `compute_surface_bounds` von `main`
- `write_display_header` von `main`
- `convert_intensity` von `c2exitcallback`
- `write_intensity` von `c2exitcallback`
- `write_surface` von `c2exitcallback`

A.7 c2lamp.h

In dieser Klasse sind die Lampe und die Befestigungszylinder implementiert.

```
class c2lamp
{
 public:
  //default constructor
  inline c2lamp(void);
  // destructor
  virtual ~c2lamp(void);

  // get the center of the lamp
  inline bxCoord3 get_center(void);
  // set the center of the lamp
  inline void set_center(bxCoord3);

  // get the x-axis of the lamp
```

```
  inline bxCoord3 get_xaxis(void);
  // get the y-axis of the lamp
  inline bxCoord3 get_yaxis(void);
  // get the z-axis of the lamp
  inline bxCoord3 get_zaxis(void);

  // get the direction of the axis of the lamp
  inline bxCoord3 get_direction(void);
  // set the direction of the axis of the lamp
  void set_direction(bxCoord3);

  // get the length of the lamp
  inline double get_length(void);
  // set the length of the lamp
  inline void set_length(double);

  // get the diameter of the lamp
  inline double get_diameter(void);
  // set the diameter of the lamp
  inline void set_diameter(double);

  // get the flux of the lamp
  inline double get_flux(void);
  // set the flux of the lamp
  inline void set_flux(double);

  // get the bounding value lower/upper x/y/z
  inline double get_bound(int, int);
  // get all the bounding values at once
  inline void get_bounds(bxCoord3&, bxCoord3&);
  // compute the bounding box
  inline void compute_bounding_box(void);

  // draw the lamp itself
  void draw(int =1, int =1, int =1, int =0);
};
```

A.8 c2matlab.h

In dieser Klasse ist die Schnittstelle zum Matlab-Programm implementiert, über die in Matlab geschriebene Algorithmen als Optimierer verwendet werden können.

```
class c2matlab
```

```
{
 public:
  // default constructor
  c2matlab(void);
  // destructor
  virtual ~c2matlab(void);

  // set int constraints and int lower and upper bounds
  void set_constraints_bounds(double*, double*, int, double*, double*, int);

  // get the actual minimum merit
  double get_min_merit(void);

  // the optimization routine fmincon from matlab itself with
  // starting point/result
  void optimize(double*&);
};
```

A.9 c2optimizer.h

Diese Klasse dient zur Verwaltung des Optimier-Algorithmus. Von ihr werden wahlweise `c2hookejeeves` mit dem Hooke-Jeeves-Algorithmus oder `c2matlab` mit einem Matlab-Optimierer aufgerufen.

```
class c2optimizer
{
 public:
  // default constructor
  c2optimizer(void);
  // destructor
  virtual ~c2optimizer(void);

  // get the weight of the integral in the merit function
  double get_weight(void);
  // set the weight of the integral in the merit function
  void set_weight(double);

  // get the optimization mode
  int get_mode(void);
  // set the optimization mode
  void set_mode(int);

  // set the optimization algorithm
```

```
void set_algorithm(int);

// get the degrees of the bezier space deformation
void get_degrees(int&, int &, int &);

// get the number of iterations
int get_iteration(void);
// set the number of iterations
void set_iteration(int);

// get the number of iterations after the surface is drawn
int get_draw_iteration(void);
// set the number of iterations after the surface is drawn
void set_draw_iteration(int);

// draw the axis and the labels in the error window
void draw_error_axis(void);

// compute the flux of the target light distribution
void compute_target_flux(void);

// compute the difference between the target and the achieved
// values (merit function)
double merit(double*);

// the optimization routine itself
void optimize(void);

// check constraints for double[int] of the optimization algorithm
void constraint(double*&, int);

// evaluate the surfaces for the given degree and set of three dimensional
// bezier control points
void bezier_eval();
// evaluate the surfaces for the given degree, knots and set of three
// dimensional spline control points
void spline_eval();

// update the lower z bound, if the surface increased
void update_lower_z_bound();

// convert the control points of a NURBS surface
// into variables for the optimization algorithm
void convert_variables(bxNURBSSurface, double*&);
```

```
  // convert the variables for the optimization algorithm
  // into control points of a NURBS surface
  void convert_variables(double*, bxNURBSSurface&);
  // convert the control points of the space deformation
  // into variables for the optimization algorithm
  void convert_variables(bxCoord3***, double*&);
  // convert the variables for the optimization algorithm
  // into control points of the space deformation
  void convert_variables(double*, bxCoord3***&);

  // initialize the hooke jeeves algorithm
  void init_hookejeeves(void);
#ifdef MATLAB
  // initialize the matlab optimization algorithm
  void init_matlab(void);
#endif
};
```

A.10 c2ray.h

In dieser Klasse ist ein Lichstrahl, wie er für den Raytracer benötigt wird, implementiert.

```
class c2ray
{
 public:
  // default constructor
  inline c2ray(void);
  // constructor with starting point and direction (must be nonzero!),
  // double intensity and int number of reflections
  c2ray(bxCoord3, bxCoord3, double, int =0);
  // destructor
  virtual ~c2ray(void);

  // get the starting point
  inline bxCoord3 get_start(void);
  // set the starting point
  inline void set_start(bxCoord3);
  // add bxCoord3 to the starting point
  inline void add_start(bxCoord3);

  // get the direction
  inline bxCoord3 get_direction(void);
  // set the direction
```

```
  inline void set_direction(bxCoord3);
  // add bxCoord3 to the direction
  inline void add_direction(bxCoord3);

  // get the intensity of the ray
  inline double get_intensity(void);
  // set the intensity of the ray
  inline void set_intensity(double);
  // multiply the intensity with double
  inline void mul_intensity(double);

  // get the number of reflections
  inline int get_num_reflections(void);
  // set the number of reflections
  inline void set_num_reflections(int);
  // add int to the number of reflections
  inline void add_num_reflections(int =1);

  // get angular values in A-, alpha- or B-, beta system
  void get_angular(double&, double&);
};
```

A.11 c2raytracer.h

In dieser Klasse ist der Raytracer, wie er für den Brick-Layer-Algorithmus verwendet wird, implementiert.

```
class c2raytracer
{
 public:
  // default constructor
  inline c2raytracer(void);
  // destructor
  virtual ~c2raytracer(void);


  // initialize the raytracer
  void init(void);

  // reset the raytracer
  void reset(void);

  // get the lowest percentage of initial ray flux
```

```
  inline double get_min_flux(void);
  // set the lowest percentage of initial ray flux
  inline void set_min_flux(double);

  // get the maximum number of intersections
  inline int get_num_intersections(void);
  // set the maximum number of intersections
  inline void set_num_intersections(int);

  // get the raytracing mode
  inline int get_mode(void);
  // set the raytracing mode
  inline void set_mode(int);

  // get the number of rays
  inline int get_num_rays(void);
  // set the number of rays
  inline void set_num_rays(int);

  // get the aposization factor (only mode == 0)
  inline double get_aposization(void);
  // set the aposization factor (only mode == 0)
  inline void set_aposization(double);

  // get the number of sectors (only mode == 1)
  inline int get_num_sectors(void);
  //set the number of sectors (only mode == 1)
  inline void set_num_sectors(int);

  // get the number of slices (only mode == 1)
  inline int get_num_slices(void);
  // set the number of slices (only mode == 1)
  inline void set_num_slices(int);

  // do the raytracing
  void raytrace(void);

  // trace a ray
  void trace_ray();
};
```

A.12 c2triangle.h

Diese Klasse repräsentiert ein Dreieck der Triangulierung, abgeleitet von der Klasse `Face` der NetzDatStrukt-Bibliothek.

```
class c2triangle: public Face
{
 public:
  // constructor from three vertices and number of the surface/triangulation,
  // this triangle belongs to
  c2triangle(c2vertex*, c2vertex*, c2vertex*, int,
             position =UNKNOWN, bool =true, bool =true);
  // destructor (virtual for heredity)
  virtual ~c2triangle(void);

  // get the corner vertice no. int
  inline c2vertex* get_corner(int);
  // set the corner vertice no. int
  inline void set_corner(int, c2vertex*);

  // get the no. of the corner c2vertex*
  int get_no_of_corner(c2vertex*);

  // get the bounding values lower/upper x/y/z
  inline double get_bound(int, int);
  // get all the bounding values at once
  inline void get_bounds(bxCoord3&, bxCoord3&);
  // compute the bounding box
  void compute_bounding_box(void);

  // get the midpoint no. int
  inline c2vertex* get_midpoint(int);
  // set the midpoint no. int to c2vertex*
  inline void set_midpoint(int, c2vertex*);
  // get all the midpoints
  inline void get_midpoints(c2vertex*&, c2vertex*&, c2vertex*&);
  // set all the midpoints
  inline void set_midpoints(c2vertex*, c2vertex*, c2vertex*);

  // get the normal
  inline bxCoord3 get_normal(void);
  // compute the normal
  void compute_normal(void);
```

```
// get the edge no. int
inline pEdge get_edge(int);
// update the edge pointers
inline void update_edges(void);

// get neighbour no. int
inline c2triangle* get_neighbour(int);
// set the neighbour pointer no. int
inline void set_neighbour(int, c2triangle*);
// get all the neighbours
inline void get_neighbours(c2triangle*&, c2triangle*&, c2triangle*&);
// set all the neighbours
inline void set_neighbours(c2triangle*, c2triangle*, c2triangle*);

// get the number of the surface this triangle belongs to
inline int get_num_surf(void);
// set the subdivision level
inline void set_num_surf(int);

// get the subdivision level
inline int get_subdiv_level(void);
// set the subdivision level
inline void set_subdiv_level(int);

// set subdiv_common_corner to int
inline void set_subdiv_corner(int);
// set subdiv_common_corner
inline int get_subdiv_corner(void);

// get the subdivision neighbour
inline int get_subdiv_neighbour(void);
// set the subdivision neighbour to int
inline void set_subdiv_neighbour(int);

// subdivide in 2 or 4 triangles
void subdivide();
// subdivide in 2 or 4 triangles with c2triangle neighbours
// in int direction to level int
int subdivide(c2triangle*, c2triangle*, int, int);

// intersect triangle with c2ray only if distance is less then double
// away
// return values:
// -1: ray absorbed by the backside
```

```
  //  0: no intersection
  //  1: intersection in double& distance (c2ray& is updated)
  int intersect_ray(c2ray, c2ray&, double&);

  // get the reflection factor
  inline double get_reflection_factor(void);
  // set the reflection factor
  inline void set_reflection_factor(double);
  // compute the reflection factor for this triangle
  void compute_reflection_factor(void);

  // maximum distance between the corners/midpoints and the surface
  double error(void);

  // statistical output on the screen and on stdout
  // for debugging purposes
  void statistics(void);

  // recompute the position of the corner points
  void update_corners(void);
};
```

A.13 c2triangulation.h

Diese Klasse dient zur Verwaltung der triangulierten Approximation einer NURBS-Fläche und ist von der Klasse `Net` der NetzDatStrukt-Bibliothek abgeleitet.

```
class c2triangulation: public Net
{
 public:
  // default constructor
  inline c2triangulation(void);
  // destructor (virtual for heredity)
  virtual ~c2triangulation(void);

  // initialization of the triangulation with surface number int
  void init(int);

  // subdivsion of the triangulation
  void subdivide(void);

  // get the bounding value
  inline double get_bound(int, int);
```

```
  // get all the bounding values at once
  inline void get_bounds(bxCoord3&, bxCoord3&);

  // compute the bounding boxes of the triangles and
  // of the triangulation
  void compute_bounding_box(void);

  // compute the reflection factors in each triangle
  void compute_reflection_factors(void);

  // checks the position of the corners
  void check_corners(void);
};
```

A.14 c2vertex.h

Diese Klasse repräsentiert einen Knoten in der Triangulierung, abgeleitet von der Klasse `Vertex` der NetzDatStrukt-Bibliothek.

```
class c2vertex: public Vertex
{
 public:
  // constructor from three doubles
  inline c2vertex(double, double, double, position =UNKNOWN, bool =false);
  // destructor
  virtual ~c2vertex(void);

  // set the x-, y-, z-coordinates
  inline void set_val(bxCoord3);
  // set the x-, y-, z-coordinates
  inline void set_val(double, double, double);

  // get the u-, v-coordinates
  inline double get_coord(int);
  // set the u-, v-coordinates
  inline void set_coord(int, double);
  // get the u-, v-coordiantes at once
  inline void get_coords(double&, double&);
  // set the u-, v-coordiantes at once
  inline void set_coords(double, double);

  // get the surface value
  inline bxCoord3 get_surf(void);
```

```
  // set the surface value
  inline void set_surf(bxCoord3);

  // checks the actual surface[i] values
  void check_surf(int);

  // get the index of the vertex
  inline int get_index(void);
  // set the index of the vertex
  inline void set_index(int);

  // get the number of triangles
  inline int get_num_triangles(void);
  // set the number of triangles
  inline void set_num_triangles(int);

  // get the temporary number of triangles
  inline int get_tmp_num_triangles(void);
  // set the temporary number of triangles
  inline void set_tmp_num_triangles(int);
  // checks the number of triangles
  inline void check_num_triangles(void);

  // sets the temporary position for the check
  inline void set_tmp_positions(double, double, double);
  // gets the temporary position for the check
  inline void get_tmp_positions(double&, double&, double&);

  // distance between this point and the surface
  inline double dist(void);

  // checks the position of the vertex
  void check_position(void);
};
```

Literatur

[App98] App, Andreas: *Approximative Triangulierung digitaler Laserscannerdaten.* Diplomarbeit, Universität Stuttgart, 1998.

[App00a] App, Andreas: *Statusreport CALMIR2 Version 2.5.* Mathematisches Institut A, Universität Stuttgart, Oktober 2000.

[App00b] App, Andreas: *Testreport CALMIR2 Version 2.5.* Mathematisches Institut A, Universität Stuttgart, Oktober 2000.

[App01a] App, Andreas: *CALMIR2 Version 3.1 Algorithms.* Mathematisches Institut A, Universität Stuttgart, Februar 2001.

[App01b] App, Andreas: *CALMIR2 Version 3.1 Concepts.* Mathematisches Institut A, Universität Stuttgart, Februar 2001.

[App01c] Apprich, Christian: *Der Penalty-Ansatz bei Spline-Galerkin-Approximation des Poisson-Problems.* Diplomarbeit, Math.Inst.A, Universität Stuttgart, 2001.

[Boh99] Bohl, Holger Frank Otto: *Kurven minimaler Energie auf getrimmten Flächen.* Doktorarbeit, Mathematisches Institut A der Universität Stuttgart, 1999.

[Brä92] Bräss, Dietrich: *Finite Elemente.* Springer-Verlag, 1992.

[BSW83] Bank, R. E., A. H. Sherman und A. Weiser: *Some refinement algorithms and data structures for regular local mesh refinement.* In: Stepleman, R. et al. (Herausgeber): *Scientific Computing*, Seiten 3–17. IMACS/North-Holland, Amsterdam, 1983.

[D+86] Dongarra, Jack J. et al.: *An Extended Set of Fortran Basic Linear Algebra Subprograms.* Technischer Bericht 41, Argonne National Laboratory, Mathematics and Computer Science Division, 1986.

[D+88] Dongarra, Jack et al.: *A Set of Level 3 Basic Linear Algebra Subprograms.* Technischer Bericht 1, Argonne National Laboratory, Mathematics and Computer Science Division, August 1988.

[DC99] Doyle, Steven und David Corcoran: *Automated mirror design using an evolution.* Optical Engineering, 38(2):323–333, February 1999.

[Fuc99] Fuchs, Alexander: *Optimierte Delaunay-Triangulierung zur Vernetzung getrimmter NURBS-Körper.* Doktorarbeit, Universität Stuttgart, Germany, 1999.

[Gau03] Gaukel, Joachim: *Effiziente Lösung polynomialer und nichtpolynomialer Gleichungssysteme mit Hilfe von Subdivisionsalgorithmen.* Doktorarbeit, TU Darmstadt, 2003.

[Gre95] Greiner, Horst: *Beam-Tracing Algorithms for Fast 3D Reflector Design.* Philips Forschungslaboratorien, D 52021 Aachen, 1995.

[Gre99] Greiner, Horst: *Fast Raytracing Algorithms for Reflector like Surfaces, Wedding Cake and Brick Layer.* Philips Forschungslaboratorien, D-52066 Aachen, 2. Auflage, Juni 1999.

[Haj99] Hajela, Prabhat: *Nongradient Methods in Multidisciplinary Design Optimization - Status and Potential.* Journal of Aircraft, 36(1):255–265, 1999.

[HH71] Hoffmann, U. und H. Hofmann: *Einführung in die Optimierung.* Verlag Chemie GmbH, Weinheim, 1971.

[Höl98] Höllig, Klaus: *Grundlagen der Numerik.* MathText, Zavelstein, 1998.

[Höl03] Höllig, Klaus: *Finite Element Methods with B-Splines.* SIAM, 2003.

[Jos96] Josuttis, Nicolai: *Objektorientiertes Programmieren in C++.* Addison-Wesley, 1996.

[Koc93] Koch, Jürgen: *C++ Matrix–Klasse Version 1.0.* Technischer Bericht, Universität Stuttgart, Mathematisches Institut A, 1993.

[Koc94] Koch, Jürgen: *Splinefunktionen Version 2.0.* Technischer Bericht, Universität Stuttgart, Mathematisches Institut A, Oktober 1994.

[Kop00a] Kopka, H.: *LaTeX- Einführung*, Band 1. Addison Wesley, 3. Auflage, 2000.

[Kop00b] Kopka, H.: *LaTeX- Ergänzung*, Band 2. Addison Wesley, 3. Auflage, 2000.

[Kop00c] Kopka, H.: *LaTeX- Erweiterungen*, Band 3. Addison Wesley, 3. Auflage, 2000.

[Kra97] Kraft, Arndt: *Eine Einführung in Display, Version 3.3.* Report 97-2, Universität Stuttgart, Mathematisches Institut A, 1997.

[L+79] Lawson, C.L. et al.: *Basic Linear Algebra Subprograms for Fortran Usage.* ACM Transactions on Mathematical Software, 5(3):309–325, 1979.

[Lip91] Lippman, Stanley B.: *C++: Einführung und Leitfaden.* Addison-Wesley, 1991.

[Mat05] The MathWorks, Inc., `http://www.mathworks.com/`: *The MathWorks*, 2005.

[MS05] Mittelmann, H. D. und P. Spellucci: *Decision Tree for Optimization Software.* `http://plato.asu.edu/guide.html`, 2005.

[Neta] The Netlib, `http://www.netlib.org/blas/`: *BLAS.*

[Netb] The Netlib, `http://www.netlib.org/lapack/`: *LAPACK.*

[NF] Nowottny, Dietrich und Alexander Fuchs: *Eine Datenstruktur für FE-Netze*. Noch nicht veröffentlicht.

[Now99] Nowottny, Dietrich: *Netzerzeugung durch Gebietszerlegung und duale Graphmethode*. Doktorarbeit, Universität Stuttgart, 1999.

[OSR05] OSRAM, `http://www.osram.de/service_corner/lichtlexikon/`: *OSRAM Lichtlexikon*, 2005.

[Phi96] Philips: *PH3D V4.0 Concepts*, 2. Auflage, Juli 1996.

[Phi97] Philips: *PH3D V4.0 Calculation Algorithms*, 4. Auflage, März 1997.

[Phi99] Philips: *Phillum, Philips Lighting's standard file format for the electronic transfer of luminaire photometric data, Version 1.3*, 1999.

[PS97a] Price, Kenneth und Rainer Storn: *Differential Evolution*. Dr. Dobbs Journal, 22:18–24, April 1997.

[PS97b] Price, Kenneth und Rainer Storn: *Differential Evolution - A Simple and Efficient Heuristic for Global Optimization over Continuous Spaces*. Journal of Global Optimization, 11:341–359, 1997.

[Rei97] Reif, Ulrich: *Orthogonality of Cardinal B-Splines in Weighted Sobolev Spaces*. SIAM Journal on Mathematical Analysis, 28(5):1258–1263, 1997.

[Riv84] Rivara, M. C.: *Algorithms for refining triangular grid suitable for adaptive and multigrid techniques*. International Journal for Numerical Methods in Engineering, 20:745–756, 1984.

[RS75] Rosenberg, I. G. und F. Stenger: *A lower bound on the angles of triangles constructed bisecting the longest side*. Math. Comp, 29:390–395, 1975.

[Sch95] Schwefel, Hans-Paul: *Evolution and Optimum Seeking*. Wiley & Sons, 1995.

[Str95] Stroustrup, Bjarne: *Die C++ Programmiersprache*. Addison-Wesley, 1995.

[Wik] Wikipedia, `http://de.wikipedia.org/`: *Wikipedia*.

Lebenslauf

Geboren am 23. März 1971 in Stuttgart als Sohn von Hermann App und Ingrid App, geb. Seid.

August 1977	–	Juli 1981	Fuchsrain-Grundschule in Stuttgart
August 1981	–	Juni 1990	Wagenburg-Gymnasium in Stuttgart
Juli 1990	–	Juni 1991	Wehrdienst
Oktober 1991	–	September 1998	Studium der Mathematik mit Nebenfach Informatik an der Universität Stuttgart
seit Oktober 1998			berufliche Tätigkeiten • im Projekt CALMIR2 • im Projekt Mathematik-Online • Computerbetreuung an der Universität Stuttgart

Abstract

This work was done in the scope of a collaboration project between the *Mathematisches Institut A* (meanwhile *Institut für Mathematische Methoden in den Ingenieurwissenschaften, Numerik und geometrische Modellierung*) at the University of Stuttgart and the Philips company , more precisely the department *Philips Lighting* in Miribel near Lyon. The official title of this project was:

Study and realization of a software for fast calculation of 3 dimensional reflector design based on beam tracing and optimization algorithms.

The aim of this project was the development of a program, capable of fully automatic optimization of a given optical system with respect to a given target light distribution. The fields of application of these optical systems are mainly in street lighting. The initial reflector configurations together with the target light distributions are provided by optical engineers at Philips.

The internal name of this project was CALMIR2, which is also the name of the resulting program. This abbreviation stands for

Cal*culation of* Mir*rors in* Mir*ibel*

and shows that it is the second project of this kind within this department. However, the first project ([Phi96], [Phi97]) was more rudimentary and only contained a raytracer and some conversion formulas.

The beamtracer ([Gre95]), that deals with rays as cones and that was originally intended for this project, was soon replaced by a more common raytracer, that treats rays of light as straight lines. For the light source the model of a lambertian cylinder is used. With this, the radiant intensity is only contingent on the cosine of the angle ϑ between the surface normal and the emitting ray

$$I(\vartheta) = L\, dA\, \cos(\vartheta)\,,$$

where L is the luminance and dA the area of an elementary patch of the lamp. Figure A.1, in which the circle describes the radiant intensity in the appropriate direction, explains this.

There are two ways to simulate this behaviour within a raytracer. The first one is the so called angular density method and the second is the random generation of starting rays. With the first method, the lamp cylinder with length l and diameter d is divided into n_l slices, each with the thickness $\Delta l = l/n_l$, and n_d sectors, each with an angle of $\pi d/n_d$. For each patch, there are then $n_\square$ rays generated, each with a flux of

$$\Phi_{\text{ray}} = \Phi/(n_l n_d n_\square)\,,$$

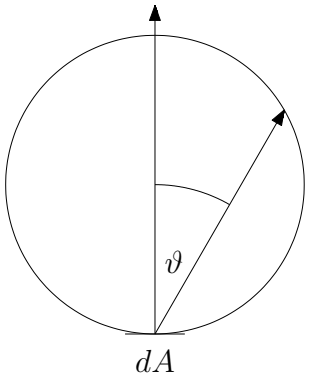


Figure A.1: Lambertian law

where Φ is the total flux of the lamp. In section 2.1.1 it is shown that the lambertian law is approximated, if for each patch the azimuth angles are divided into $n_\vartheta = \left\lceil \sqrt{n_\square \pi / 8} \right\rceil$ and the zenith angles into $n_\varphi = [8 n_\vartheta \cos(\vartheta) \sin(\vartheta)]$ equally sized sectors and for each of those one ray is generated.

For the random generation of starting rays a starting point is chosen randomly along the length $l_{\text{rnd}} = l\,\text{rnd}_1$ and along the perimeter $u_{\text{rnd}} = \pi d\,\text{rnd}_2$ of the lamp. Then an azimuth angle $\varphi_{\text{rnd}} = 2\pi\,\text{rnd}_3$ and a zenith angle $\vartheta_{\text{rnd}} = \arccos(\sqrt[n+1]{\text{rnd}_4})$ are chosen randomly and a ray is generated with this direction and the flux $\Phi_{\text{ray}} = \Phi / n_{\text{rays}}$.

The ray is then followed until it is reflected or leaves the optical system. When the ray is reflected the concept of narrow diffusion is applied. This means, the reflected ray is chosen randomly within a cone around the specular reflection path using $\vartheta_{\text{rnd}} = 2\vartheta_{\text{diffusion}} \arccos\left(\sqrt[b+1]{\text{rnd}}\right)/\pi$, which is similar to the angular density method of creating rays above.

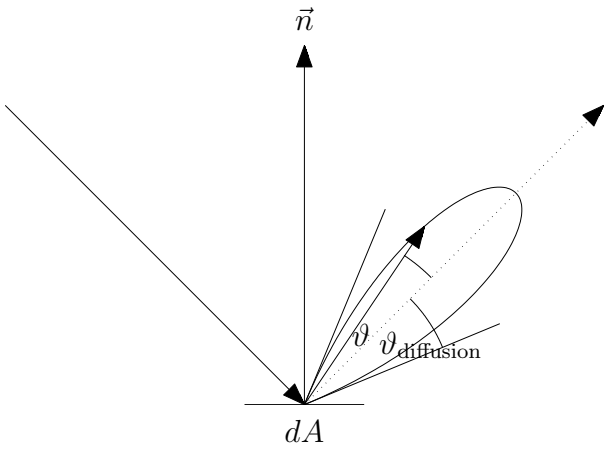


Figure A.2: Narrow diffusion

The reflector surfaces used in this work are NURBS surfaces. However, during the project it was soon discovered that computing reflections directly with these kinds of surfaces is rather expensive and slow. So the NURBS surface is first approximated by a triangulation, since it is much faster to calculate intersections with the light ray and normals for triangles. To gain better results with this approximation the normal of the NURBS surface is interpolated.

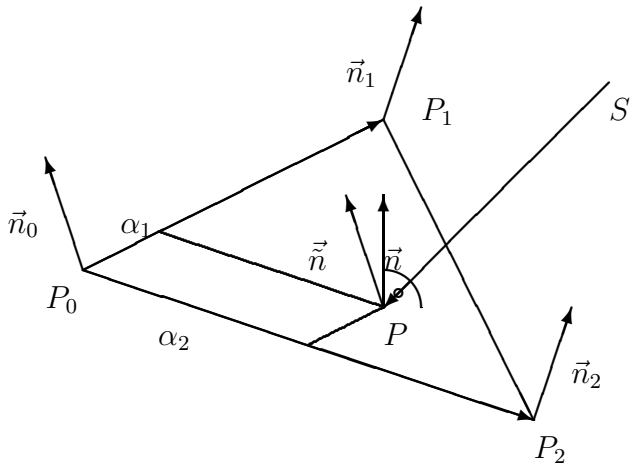


Figure A.3: Interpolation of the surface normal

To achieve this, for each corner of a triangle the corresponding normal of the NURBS surface is stored. If now a light ray hits the triangle in the intersection point P, this can be calculated as

$$P = P_0 + \alpha_1(P_1 - P_0) + \alpha_2(P_2 - P_0)\,.$$

For the interpolated normal $\vec{\tilde{n}}$, the formula

$$\vec{\tilde{n}} = (1 - \alpha_1 - \alpha_2)\vec{n}_0 + \alpha_1\vec{n}_1 + \alpha_2\vec{n}_2$$

is used. This method can even be combined with the narrow diffusion.

The tracing of the ray through the optical system is done with the so-called *brick layer algorithm*, described in section 3.9. For this algorithm a bounding box parallel to the coordinate axes around the optical system is divided in each direction into smaller ones (*bricks*). For each brick a list of triangles that intersect this brick is generated. If now a ray passes through a specific brick, only this list has to be checked for intersections instead of all triangles. Furthermore, local offsets in each coordinate direction (3.106, 3.107, 3.108) are calculated. With these, only five comparisons and two addition operations have to be done to pass through an empty brick.

When the ray finally leaves the optical system, its flux is described in a spherical coordinate system. The merit function of the optimizer measures the differences between the

achieved flux $\Phi_{i,j}$ and the target flux $\tilde{\Phi}_{i,j}$ for a given angular direction by

$$M = w \sqrt{\frac{\sum_{i,j} w_{i,j} \left(\frac{\Phi_{i,j}}{\Phi} - \frac{\tilde{\Phi}_{i,j}}{\tilde{\Phi}} \right)^2}{\sum_{i,j} w_{i,j} \left(\frac{\tilde{\Phi}_{i,j}}{\tilde{\Phi}} \right)^2}} + (1-w) \left(1 - \frac{\Phi}{\tilde{\Phi}} \right)$$

where Φ denotes the total achieved flux and $\tilde{\Phi}$ the total target flux. The scalars $w_{i,j}$ are weights for the individual angular direction, and w is a parameter given by the user to adjust wether the differences in the angular directions (the first term) or the difference in the achieved flux (the second term) have to be weighted more.

One of the main parts of this work is the possibility to compute the triangulation of a given NURBS surface very fast with a new technique called orthogonal B-splines (see section 3.5). The most important idea is to use a modified Sobolev scalar product $\langle \cdot, \cdot \rangle^o$, with respect to which linear B-splines over a plane triangulation become orthogonal. The main advantage of this method is that approximations over this orthognal B-spline basis can now be done very fast by the projection lemma. This new scalar product does not only account for the difference in the function values, but also for differences in the first derivatives or – equivalently for the local surface normal, which is an important aspect when dealing with reflections. It turns out, that the resulting approximation technique is of order $O(l^2)$, where l is the maximum length of an edge within a given triangle.

Since there is still a need to compute certain integrals to use this method, a discrete approximation (see section 3.7) of this new scalar product is used here. For this, the function that is to be approximated is evaluated at the corners and the midpoints of all edges of each triangle.

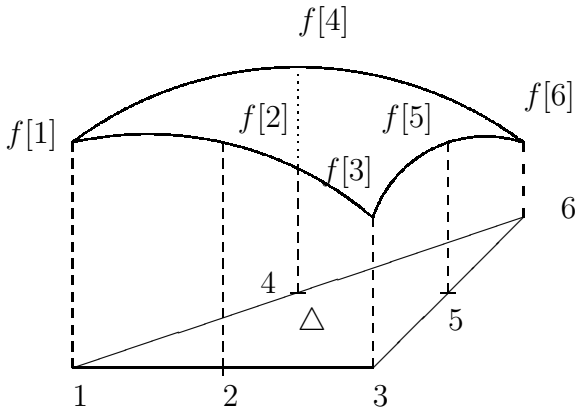


Figure A.4: Evaluation of f at six points over a triangle

The aim of the discrete approximation is to find a 6-by-6-matrix V, which approximates the scalar product found above by those six function values

$$\langle f, g \rangle^o_{\triangle} \approx f[1, \ldots, 6]^{\mathrm{t}} V g[1, \ldots, 6] \,.$$

If f and g are polynomials of degree two, it is even possible to achieve equality. So with $1-x-y, x, y, x^2, y^2, (1-x-y)^2$ replacing f and g sequentially, there is the unique solution

$$V = \begin{pmatrix} \frac{3}{5} & -\frac{1}{3} & \frac{1}{15} & -\frac{1}{3} & -\frac{1}{15} & \frac{1}{15} \\ -\frac{1}{3} & \frac{38}{15} & -\frac{1}{3} & -\frac{2}{5} & -\frac{2}{5} & -\frac{1}{15} \\ \frac{1}{15} & -\frac{1}{3} & \frac{3}{5} & -\frac{1}{15} & -\frac{1}{3} & \frac{1}{15} \\ -\frac{1}{3} & -\frac{2}{5} & -\frac{1}{15} & \frac{38}{15} & -\frac{2}{5} & -\frac{1}{3} \\ -\frac{1}{15} & -\frac{2}{5} & -\frac{1}{3} & -\frac{2}{5} & \frac{38}{15} & -\frac{1}{3} \\ \frac{1}{15} & -\frac{1}{15} & \frac{1}{15} & -\frac{1}{3} & -\frac{1}{3} & \frac{3}{5} \end{pmatrix}.$$

It turns out, that this discrete approximation is also of order $O(l^2)$, where l stands for the maximum length of an edge within a given triangle. In addition, this method only requires two function evaluations for each triangle, compared to 3/2 function evaluations for the standard interpolation method with an extra function evaluation at the midpoint of the triangle to measure the error, which is needed for adaptive triangulations.

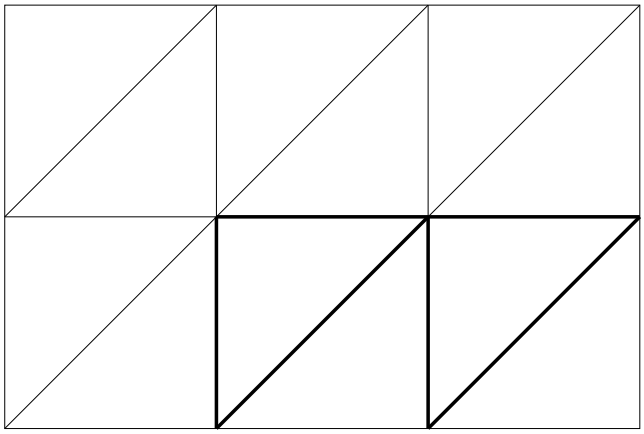


Figure A.5: Initial triangulation

These adaptive triangulations are done with the red-green-refinement from Bank, Sherman and Weiser [BSW83]. This algorithm uses three steps:

- all triangles with inadequate approximation are regularly subdivided into four similar triangles ("red refinement")
- as long as there are triangles with two or more hanging nodes, they are also regularly subdivided
- all triangles with one hanging node are subdivided into two triangles at this vertex ("green refinement")

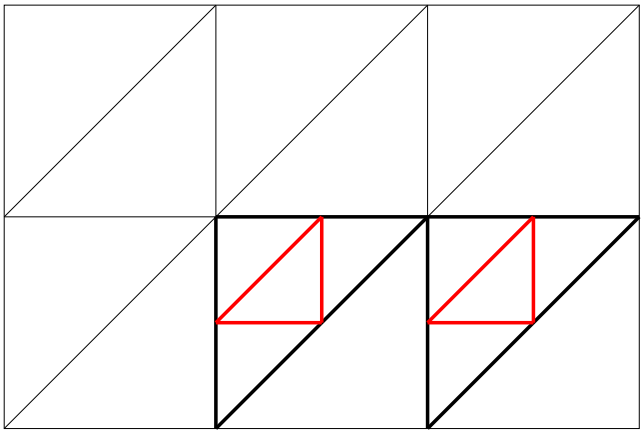


Figure A.6: Red refinement

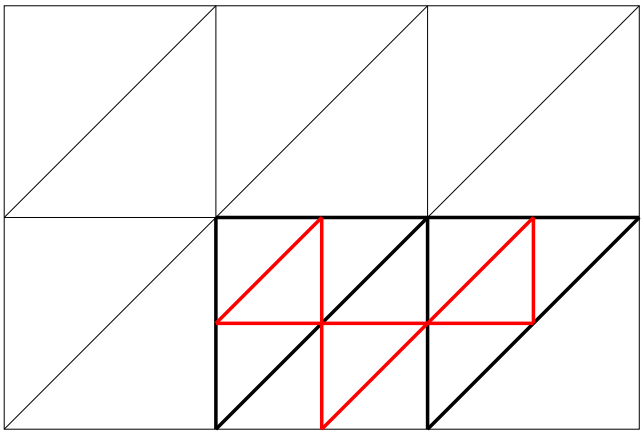


Figure A.7: Triangles with two or more hanging nodes are subdivided

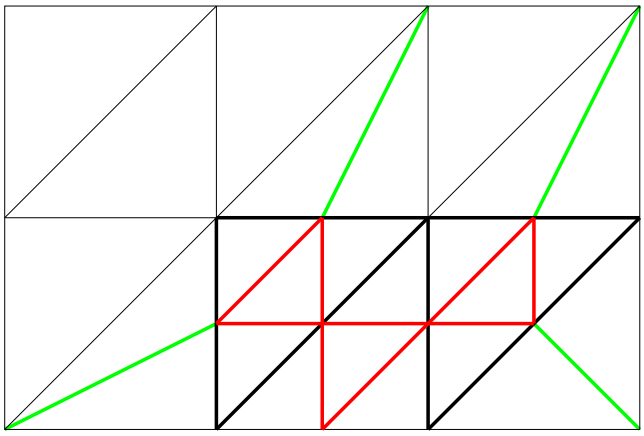


Figure A.8: Green refinement

The second main part of this work is the optimizing section. Since the original intended direct manipulation of the control points of the NURBS surfaces sometimes produced strange “bumps” and artifacts, this method turned out to be inappropriate. Instead the optimizer now uses a three dimensional Bézier space deformation to alter the given reflector surfaces, as described in section 4.1. This deformation allows a smooth modification of the surfaces. Nevertheless, it is possible to do only locally restricted changes with this technique, too. This method is also used in computer graphics as so-called “morphing” and is shown in the following two figures for the two dimensional case.

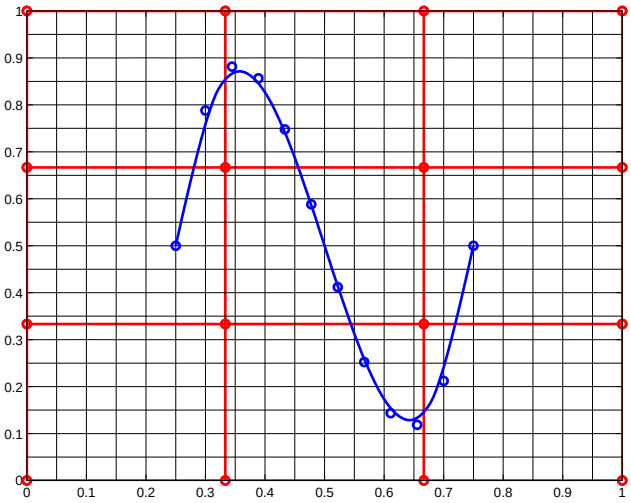


Figure A.9: Initial transformation

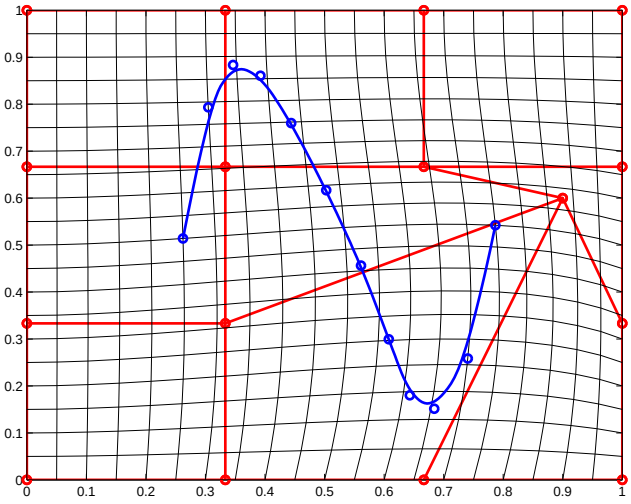


Figure A.10: Bézier space deformation

The optimizing algorithm used in this project is the so-called Hooke-Jeeves algorithm ([Sch95],[HH71]), which is described in detail in section 4.3. Actually every optimizing algorithm that doesn't use derivatives is suitable, but the Hooke-Jeeves algorithm performed quite well in tests.

A complete survey of these tests is given in [App00b], whereas in this work only selected tests are presented in chapter 6. The results of these tests were more or less as expected and show that CALMIR2 achieves its desired goals.

In summary (see chapter 7), these tests show that CALMIR2 is a tool, capable of fully automatic optimization of a given reflector configuration with respect to a given target light distribution in reasonable time. So this program will be a big help for the optical engineers to find better reflector configurations, since it speeds up the optimizing progress a lot, which is currently still done by hand.