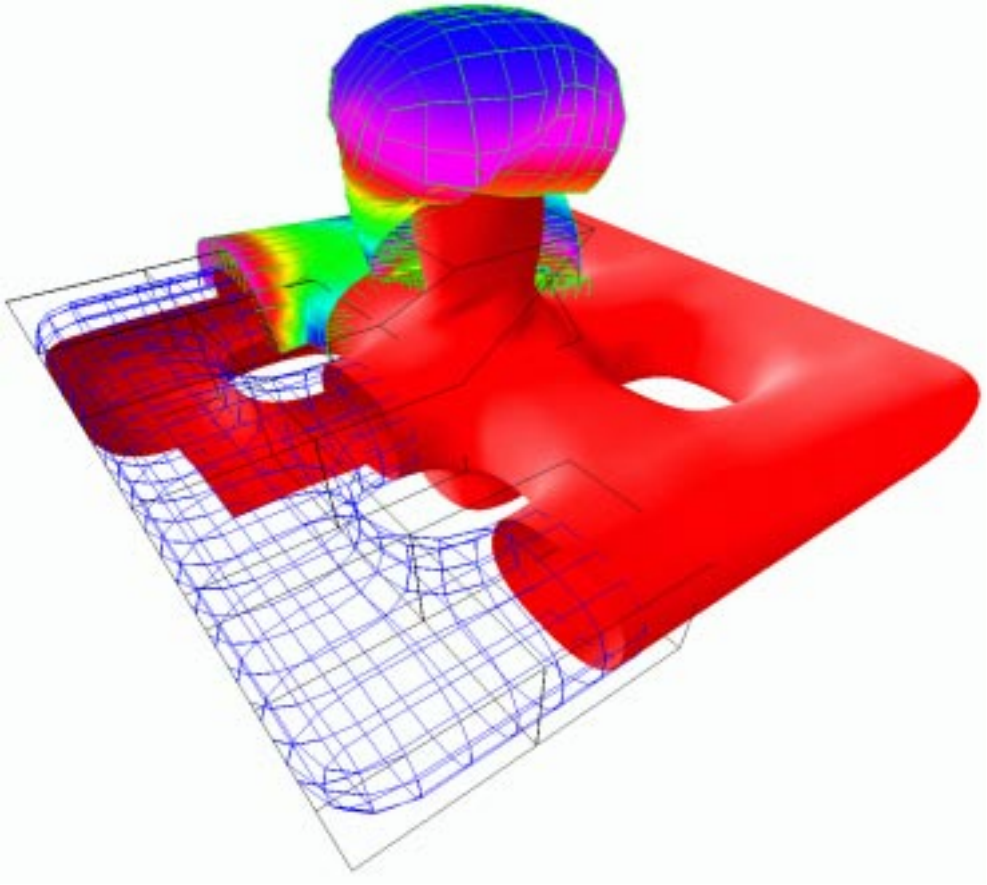
Visualisierung von Funktionalen auf Freiformflächen

Diplomarbeit von Frank Langbein

Mai 1999



bei Prof. Dr. Klaus Höllig und Dr. Ulrich Reif

Mathematisches Institut A, 2. Lehrstuhl
Universität Stuttgart

As time wore along, his absorption in the irregular wall and ceiling of his room increased; for he began to read into the odd angles a mathematical significance which seemed to offer vague clues regarding their purpose.

— H.P. Lovecraft
"The Dreams in the Witch–House"

Danksagung

Diese Arbeit wurde am zweiten Lehrstuhl des Mathematischen Instituts A der Universität Stuttgart unter Leitung von Herrn Prof. Dr. Klaus Höllig angefertigt.

Mein Dank gilt Herrn Prof. Dr. Klaus Höllig für die interessante Aufgabenstellung und seine Unterstützung bei der Ausführung. Zudem danke ich Herrn Dr. Ulrich Reif für seine gute Betreuung und für die vielen wertvollen Anregungen. Ich möchte mich auch bei Frau Sonja Schirmer für ihre konstruktive Kritik und ihre Ermutigungen, die mich immer wieder neu motiviert haben, bedanken.

Ich möchte dankend alle Entwickler freier Software und vor allem die GNU/Linux–Entwickler erwähnen. Ohne die exzellente Arbeit all dieser Enthusiasten hätte weder dieses Dokument noch das Programm erstellt werden können.

Mein größter Dank gilt meinen Eltern, deren Unterstützung mein Mathematikstudium in dieser Form erst möglich gemacht hat.

Frank Langbein[1]
Stuttgart, Mai 1999

[1] E–Mail: <langbein@mathematik.uni-stuttgart.de>

Inhaltsverzeichnis

Abbildungsverzeichnis

Algorithmenverzeichnis

Einleitung

Wir beschäftigen uns in dieser Arbeit mit Funktionen und Flächen in der geometrischen Datenverarbeitung. Splines sind gut zur Modellierung von drei–dimensionalen Flächen geeignet. Uns interessieren nun vor allem auf diesen Flächen definierte Funktionen.

Unser Ziel ist die Entwicklung eines Modells zur Darstellung und Integration von Funktionen auf Flächen. Dabei sollen die Funktionen wie die Flächen durch Splines beschrieben werden. Im besonderen sollen dabei biquadratische Splines verwendet werden. Zusätzlich sollen auch getrimmte Flächen in diesem Zusammenhang behandelt werden.

Für diese Arbeit wurde das Programm LiLit zur Darstellung und Integration von Funktionen auf Flächen in C++ entwickelt. LiLit wird frei über die GNU General Public License vertrieben. Das Programm liest die Objektbeschreibungen über eine ODL–Datei ein und kann Punkte, Polygone, G–Spline–Flächen und Funktionen mit den vorgestellten Algorithmen bearbeiten und mit Hilfe von OpenGL darstellen. Die Ausgabe erfolgt entweder über ein Grafikdisplay oder als encapsulated PostScript–Datei.

In Kapitel 1 werden geometrisch stetige Splinekurven eingeführt und über die Tensorproduktmethode werden hieraus Splineflächen erzeugt. Über geometrisch glatte Flächen werden die biquadratischen G–Spline–Flächen vorgestellt und die grundlegenden Algorithmen für diese Flächen beschrieben. Für diese Arbeit werden nur biquadratische G–Spline–Flächen verwendet. Zum Erzeugen von Kontrollnetzen für biquadratische G–Splines wird noch der Doo–Sabin–Subdivision–Algorithmus vorgestellt.

In Kapitel 2 werden zunächst zwei geometrisch glatt zusammengesetzte, biquadratische Bézierflächenstücke betrachtet, auf denen jeweils eine Funktion gegeben ist. Es wird eine Bedingung gesucht, um diese beiden Funktionen stetig zusammenzusetzen. Hieraus wird die Darstellung dieser Funktionen durch biquadratische Splines entwickelt. Es werden die Darstellungen über Farben, als Stacheln, Gitter über der Fläche und farbige Flächen über der Fläche vorgestellt. Zudem wird der bivariate Romberg–Algorithmus für Oberflächenintegrale über diesen durch Splines dargestellten Funktionen beschrieben. Zusätzlich wird ein Algorithmus zur Darstellung einer Funktion über Isolinien betrachtet und die Krümmung der Fläche als spezielle Funktion eingeführt.

In Kapitel 3 werden getrimmte Flächen behandelt. Aus den Flächen können dabei sowohl über implizite Funktionen im Raum der Fläche als auch über Kurven im Parameterbereich Teile ausgeschnitten werden. Es wird ein Algorithmus zur Darstellung solcher Flächen beschrieben, der gleichzeitig zeigt, wie andere Algorithmen auf getrimmte Flächen angepaßt werden können. Die Auswirkungen auf die Darstellung und Integration von durch Splines dargestellten Funktionen auf getrimmten Flächen werden kurz angesprochen.

In Kapitel 4 wird das für diese Arbeit entwickelte Programm LiLit vorgestellt. Es wird sowohl kurz die Programmstruktur, als auch die Bedienung des Programms behandelt. Vor allem wird das von LiLit verwendete ODL–Dateiformat beschrieben. Zusätzlich werden noch zwei Dateiformate zur vereinfachten Erzeugung von Grafikobjekten in ODL besprochen. Mit dem SBW–Format können Flächenkontrollnetze aus gleichgroßen Würfeln zusammengesetzt werden. Mit dem FNC–Format können die Kontrollnetze über eine Parametrisierung beschrieben werden. In beiden Formaten können auch die Kontrollnetze für die Funktionen auf den Flächen über Parametrisierungen angegeben werden.

Anhang A enthält eine vollständige Programm–Dokumentation, in der alle in LiLit vorkommenden Datenstrukturen, Methoden und Variablen beschrieben werden.

Kapitel 1

Kurven und Flächen

Wir werden in diesem Kapitel ein Modell zur Darstellung von Flächen einführen, die durch möglichst allgemeine Kontrollnetze beschrieben werden. Dazu betrachten wir zunächst die Approximation von stetigen Funktionen durch Polynome. Um die guten lokalen Eigenschaften der Polynomapproximation ohne die globalen Nachteile auszunutzen, führen wir stückweise polynomiale Funktionen über die B–Spline–Basis ein. Mit Hilfe dieser Splines definieren wir Splinekurven und behandeln besonders die Bézierdarstellung und deren Eigenschaften. Nach einer kurzen Betrachtung der geometrischen Stetigkeit für Kurven, verwenden wir die Tensorproduktmethode, um aus den Kurven Flächen im $\mathbb{R}^3$ zu erzeugen. Auch hier untersuchen wir die geometrischen Glattheitsbedingungen. Für eine vollständige Darstellung dieser Themen sei auf [Höl94], [Far93] und [HL92] verwiesen. Für unsere Zwecke sind vor allem die biquadratischen G–Spline–Flächen nach [Rei95] interessant. Wir stellen die wichtigsten Algorithmen zur Berechnung der G–Splines vor, die wir im nächsten Kapitel für die Darstellung von Funktionen auf den Flächen erweitern werden. Da die G–Splines semi–reguläre Kontrollnetze benötigen, beschreiben wir am Schluß noch den Doo–Sabin–Subdivision–Algorithmus, um auch allgemeinere Kontrollnetze verwenden zu können.

1.1 Approximation mit Polynomen

Nach dem Approximationssatz für stetige Funktionen von Weierstraß lassen sich stetige Funktionen auf einem endlichen, abgeschlossenen Intervall beliebig genau durch Polynome approximieren. In der Anwendung könnte man also eine solche Funktion $f : \mathbb{R} \supset [a, b] \to \mathbb{R}^N$ $(N \in \mathbb{N})$ durch ein Polynom darstellen.

Eine Möglichkeit hierfür wäre die diskrete Approximation bzw. Interpolation. Dazu wählt man $d+1$ Stützstellen $a \le t_0 \le \cdots \le t_d \le b$ zu den Stützwerten $f(t_0), \ldots, f(t_d)$ und fordert, daß das Interpolationspolynom $p : [a, b] \to \mathbb{R}^N$ die Stützwerte an den Stützstellen annimmt, also $p(u_j) = f(u_j)$ für $j = 0 : d$. Dann ist p eindeutig durch die Lagrange'sche Interpolationsformel bestimmt,

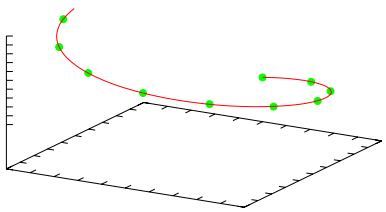


Abbildung 1.1: Beispiel zur polynomialen Interpolation einer Kurve im $\mathbb{R}^3$

$$p(t) = \sum_{j=0}^{d} f(t_j) L_j(t), \quad L_j(t) := \prod_{\substack{k=0 \\ k \neq j}}^{d} \frac{t - t_k}{t_j - t_k}. \tag{1.1}$$

Die L_j heißen Lagrange–Polynome. Abbildung 1.1 zeigt eine drei–dimensionale Kurve, die mittels dieser polynomialen Interpolation erzeugt wurde. Die Interpolationsdaten für diese Kurve wurden über $[\sin(t), \cos(t), t]^T$ erzeugt mit $t = 0 : 0.5 : \frac{3}{2}\pi$.

Allerdings ergeben sich bei dieser Methode auch einige Probleme. Jedes L_j ist von allen Stützstellen abhängig, d.h. p hängt global von den Stützstellen ab und muß bei jeder Änderung einzelner Approximationsdaten vollständig neu berechnet werden.

Ist der Funktionswert einer Polynomfunktion an einer Stelle von 0 verschieden und nicht konstant, dann strebt die Funktion für große x gegen unendlich. Eine Funktion mit kompaktem Träger kann so nur ungenügend durch ein Polynom approximiert werden.

Weiterhin erhöht sich der Grad von p um eins für jede weitere Stützstelle, womit auch die Berechnung des Polynoms selbst sehr komplex werden kann. Ein Polynom von hohem Grad approximiert den Gesamtverlauf der Funktion auch nicht unbedingt zufriedenstellend. Siehe hierzu auch Abbildung 1.2, in der die Funktion $\frac{1}{1+25x^2}$ an den Stellen $-1 : \frac{1}{3} : 1$ interpoliert wurde (vgl. [Höl98]). Die Orginalfunktion ist gepunktet eingezeichnet.

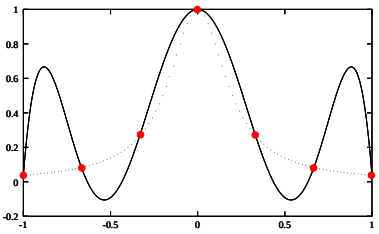


Abbildung 1.2: Polynomiale Interpolation der Funktion $\frac{1}{1+25x^2}$

1.2 Splineräume und B–Splines

Polynome sind dagegen sehr gut zur lokalen Approximation von glatten Funktionen geeignet. Um die Probleme der polynomialen Interpolation zu vermeiden, liegt es daher nahe, die Funktion stückweise durch Polynome zu approximieren. Dazu teilen wir ein Interval $U = [u_0, u_n)$ in n Teilintervalle $[u_{i-1}, u_i)$, $i = 1 : n$ auf, die durch die Knotenfolge $u = \{u_0, u_1, \ldots, u_n\}$ mit

$$u_0 < u_1 \leq u_2 \leq \cdots \leq u_{n-1} < u_n \tag{1.2}$$

bestimmt werden. Auf jedem dieser Teilintervalle stellen wir die approximierende Funktion p als Polynom vom maximalen Grad d dar. Dabei kann in der Knotenfolge jeder Knoten maximal $d+1$–mal vorkommen. Kommt ein Knoten u_i genau m–mal vor, d.h. $u_{i-1} < u_i = u_{i+1} = \cdots = u_{i+m-1} < u_{i+m}$, dann sind die ersten $d - m$ Ableitungen von p an diesem Knoten stetig. m heißt die Vielfachheit von u_i. p ist dann eine Splinefunktion zur Knotenfolge u vom Grad d mit dem Definitionsbereich U. Den hieraus entstehenden Splineraum bezeichnen wir mit $S_{u,d}(U)$. Um eine Basis für diesen Raum zu erhalten, erweitern wir zunächst die Knotenfolge u in beide Richtungen um jeweils d Punkte:

$$u_{-d} \leq \cdots \leq u_{-1} \leq u_0 < u_1 \leq u_2 \leq \cdots \leq u_{n-1} < u_n \leq u_{n+1} \leq \cdots \leq u_{n+d}. \tag{1.3}$$

Wir können auch annehmen, daß u in beide Richtungen unendlich fortgesetzt wird und wir nur die jeweils relevanten Knoten betrachten. Die Knoten außerhalb des Intervalls U haben keinen Einfluß auf die im Splineraum enthaltenen Funktionen. Eine mögliche Basis für diesen Raum bilden die abgebrochenen Potenzen. Jedes $p \in S_{u,d}(U)$ läßt sich darstellen als

$$p(x) = \sum_{j=-d}^{n-1} p_j (x - u_j)_+^{d-\#j} \tag{1.4}$$

mit den Koeffizienten $p_j \in \mathbb{R}$ und

$$\begin{aligned} (x - u_j)_+ &:= max\{x - u_j, 0\}, \\ \#j &:= max\{k : u_{j+k} = u_j\}. \end{aligned} \tag{1.5}$$

Dabei ergeben die ersten $d + 1$ abgebrochenen Potenzen eine Basis für die Polynome vom maximalen Grad d auf dem Intervall $[u_0, u_1)$. Die restlichen Terme bestimmen die "Sprünge" in den Ableitungen an den Knoten. Aus dieser Darstellung folgt, daß die Dimension von $S_{u,d}(U)$ gleich $n + d$ ist.

Die Basis der abgebrochenen Potenzen (1.4) für $S_{u,d}(U)$ besitzt allerdings keinen kompakten Support, d.h. Änderungen der Koeffizienten p_j haben globale Auswirkungen. Dieser Nachteil wird durch die B–Splines aufgehoben.

Definition 1.1 (B–Splines). *Zu einer vorgegebenen Knotenfolge $u = \{u_j\}_{j\in\mathbb{Z}}$ für die $\lim_{j\to\pm\infty} u_j = \pm\infty$ gilt, sei der B–Spline $B_{j,d,u}$ vom Grad d wie folgt rekursiv definiert.*
Für $d = 0$ sei

$$B_{j,0,u}(x) := \begin{cases} 1 & \text{falls } u_j \le x < u_{j+1}, \\ 0 & \text{sonst.} \end{cases} \tag{1.6}$$

Für $d > 0$ sei

$$B_{j,d,u}(x) := \omega_{j,d}(x) B_{j,d-1,u}(x) + (1 - \omega_{j+1,d}(x))\, B_{j+1,d-1,u}(x) \tag{1.7}$$

mit

$$\omega_{j,d}(x) := \frac{x - u_j}{u_{j+d} - u_j}. \tag{1.8}$$

Die rekursive Konstruktion eines B–Splines vom Grad 2 für eine äquidistante Knotenfolge ist in Abbildung 1.3 veranschaulicht. Mit Hilfe der Marsden Identität erhält man folgendes Resultat.

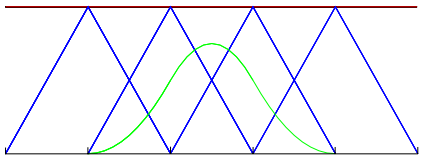


Abbildung 1.3: Rekursive Definition eines B–Splines vom Grad 2

Theorem 1.2 (B–Spline Basis). *Für das Intervall $U = [u_0, u_n)$ bilden die B–Splines $B_{j,d,u}$ für $-d \le j < n$ eine Basis für $S_{u,d}(U)$ mit der Knotenfolge $u_{-d} \le \cdots \le u_{n+d}$ und es gilt $\operatorname{supp} B_{j,d,u} = [u_j, u_{j+d+1}]$. Falls $u_j < u_{j+1}$, bilden die B–Splines $B_{j-d,d,u}, \ldots, B_{j,d,u}$ eine Basis der Polynome vom maximalen Grad d auf $[u_j, u_{j+1})$.*

Beweis. Den Beweis hierzu findet man in [Höl94] und [Höl98]. ∎

Eine Splinekurve c vom Grad d im $\mathbb{R}^N$ mit den Kontrollpunkten $p_j = [p_{j,1}, \ldots, p_{j,D}]^T \in \mathbb{R}^N$ hat demnach die Form

$$c(t) := \sum_{j=-d}^{n-1} p_j B_{j,d,u}(t), t \in U. \tag{1.9}$$

Damit sind die Koordinatenfunktionen von c Splinefunktionen aus $S_{u,d}(U)$.

1.3 Bézierkurven

Wir betrachten nun eine Knotenfolge, in der jeder Knoten die maximale Vielfachheit $d+1$ besitzt. Es genügt ein einzelnes Intervall mit $d+1$–fachen Randknoten zu untersuchen, z.B.

$$u_{-d} = \cdots = u_0 < u_1 = \cdots = u_{1+d}. \tag{1.10}$$

Dann lautet die Rekursionsformel für die für dieses Intervall relevanten B–Splines

$$B_j^d(x) = \frac{x - u_0}{u_1 - u_0} B_j^{d-1}(x) + \frac{u_1 - x}{u_1 - u_0} B_{j+1}^{d-1}(x) \tag{1.11}$$

mit $B_j^d := B_{j-d,d}$. Über vollständige Induktion kann man dann beweisen, daß

$$B_j^d(x) = \binom{d}{j} \left(\frac{x - u_0}{u_1 - u_0}\right)^{d-j} \left(\frac{u_1 - x}{u_1 - u_0}\right)^j \tag{1.12}$$

gilt. Für das Intervall $[u_0, u_1] = [0, 1]$ erhält man so die Bernstein–Polynome.

Definition 1.3 (Bernstein–Polynome). *Die Bernstein–Polynome B_j^d für $j = 0 : d$ vom Grad $d > 0$ seien definiert durch*

$$B_j^d(t) := \binom{d}{j} t^j (1-t)^{d-j}, \tag{1.13}$$

wobei die Binomialkoeffizienten durch

$$\binom{d}{j} := \begin{cases} \frac{d!}{j!(d-j)!} & \text{für } 0 \le j \le d, \\ 0 & \text{sonst} \end{cases} \tag{1.14}$$

gegeben sind.

In Abbildung 1.4 sind die quadratischen Bernstein–Polynome dargestellt.

Es gilt

$$\begin{aligned} B_j^d(t) &= \left(\binom{d-1}{j} + \binom{d-1}{j-1} \right) t^j (1-t)^{d-j} \\ &= (1-t) B_j^{d-1}(t) + t B_{i-1}^{d-1}(t) \end{aligned} \tag{1.15}$$

mit $B_0^0(t) \equiv 1$ und $B_j^d(t) \equiv 0$ für $j \notin \{0, 1, \ldots, d\}$. Hieraus erhält man eine einfache Methode, um Bernstein–Polynome von Grad d aus denen von Grad $d-1$ zu berechnen. Weiterhin folgt aus dem binomischen Satz, daß

$$\begin{aligned} \sum_{j=0}^{d} B_j^d(t) &= \sum_{j=0}^{d} \binom{d}{j} t^j (1-t)^{d-j} \\ &= (t + (1-t))^d = 1. \end{aligned} \tag{1.16}$$

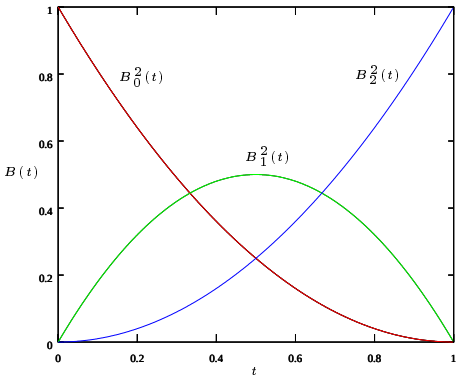


Abbildung 1.4: Quadratische Bernstein–Polynome

Die Ableitung eines Bernstein–Polynoms berechnet sich nach der Rekursionsformel

$$\frac{d}{dt} B_j^d(t) = d \left(B_{j-1}^{d-1}(t) - B_j^{d-1}(t) \right). \tag{1.17}$$

Mit Hilfe der Bernstein–Polynome lassen sich nun die Bézierkurven definieren.

Definition 1.4 (Bézierkurven). *Eine Bézierkurve vom Grad $d > 0$ mit den Kontrollpunkten (Bézierpunkten) $p_j \in \mathbb{R}^N$, $j = 0 : d$ sei definiert durch*

$$c(t) := \sum_{j=0}^{d} p_j B_j^d(t). \tag{1.18}$$

Jede Splinekurve (1.9) läßt sich durch Bézierkurven darstellen, indem man in die gegebene Knotenfolge u jeden Knoten solange einfügt bis er die Vielfachheit d hat (siehe [Höl94]). Die Splinekurve wird dann aus Bézierkurven vom Grad d mit jeweils $d+1$ Kontrollpunkten zusammengesetzt. Bézierkurven sind damit spezielle Splinekurven mit sehr nützlichen Eigenschaften. Man erkennt dies schon an Abbildung 1.5, welche eine Bézierkurve vom Grad 3 zeigt.

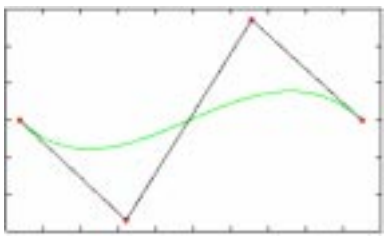


Abbildung 1.5: Bézierkurve

Wegen (1.16) und $B_j^d(t) \ge 0$ folgt direkt aus der Definition, daß Bézierkurven immer in der konvexen Hülle ihrer Kontrollpunkte

liegen. Aus den beiden Eigenschaften folgt auch, daß Bézierkurven affin invariant sind, da baryzentrische Kombinationen invariant unter affinen Abbildungen sind, d.h. wenn man eine affine Abbildung auf die Kurve anwendet, erzeugt dies die gleiche Kurve, wie wenn man die affine Abbildung auf die Kontrollpunkte anwendet. Eine Bézierkurve ist auch invariant unter einer affinen Parametertransformation $T : [0,1] \to [a,b], t \mapsto t'$, d.h. es gilt

$$\sum_{j=0}^{d} p_j B_j^d(t) = \sum_{j=0}^{d} p_j B_j^d\left(\frac{t'-a}{b-a}\right). \tag{1.19}$$

Weiter erhält man sofort aus der Definition der Bernstein–Polynome, daß

$$B_j^d(t) = B_{d-j}^d(1-t). \tag{1.20}$$

Damit gilt

$$\sum_{j=0}^{d} p_j B_j^d(t) = \sum_{j=0}^{d} p_{d-j} B_j^d(1-t). \tag{1.21}$$

Dies bedeutet, daß die Kontrollpunkte sowohl mit $p_0, \ldots, p_d$, als auch mit $p_d, \ldots, p_0$ bezeichnet werden können.

Es gilt auch $c(0) = p_0$ und $c(1) = p_d$, da $B_j^d(0) = \delta_{j,0}$ und $B_j^d(1) = \delta_{j,d}$ wobei $\delta_{j,d}$ das Kroneckersymbol bezeichnet. Damit interpolieren die Bézierkurven immer den Anfangs- und Endpunkt des Kontrollpolygons. Weiter gilt auch, daß die Richtung der Tangente der Kurve in $c(0)$ und $c(1)$ der Richtung der ersten Strecke des Kontrollpolygons von p_0 nach p_1 bzw. der letzten Strecke von p_{d-1} nach p_d entspricht.

Man kann weiter zeigen, daß gilt

$$\sum_{j=0}^{d} \frac{j}{d} B_j^d(t) = t. \tag{1.22}$$

Wenn man also die Kontrollpunkte gleichmäßig auf einer Geraden, die die Punkte a und b verbindet, verteilt, d.h. $p_j = \left(1 - \frac{j}{d}\right) a + \frac{j}{d} b$, $j = 0 : d$, dann erzeugt c genau die Strecke von a nach b. Diese Eigenschaft heißt lineare Präzision.

Das Bernstein–Polynom B_j^d besitzt nur ein Maximum an der Stelle $\frac{j}{d}$, d.h. wenn wir nur einen Kontrollpunkt p_j ändern, ist die Änderung der Kurve im Bereich des Parameterwertes $\frac{j}{d}$ am stärksten. Dadurch werden die Auswirkungen von Änderungen vorhersehbar und vor allem ändert sich der Verlauf der Kurve nur in einem lokal beschränkten Bereich.

1.4 Geometrische Stetigkeit

Durch die Knotenfolge (1.3) wird die Glattheit der aus Polynomen zusammengesetzten Kurve festgelegt. Kommt ein Knoten genau m–mal vor, so sind die ersten $d - m$ Ableitungen der Kurve an dieser Stelle stetig. Anstatt die Glattheit der Parametrisierung zu fordern, können wir auch verlangen, daß die Kurve geometrisch glatt ist.

Definition 1.5 (Geometrische Stetigkeit). *Sei $c : [u_0, u_n] \to \mathbb{R}^N$ eine reguläre Parametrisierung einer Splinekurve. Die Splinekurve ist eine m–mal stetig differenzierbare Kurve, wenn für jeden Knoten u_j aus der Knotenfolge* (1.3) *eine stetige Funktion ϕ mit $\phi(u_j) = u_j$ und $\phi' > 0$ existiert, so daß für $d(s) := c(\phi(s))$ gilt*

$$c_+^{(k)}(u_j) = d_-^{(k)}(u_j), \quad k = 0 : m. \tag{1.23}$$

Für eine stetig differenzierbare Kurve erhalten wir damit die Bedingungen

$$c_+(u_*) = c_-(u_*), \tag{1.24}$$
$$c'_+(u_*) = c'_-(u_*)\phi'(u) \tag{1.25}$$

an einem Knoten u_*. Dabei bedeutet die erste Gleichung nur, daß die Kurve am Knoten u_* zusammenhängt. Die zweite Gleichung bedeutet, daß die Tangentenvektoren c_+ und c_- in u_* parallel, aber nicht notwendig gleich sind.

Wir untersuchen dies nun speziell für Bézierkurven. Dazu können wir die einzelnen als Bézierkurve darstellbaren Kurvensegmente zunächst unabhängig voneinander betrachten. Jedes dieser Segmente wird durch $d+1$ Kontrollpunkte bestimmt. Soll die gesamte Kurve einmal stetig differenzierbar sein, dann müssen obige Gleichungen jeweils an den Stellen gelten, an denen die Kurvensegmente zusammengesetzt werden. Seien $p_0, \ldots, p_d$ und $q_0, \ldots, q_d$ die Kontrollpunkte zweier benachbarter Kurvensegmente (vgl. Abbildung 1.6). Wir nehmen an, daß (1.24) an p_d erfüllt ist. Also gilt $p_d = q_0$, da Bézierkurven die Anfangs- und Endpunkte des Kontrollpolygons interpolieren. Nach (1.25) müssen die Tangenten an $p_d = q_0$ parallel sein, d.h. Die Punkte p_{d-1}, $p_d = q_0$, und q_1 müssen kolinear sein, denn die Tangenten an den Anfangs- und Endpunkten der Bézierkurven werden ebenfalls durch das Kontrollpolygon bestimmt.

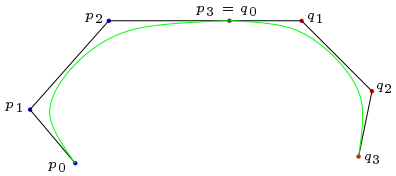


Abbildung 1.6: Geometrisch stetige Bézierkurve

Falls eine Kurve eine glatte, reguläre Parametrisierung $c(t)$ hat, dann ist auch die Parametrisierung nach der Bogenlänge

$$s(t) = \int_{u_0}^{t} |c'(x)|dx \tag{1.26}$$

glatt. Hieraus lassen sich dann wie in [Höl94] weitere Bedingungen an die Kontrollpunkte der Bézierkurven für Krümmungsstetigkeit, Stetigkeit der Binormalen, etc. ableiten. Im folgenden werden wir dies aber nicht benötigen.

1.5 Tensorproduktflächen

Nun sollen mit Hilfe der Splinefunktionen, insbesondere der Bézierkurven, Flächen im $\mathbb{R}^3$ dargestellt werden. Intuitiv kann man eine Fläche auch als Kurve auffassen, die sich durch den Raum bewegt und dabei ihre Form verändert (vgl. [Far93]). Dies ist schematisch in Abbildung 1.7 dargestellt. Wir beschränken uns im folgenden darauf, daß diese Kurve zu jedem Zeitpunkt eine Bézierkurve vom Grad α ist. Sei c^v die Kurve zum Zeitpunkt v, wobei c^0 die Anfangskurve und c^1 die Endkurve sei,

$$c^v : [0,1] \to \mathbb{R}^3, \quad u \mapsto \sum_{j=0}^{\alpha} P_j(v) B_j^{\alpha}(u). \tag{1.27}$$

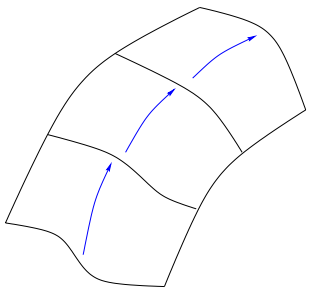


Abbildung 1.7: Tensorproduktfläche

Bewegt sich diese Kurve im Raum, so wird ihre Form zu jedem Zeitpunkt v durch eine Menge von Kontrollpunkten $\{P_j(v)\}_{j=0:\alpha}$ bestimmt. Damit geht von jedem einzelnen ursprünglichen Kontrollpunkt $P_j(0)$ eine Kurve aus, die die Lage der folgenden Kontrollpunkte beschreibt. Wir beschränken uns nun weiter darauf, daß auch diese Kurven Bézierkurven vom festen Grad β sind, die selbst durch Kontrollpunkte beschrieben werden, d.h. $P_j(v)$ wird beschrieben durch

$$P_j : [0,1] \to \mathbb{R}^3, \quad v \mapsto \sum_{k=0}^{\beta} P_{j,k} B_k^{\beta}(v). \tag{1.28}$$

Kombiniert man beide Gleichungen, erhält man eine Fläche S als Tensorprodukt

$$S : [0,1]^2 \to \mathbb{R}^3, \quad (u,v) \mapsto \sum_{j=0}^{\alpha} \sum_{k=0}^{\beta} P_{j,k} B_j^{\alpha}(u) B_k^{\beta}(v), \tag{1.29}$$

die durch die Punkte $P_{j,k}$ vollständig beschrieben wird. Viele der Eigenschaften dieser Bézierflächen lassen sich direkt aus den Eigenschaften der Bézierkurven ableiten (siehe auch [Far93]).

Die Originalkurve c^0 hat die Kontrollpunkte $\{P_{i,0}\}_{i=0:\beta}$ und es ist einfach zu zeigen, daß jede beliebige Randkurve von S als Anfangskurve zu verwenden ist. Die Randkurven der Fläche sind Polynomkurven und werden durch die Randpunkte des Kontrollnetzes $P_{j,k}$ festgelegt. Insbesondere liegen alle vier Ecken des Kontrollnetzes auf der Fläche.

Es gilt

$$\sum_{j=0}^{\alpha} \sum_{k=0}^{\beta} B_j^{\alpha}(u) B_k^{\beta}(v) \equiv 1, \tag{1.30}$$

d.h. (1.29) ist eine affine Kombination und somit affin invariant.

Für $0 \leq u, v \leq 1$ sind $B_j^{\alpha}(u) B_k^{\beta}(v)$ nicht negativ. Zusammen mit (1.30) folgt damit, daß (1.29) eine Konvexkombination ist. Die Fläche liegt somit in der konvexen Hülle der Kontrollpunkte.

1.6 Geometrisch glatte Flächen

Die Tensorproduktflächen der Form (1.29) lassen sich nun ähnlich wie die Splinekurven geometrisch glatt zusammensetzen. Wir betrachten die gesamte Fläche als aus Flächenstücken der Form (1.29) erzeugt, wobei die Kontrollpunkte $P_{j,k}$ bestimmte Glattheitsbedingungen erfüllen müssen. Dazu folgen wir hier dem Ansatz aus [Rei95].

Seien die folgenden beiden Flächenstücke p und q gegeben:

$$p : \quad [0,1]^2 \mapsto \mathbb{R}^3, \quad p(u,v) = \sum_{j=0}^{\alpha} \sum_{k=0}^{\beta} P_{j,k} B_j^{\alpha}(u) B_k^{\beta}(v), \tag{1.31}$$

$$q : \quad [0,1]^2 \mapsto \mathbb{R}^3, \quad q(u,v) = \sum_{j=0}^{\alpha} \sum_{k=0}^{\beta} Q_{j,k} B_j^{\alpha}(u) B_k^{\beta}(v). \tag{1.32}$$

Beide werden vollständig durch die Kontrollpunkte $P_{j,k}$ und $Q_{j,k}$ bestimmt. Zunächst fordern wir, daß P und Q eine gemeinsame Randkurve $c : [0,1] \to \mathbb{R}^3$ besitzen. Dies werden wir im folgenden auch als C^0–Bedingung bezeichnen. Durch die Forderung, daß die resultierende Fläche parametrisch glatt sein soll, würden wir unsere Möglichkeiten erheblich einschränken. Ähnlich wie bei den Kurven genügt es jedoch zu fordern, daß die Fläche "nur" geometrisch glatt ist.

Definition 1.6 (Geometrische Glattheit). *Es seien zwei Flächenstücke durch die regulären Parametrisierungen $p(u,v)$ und $q(u,v)$ $(0 \leq u, v \leq 1)$ mit einer gemeinsamen Randkurve*

$$c : [0,1] \to \mathbb{R}^3, \quad c(u) = p(u,0) = q(u,0) \text{ für } t \in [0,1] \tag{1.33}$$

gegeben. Die aus den Flächenstücken zusammengesetzte Fläche ist geometrisch glatt oder einmal stetig differenzierbar längs c, wenn gilt

$$\Phi p_u(\cdot,0) + \Psi p_v(\cdot,0) + q_v(\cdot,0) \equiv 0 \tag{1.34}$$

mit $\Phi : [0,1] \to \mathbb{R}$ und $\Psi : [0,1] \to \mathbb{R}^+$.

Dabei können natürlich beliebige Randkurven der beiden Flächenstücke verwendet werden. Betrachten wir (1.34) genauer, dann bedeutet dies zunächst, daß die Tangentialebenen von p und q in jedem Punkt von c übereinstimmen. Die Tangentenebene in $c(u)$ wird von den beiden Vektoren $[p_u(u,0), p_v(u,0)]$ bzw. $[q_u(u,0), q_v(u,0)]$ aufgespannt. Wegen (1.33) gilt $p_u(u,0) = \lambda(u) q_u(u,0)$. Hieraus folgt dann die Bedingung (1.34).

Offensichtlich genügt es jedoch nicht zu fordern, daß die Tangentialebenen übereinstimmen. Man könnte so z.B. die gleiche Fläche p oder Flächen die sich längs c tangential wie p verhalten zusammensetzen und so eine "Spitze" erhalten. Wir benötigen eine weitere Bedingung, die sicherstellt, daß die beiden Flächen auf "verschiedenen Seiten von c" liegen. Betrachten wir die Tangentialebene im Punkt $c(u_0)$, dann wird sie durch die durch $p_u(u_0,0)$, bzw. $q_u(u_0,0)$ erzeugte Gerade in zwei Teilebenen zerlegt. Wenn wir nun fordern, daß $p_v(u_0,0)$ und $q_v(u_0,0)$ auf verschiedenen Seiten dieser Geraden liegen, ergeben sich keine "Spitzen" mehr. Um dies sicherzustellen genügt es wie in obiger Definition zu fordern, daß $\Psi > 0$ ist.

Die Gleichung (1.34) ist noch sehr allgemein. Wählt man Φ und Ψ nach [Rei95], dann erhält man speziellere Glattheitsbedingungen, die wir im folgenden für die G–Splines verwenden werden.

Definition 1.7. *Die geometrische Glattheitsbedingung* (1.34) *ist*

i) vom Typ C, wenn $\Phi \equiv 0$ und $\Psi \equiv 1$;

ii) vom Typ G_I, wenn $\Phi \not\equiv 0$ und $\Psi \equiv 1$;

iii) vom Typ G_{II}, wenn $\Phi \equiv 0$ und $\Psi \not\equiv 1$.

Wir werden nun alle drei Typen für biquadratische p und q (d.h. $\alpha = \beta = 2$) etwas näher untersuchen.

Im Fall von Typ C erhalten wir die Bedingung für die parametrische Glattheit (C^1–Bedingung),

$$p_v(\cdot, 0) + q_v(\cdot, 0) \equiv 0. \tag{1.35}$$

p und q eingesetzt ergibt

$$\sum_{j=0}^{2} \sum_{k=0}^{2} (P_{j,k} + Q_{j,k})\, B_j^2(u) B_{k\,v}^2(0) = 0. \tag{1.36}$$

Es gilt $B_{0\,v}^2(0) = -2 = -B_{1\,v}^2(0)$ und $B_{2\,v}^2(0) = 0$ und aus der C^0–Bedingung folgt, daß $P_{j,0} = Q_{j,0}$ für $j = 0, 1, 2$. Also erhalten wir

$$\sum_{j=0}^{2} (2P_{j,0} - P_{j,1} - Q_{j,1})\, B_j^2(u) = 0. \tag{1.37}$$

Hieraus ergeben sich die folgenden Bedingungen für die Bézierpunkte:

$$P_{j,0} = Q_{j,0} = \frac{P_{j,1} + Q_{j,1}}{2} \text{ für } j = 0, 1, 2. \tag{1.38}$$

Bei einer Vierer–Nachbarschaftsstruktur, bei der die Flächenstücke an allen vier Randkurven geometrisch glatt zusammengesetzt werden, bestimmen damit die "mittleren" Bézierpunkte $P_{1,1}$ alle anderen Bézierpunkte. Siehe hierzu auch Abbildung 1.8. Hier können die "mittleren", mit nicht ausgefüllten Kreisen gekennzeichneten Bézierpunkte, frei gewählt werden und die mit ausgefüllten Kreisen markierten Bézierpunkte ergeben sich als Mittel der angrenzenden, nicht ausgefüllten Bézierpunkte. Damit lassen sich alle regulären Netze, die in diesem Fall die mittleren Bézierpunkte bestimmen, in Bézierkontrollnetze umwandeln.

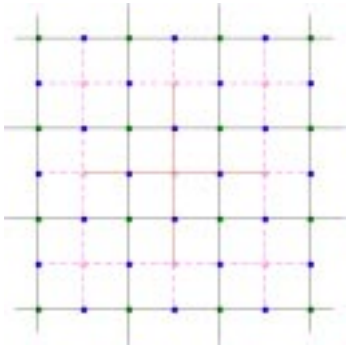


Abbildung 1.8: Kontrollpunkte für Typ C

Für den G_I Typ erhalten wir die Gleichung

$$\Phi p_u(\cdot,0) + p_v(\cdot,0) + q_v(\cdot,0) \equiv 0. \tag{1.39}$$

Nachdem $p_v(u,0)$ und $q_v(u,0)$ quadratisch in u sind und $p_u(u,0)$ linear in u ist, ist die Funktion Φ linear. Also sei $\Phi(u) := 2\gamma u$ mit $\gamma \in \mathbb{R}$. Setzt man dann (1.29) in (1.39) ein, erhält man folgende Gleichungen für die Kontrollpunkte,

$$\begin{aligned} P_{0,1} + Q_{0,1} &= 2P_{0,0}, \\ P_{1,1} + Q_{1,1} &= (2-\gamma)P_{1,0} + \gamma P_{0,0}, \\ P_{2,1} + Q_{2,1} &= (2-2\gamma)P_{2,0} + 2\gamma P_{1,0}. \end{aligned} \tag{1.40}$$

Für Typ G_{II} gilt die Gleichung

$$\Psi p_v(\cdot,0) + q_v(\cdot,0) \equiv 0. \tag{1.41}$$

Zur Lösung dieser Gleichung nehmen wir an, daß $p_v(u,0)$ und $q_v(u,0)$ im gleichen linearen Raum liegen, d.h. $p_v(u,0) = a(u)T$ und $q_v(u,0) = b(u)T$ mit $a, b : [0,1] \to \mathbb{R}$ und $T \in \mathbb{R}^3$. Dann ist die Gleichung für $\Psi = -\frac{b}{a}$ erfüllt. Eine sinnvolle Bedingung für die Bézierpunkte ist somit

$$\begin{aligned} \frac{P_{0,1} - P_{0,0}}{\kappa} &= P_{1,1} - P_{1,0} = P_{2,1} - P_{2,0} \\ &= P_{0,0} - Q_{0,1} = P_{1,0} - Q_{1,1} = P_{2,0} - Q_{2,1} \end{aligned} \tag{1.42}$$

mit $\kappa \in \mathbb{R}^+$. Aus $\kappa > 0$ folgt daß $\Psi = -\frac{q_v(\cdot,0)}{p_v(\cdot,0)} = \frac{1}{\kappa B_0^2 + B_1^2 + B_2^2} > 0$, wie ursprünglich gefordert.

1.7 Biquadratische G–Splines

Im vorherigen Abschnitt haben wir bereits eine Methode angesprochen, mit der man aus einem regulären Kontrollnetz Bézierkontrollnetze erzeugen kann, so daß die Fläche geometrisch glatt ist. Diese Methode läßt sich auf semi–reguläre Netze erweitern. Hieraus werden wir die biquadratischen G–Spline–Flächen erhalten. Die durch das Kontrollnetz repräsentierte Fläche soll durch eine Anzahl biquadratischer Bézierflächen dargestellt werden, d.h. zu jedem Punkt des Kontrollnetzes wird eine Bézierfläche mit neun Bézierpunkten erzeugt. Am Rand sollen diese Flächenstücke geometrisch glatt zusammengesetzt werden. Im regulären Fall können wir hierfür die Glattheitsbedingung von Typ C verwenden. In der Umgebung einer Irregularität der Ordnung $n \neq 4$ ist dies jedoch nicht möglich, nachdem hier mehr oder weniger als vier Flächenstücke zusammentreffen. An der Irregularität enthält das Kontrollnetz ein Polygon mit mehr oder weniger als vier Kanten. Dem Ansatz von [Rei95] folgend fordern wir in der Umgebung der Irregularität, daß die Glattheitsbedingungen vom Typ G_I oder G_{II} erfüllt sind, wie in Abbildung 1.9 gezeigt. Dabei stellen gepunktete Linien den Typ G_I, gestrichelte den Typ G_{II} und der Rest den Typ C dar.

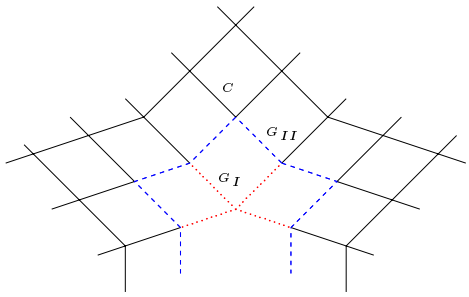


Abbildung 1.9: Glattheitsbedingungen in der Nähe einer Irregularität

In der Umgebung der Irregularität bezeichnen wir die Bézierpunkte wie in Abbildung 1.10 angegeben. Der Index j läuft dabei von 0 bis $n-1$ und wird modulo n hochgezählt. Wir fassen die Punkte vom gleichen Typ zu Vektoren zusammen, also $A = [A_0, A_1, \ldots, A_{n-1}]$, ... etc. Nach [Rei95] lassen sich aus den Kontrollpunkten D, F, J und L die Bézierkontrollnetze berechnen, wenn diese bestimmten Gleichungen genügen.

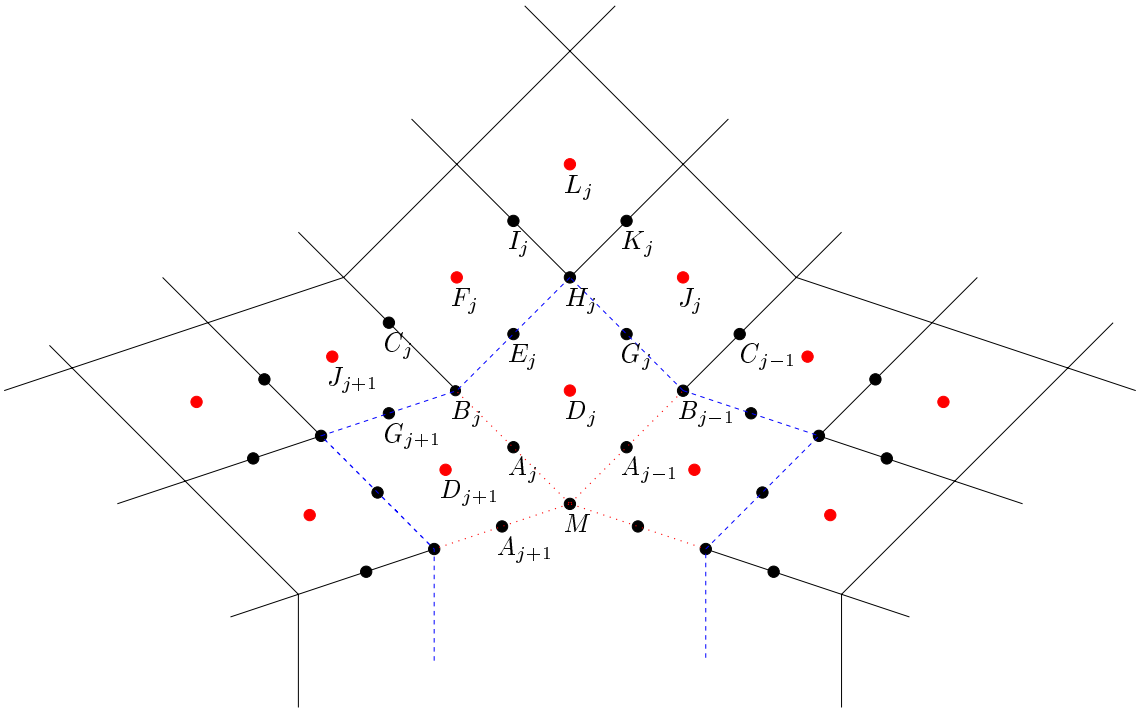


Abbildung 1.10: Bezeichnungen der Bézierpunkte in der Nähe einer Irregularität

Theorem 1.8. *Wenn die Kontrollpunkte die Gleichungen*

$$\begin{aligned} D - F - J + L &= 0, \\ (S - E)D + F - SJ &= 0, \\ T((S + \kappa E)D + (1 - \kappa)F) &= 0 \end{aligned} \tag{1.43}$$

mit dem Shift Operator $S : A_j \to A_{j+1}$ und der $(n-3) \times n$ Matrix T gegeben durch

$$T_{jk} := \begin{cases} \cos(\frac{2\pi jk}{n}) & \text{für } 2 \le j \le \frac{n}{2}, \\ \sin(\frac{2\pi jk}{n}) & \text{für } \frac{n}{2} < j \le n-2 \end{cases} \tag{1.44}$$

erfüllen, dann beschreiben die durch folgende Gleichungen gegebenen Bézierpunkte eine glatte, biquadratische Splinefläche:

$$\begin{aligned} A &= \frac{(S + \kappa E)D + (1 - \kappa)F}{2}, & G &= \frac{D + J}{2}, \\ B &= \frac{D + F + SD + SJ}{4}, & H &= \frac{D + F + J + L}{4}, \\ C &= \frac{F + SJ}{2}, & I &= \frac{F + L}{2}, \\ E &= \frac{D + F}{2}, & K &= \frac{J + L}{2}, \\ M &= \frac{1}{n}\sum_{j=0}^{n-1} A_j, & \kappa &= \frac{2}{2 - \cos\left(\frac{2\pi}{n}\right)}. \end{aligned} \tag{1.45}$$

Beweis. Der Beweis ist in [Rei95] zu finden. ∎

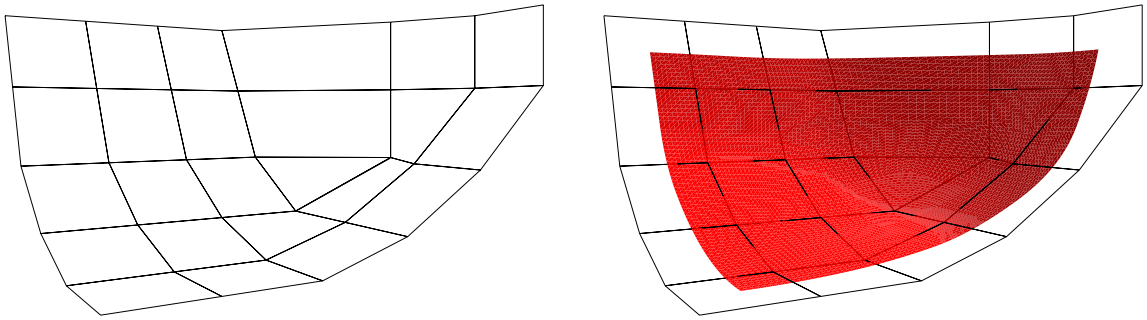
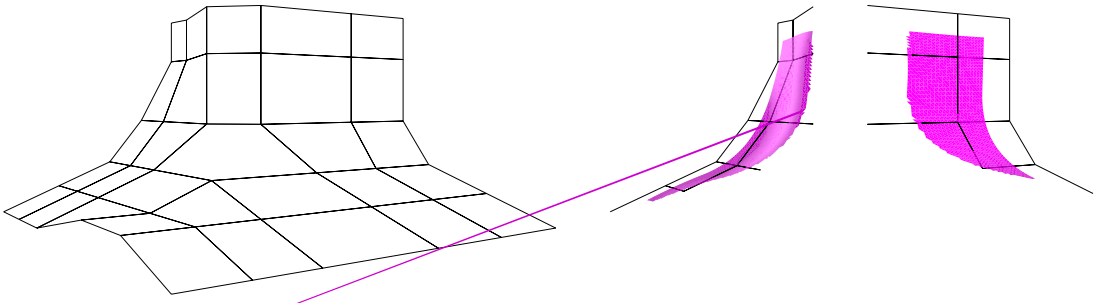


Abbildung 1.11: G–Spline–Fläche mit Irregularität der Ordnung 3ildun 112: plinelc



ptim

eunden dieses olem lieet die seudoin von

(148)

amit lassen sic also alle iepunte in de meun de eulaitt mit de leic un (148)
und dem eoem 18 aus elieien ontollpunten eec nen ie ildunen 111 und 112
eien das ontollnet und die eec nete plinelc e ontollnete mit eine eulaitt
de dn nd

18 loitmen iuadatisc e plines

i eden nun die loitmen u eec n un und
lc en vostellen au en i unc st eine at

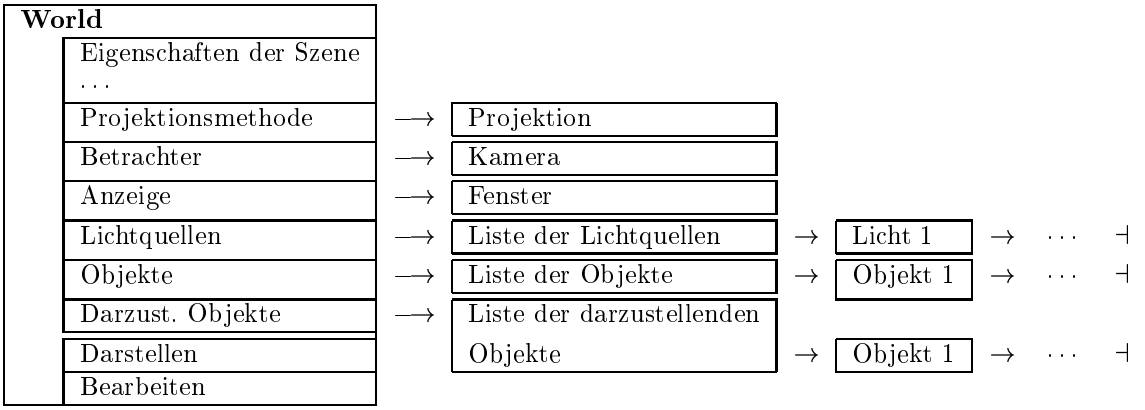


Abbildung 1.13: `World` Klasse

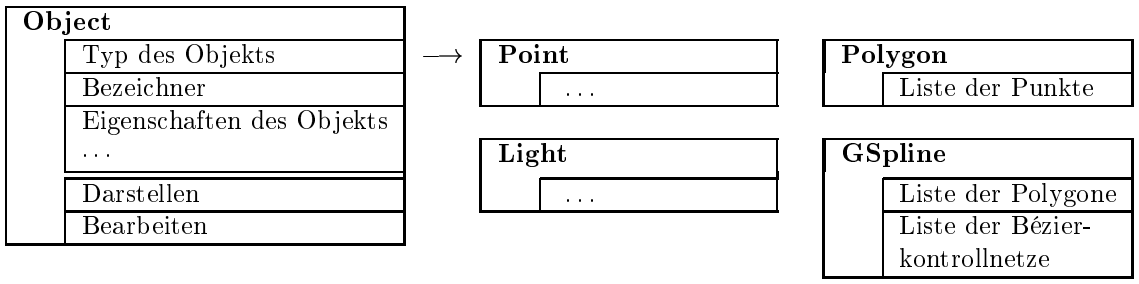


Abbildung 1.14: `Object` und davon abgeleitete Klassen

geometrischen Objekte für eine Szene beschreiben können. Wir beschränken uns momentan darauf, nur die grobe Struktur anzugeben. Die vollständige Dokumentation der Klassenstruktur in C++ ist im Anhang A zur Programm–Dokumentation beschrieben.

Eine einzelne Szene fassen wir zu einer `World` Klasse zusammen. Diese Klasse enthält zunächst die generellen Eigenschaften der Szene, Informationen zur Projektionsmethode, zum Betrachter und über das Fenster, in dem die Szene angezeigt werden soll. Weiter enthält sie eine Liste von grafischen Objekten, die zu der Szene gehören, eine Liste der Lichtquellen und schließlich eine Liste der Objekte, die angezeigt werden sollen. Die Objekte in der letzten Liste müssen auch in der Liste aller grafischen Objekte vorhanden sein. Es können durchaus Objekte zur Szene gehören, die nicht angezeigt, aber zu deren Aufbau benötigt werden. Dies können z.B. Teile eines Kontrollnetzes sein. Die Klasse stellt auch Methoden zur Darstellung und Bearbeitung der Objekte zur Verfügung. Dabei muß zunächst die Methode zur Bearbeitung der Objekte aufgerufen werden, die z.B. die G–Spline–Kontrollnetze in Bézierkontrollnetze umwandelt. Die Methode zur Darstellung zeichnet die vorher bearbeiteten, darzustellenden Objekte direkt. Abbildung 1.13 zeigt den groben Aufbau dieser Klasse.

Die grafischen Objekte können Punkte, Polygone, G–Splines, etc. sein, aber auch Lichtquellen können so beschrieben werden. Die Lichtquellen werden in der `World` Klasse aus Effizienzgründen in einer speziellen Liste eingetragen. Die Klassen für diese Objekte werden alle von der Klasse `Object` abgeleitet. Dabei enthält `Object` die allgemeinen Eigenschaften eines solchen grafischen Objekts. Dies sind z.B. ein Bezeichner für das Objekt, die Materialeigenschaften und Methoden zum Darstellen, Bearbeiten, etc. des Objekts. Für die speziellen Typen der Objekte kommen dann weitere spezielle Eigenschaften hinzu. Teilweise werden auch spezielle Methoden zum Darstellen und Bearbeiten des Objekts zur Verfügung gestellt. Für ein Polygon speichern wir z.B. als zusätzliche Eigenschaft eine Liste von Punkt–Objekten, die die Kanten des Polygons beschreiben. Ein G–Spline–Objekt enthält dann eine Liste von Polygon–Objekten, aus denen das G–Spline–Kontrollnetz aufgebaut ist. Weiter enthält es eine Liste von Bézierkontrollnetzen für biquadratische Bézierflächen, die aus dem Kontroll-

netz erzeugt wurden. Wir werden später weitere Eigenschaften und Objekte hinzufügen. Die grobe Struktur der `Object` Klassen wird in Abbildung 1.14 dargestellt.

Ein Kontrollnetz einer G–Spline–Fläche wird durch Punkte dargestellt, die durch Polygone zu einem Netz verbunden sind. Soweit die Fläche selbst orientierbar ist, sollte jedes dieser Polygone die gleiche Orientierung besitzen. Für einen einfachen G–Spline–Algorithmus benötigen wir die Orientierung allerdings nicht. Sie wird aber für bestimmte Darstellungen von Funktionen auf den Flächen notwendig. Auch können wir die Orientierung bei der Berechnung des G–Splines ausnutzen. Wir setzen aber nicht voraus, daß die Polygone gleich mit der richtigen Orientierung vorgegeben wurden. Der Algorithmus `gspline_orientation` orientiert die Polygone eines G–Spline–Kontrollnetzes.

Algorithmus 1.1. `gspline_orientation`
G–Spline–Kontrollnetz orientieren

I. Falls das Kontrollnetz der G–Spline–Fläche schon als orientiert markiert ist, ist nichts zu tun.

II. Erzeuge eine Liste `pfl`, die zu jedem Kontrollpunkt die Polygone angibt, die diesen Punkt enthalten. Dazu bearbeite jedes Polygon `facet` des G–Spline–Kontrollnetzes einzeln:

   A. Für jeden Punkt eines solchen Polygons unterscheide die beiden Fälle:

      1. Wenn der Punkt noch nicht in der Liste `pfl` vorhanden ist, füge ein neues Element für diesen Punkt mit dem Polygon `facet` hinzu.
      2. Wenn der Punkt schon in der Liste vorhanden ist, füge das Polygon `facet` zu dem Element hinzu.

III. Markiere alle Polygone als nicht orientiert.

IV. Orientiere jedes Polygon `allpolygon` des G–Spline–Kontrollnetzes. Dazu führe Folgendes für jedes `allpolygon` aus:

   A. Falls das Polygon `allpolygon` nicht orientiert ist, verwende die Orientierung des Polygons `allpolygon` um alle Polygone zu orientieren, die in dem zusammenhängenden Teilnetz von `allpolygon` liegen:

      1. Markiere `allpolygon` als orientiert.
      2. Falls vom Benutzer gefordert wird die Orientierung der Fläche zu ändern, drehe die Reihenfolge der Punkte in `allpolygon` um.
      3. Füge `allpolygon` in die leere Polygonliste `poly_list` ein.
      4. Nun orientiere alle Polygone, die zum zusammenhängenden Teilnetz von `allpolygon` gehören. Dazu wiederhole Folgendes, solange `poly_list` nicht leer ist:
         i. Entferne das erste Element `polygon` aus `poly_list`.
         ii. Finde für jede Kante von `polygon` die anderen Polygone, die diese Kante enthalten. Benutze dazu die am Anfang erzeugte Liste `pfl`.
         iii. Jedes dieser anderen Polygone `polygon2` muß die entsprechende Kante in die entgegengesetzte Richtung durchlaufen, damit die Orientierung korrekt ist. Unterscheide hierfür folgende Fälle:
            a. Wenn die Kante in `polygon2` entgegengesetzt durchlaufen wird und `polygon2` noch nicht als orientiert markiert ist, dann markiere es als orientiert und füge es zur Liste `poly_list` hinzu.
            b. Wenn die Kante in `polygon2` entgegengesetzt durchlaufen wird und `polygon2` als orientiert markiert ist, ignoriere es.
            c. Wenn die Kante in `polygon2` in der gleichen Richtung durchlaufen wird und `polygon2` als nicht orientiert markiert ist, dann drehe die Reihenfolge der Punkte in `polygon2` um, markiere es als orientiert und füge es zur Liste `poly_list` hinzu.

d. Wenn die Kante in `polygon2` in der gleichen Richtung durchlaufen wird und `polygon2` als orientiert markiert ist, dann ist die Fläche nicht orientierbar. Gib eine entsprechende Warnung aus.

Damit obiger Algorithmus funktioniert, muß die Fläche orientierbar sein. Bei nicht orientierbaren Flächen wird lediglich die Anzahl der Kanten, an denen das Kontrollnetz nicht orientiert ist, reduziert. Es bleiben die Kanten übrig, für die der Fall d. von IV–A–4 zutrifft. Dies kann an beliebigen Stellen im Kontrollnetz erfolgen, ohne daß diese Kanten miteinander verbunden sind und hängt von der Reihenfolge ab, in der die Polygone bearbeitet werden.

Normalerweise nehmen wir auch an, daß nur maximal zwei Polygone eine gemeinsame Kante haben. Der Algorithmus funktioniert aber auch, wenn es mehr Polygone sind, allerdings kann man dann genau genommen nicht mehr von einer Orientierung sprechen. Spätestens bei der Umwandlung des Kontrollnetzes in Bézierpunkte werden an solchen Stellen nur noch zwei Polygone berücksichtigt. Welche dies sind, ist dabei nicht festgelegt und hängt wieder von der Reihenfolge ab, in der die Polygone bearbeitet werden.

Es bietet sich auch an, die Liste `pfl` durch eine weiteres Objekt zu repräsentieren, da wir solche und ähnliche Listen häufiger benötigen. Wir werden sie also durch eine Klasse implementieren, die Methoden zur Umwandlung des Kontrollnetzes in eine solche Liste enthält.

Unser nächstes Ziel ist die Entwicklung eines Algorithmus, der die Kontrollpunkte für die G–Spline–Fläche in Bézierpunkte für biquadratische Bézierflächen nach Theorem 1.8 umwandelt. Hier ist das Hauptproblem, die entsprechenden Punkte in dem über Polygone definierten semi–regulären Kontrollnetz zu finden. Dazu behandeln wir zunächst alle irregulären Stellen des Netzes. Jedes Polygon p des Kontrollnetzes mit mehr oder weniger als vier Punkten beschreibt eine Irregularität. Die Punkte in der Umgebung dieses Polygons werden unter Berücksichtigung der Orientierung durch folgenden Algorithmus bestimmt und über Theorem 1.8 werden die Bézierpunkte berechnet. Dabei werden zunächst nur die Flächenstücke für die Punkte des Polygons p berechnet und die Quasikontrollpunkte gespeichert. Denn die Gleichungen für den äußeren Ring um p stimmen mit dem regulären Fall überein, falls die Quasikontrollpunkte verwendet werden. Nachdem alle Irregularitäten behandelt wurden, werden die restlichen Bézierpunkte für den regulären Fall berechnet, wobei die entsprechenden Quasikontrollpunkte und die schon berechneten Bézierpunkte berücksichtigt werden.

Algorithmus 1.2. `process_gspline_object`
G–Spline–Kontrollnetz in Bézierkontrollnetze umwandeln

I. Orientiere das G–Spline–Kontrollnetz mit `gspline_orientation`.

II. Erzeuge eine Liste `pfl`, die zu jedem Kontrollpunkt die Ordnung und die Polygone angibt, die diesen Punkt enthalten. Bearbeite dazu jedes Polygon `facet` des G–Spline–Kontrollnetzes einzeln:

   A. Bestimme zunächst die Anzahl `n` der Punkte des Polygons.

   B. Für jeden Punkt des Polygons unterscheide die beiden Fälle:

      1. Wenn der Punkt schon in der Liste vorhanden ist, füge das Polygon `facet` zu dem Element hinzu:

         i. Falls `n` nicht 4 ist, setze die Ordnung des Punktes in der Liste auf `n`. Sollte die Ordnung des Punktes in der Liste vorher schon von 4 verschieden sein, dann liegen zwei irreguläre Teile des Netzes nebeneinander. Beende in diesem Fall den Algorithmus mit einer entsprechenden Fehlermeldung.

         ii. Falls das Element für den Punkt in `pfl` wenigstens 4 Polygone auflistet, die diesen Punkt enthalten, dann beende den Algorithmus ebenfalls mit einer entsprechenden Fehlermeldung.

         iii. Füge `facet` dem Element für den Punkt in `pfl` hinzu.

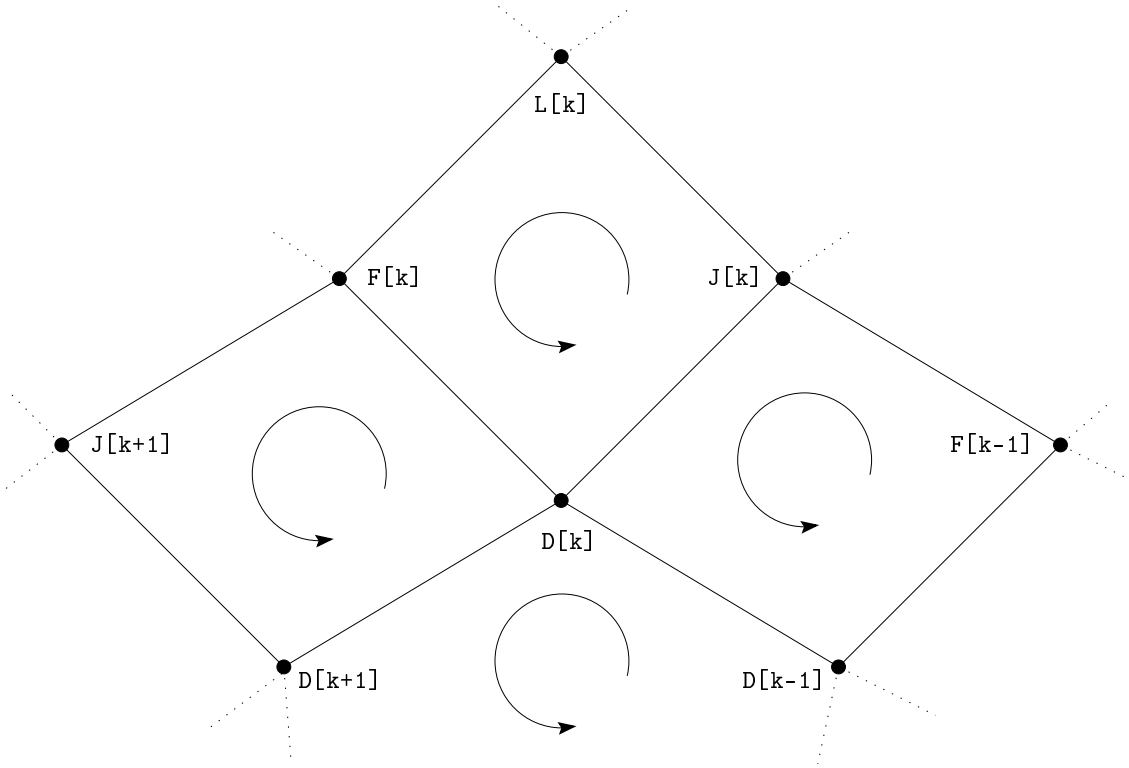


Abbildung 1.15: G–Spline–Kontrollpunkte im irregulären Fall

2. Wenn der Punkt noch nicht in der Liste vorhanden ist, erzeuge ein neues Element in der Liste für diesen Punkt mit dem Polygon `facet` und der Ordnung `n`.

III. Berechne die Bézierkontrollnetze für die irregulären Teile des G–Spline–Kontrollnetzes. Betrachte dazu jeden Punkt `p` aus der Liste `pfl` mit der Ordnung `n` ungleich 4, der in 4 Polygone des Kontrollnetzes enthalten ist und noch nicht bearbeitet wurde. Die Bézierkontrollnetze für diese Irregularität werden wie folgt erzeugt:

A. Finde das Polygon `fi`, das genau `n` Punkte enthält, von denen einer `p` ist. Dies ist das Polygon welches die Irregularität im Netz erzeugt.

B. Bestimme die Punkte `D` (vgl. Abbildung 1.15) als Punkte von `fi`. Die Numerierung sei durch die Reihenfolge der Punkte in `fi` beginnend mit `p` festgelegt.

C. Markiere alle Punkte `D` als bearbeitet.

D. Wenn einer der Punkte `D` nicht genau in 4 Polygonen enthalten ist, fahre mit dem nächsten Punkt fort, da die Irregularität nicht vollständig ist.

E. Finde das Polygon `D[0]`, `J[0]`, `L[0]`, `F[0]` (vgl. Abbildung 1.15). Dies ist das einzige Polygon, das `D[0]` enthält, aber nicht `D[1]` und `D[n-1]`. Nehme zunächst an, daß die Punkte `J[0]`, `L[0]`, `F[0]` dem Punkt `D[0]` in dieser Reihenfolge im Polygon folgen.

F. Finde den Punkt `J[1]`. Suche dazu das Polygon, welches `D[1]`, `D[0]` und `F[0]` in dieser Reihenfolge enthält. Existiert dieses Polygon nicht, dann stimmt die vorher gewählte Orientierung an dieser Stelle nicht. Suche in diesem Fall das Polygon mit der Kante von `D[1]` nach `D[0]` und vertausche `J[0]` und `F[0]`. In beiden Fällen ist `J[1]` der fehlende Punkt des Polygons.

G. Bestimme die restlichen Punkte `F[k]`, `L[k]`, `J[k+1]` für $\texttt{k} = 1 : n-1$ (vgl. Abbildung 1.15):

1. Finde das Polygon `D[k]`, `J[k]`, `L[k]`, `F[k]`. Dazu suche zunächst das Polygon, das die Kante `D[k]`, `J[k]` enthält. Folgt diesen Punkten im Polygon nicht `F[k-1]`, dann sind die nächsten beiden Punkte des Polygons `L[k]` und `F[k]`. Wird dieses Polygon nicht gefunden, dann stimmt die Orientierung an dieser Stelle nicht. Suche in diesem Fall das Polygon mit der Kante von `J[k]` nach `D[k]`. Folgt auf diese Kante im Polygon nicht `D[k-1]`, dann sind die nächsten beiden Punkte des Polygon `F[k]` und `L[k]`.
2. Ist `k` kleiner als `n` − 1, dann bestimme `J[k+1]`. Dazu suche zunächst das Polygon mit der Kante von `D[k]` nach `F[k]`. Ist der nächste Punkt in diesem Polygon nicht der Punkt `D[k+1]`, dann ist dieser `J[k+1]`. Existiert ein solches Polygon nicht, dann stimmt die Orientierung an dieser Stelle nicht. In diesem Fall suche das Polygon mit der Kante von `D[k]` nach `D[k+1]`. Folgt hier nicht der Punkt `F[k]`, dann ist dies der Punkt `J[k+1]`.

H. Berechne die Quasikontrollpunkte `Dv`, `Fv`, `Jv`, `Lv` aus `D`, `F`, `J`, `L` über (1.48).

I. Berechne die Bézierpunkte `A`, `B`, `C`, `E`, `G`, `H`, `I`, `K`, `M` aus den Quasikontrollpunkten über die Gleichungen (1.45).

J. Erzeuge für jeden Kontrollpunkt das entsprechende Bézierkontrollnetz aus den bisher berechneten Punkten und füge es zur Liste der Bézierkontrollnetze des G–Splines mit einer Referenz auf den Kontrollpunkt hinzu.

K. Speichere für jeden Kontrollpunkt `D`, `F`, `J`, `L` zusätzlich die Position des zugehörigen Quasikontrollpunktes. Diese wird dazu verwendet den äußeren Ring der Bézierpunkte im nächsten Teil des Algorithmus' zu berechnen.

IV. Berechne die Bézierkontrollnetze für die regulären (noch nicht bearbeiteten) Teile des G–Spline–Kontrollnetzes. Hierzu wird ein Bézierkontrollnetz für jeden Punkt `m` aus der Liste `pfl` der Ordnung 4 erzeugt, der in 4 Polygonen des G–Spline–Kontrollnetzes enthalten ist und noch nicht bearbeitet wurde (vgl. Abbildung 1.17):

A. Wähle ein beliebiges Polygon `f0` aus, das `m` enthält und markiere dieses Polygon als benutzt.

B. Wähle Punkte `n0`, `nn0`, `n1` als die Punkte, die `m` in der Punktliste des Polygons in dieser Reihenfolge folgen.

C. Finde das Polygon `f1`. Dazu suche in allen nicht als benutzt markierten Polygonen, die `m` enthalten, die Kante von `m` nach `n1`. Nach diesen beiden Punkten folgen die Punkte `nn1` und `n2`. Wird die Kante in die entgegengesetzte Richtung durchlaufen, stimmt die Orientierung an dieser Stelle nicht. In diesem Fall folgen auf die Kante von `n1` nach `m` die Punkte `n2`, `nn1`. Markiere schließlich das Polygon `f1` als benutzt.

D. Finde das Polygon `f2` analog zu `f1`, wobei die gesuchte Kante nun `m`, `n2` ist. `nn2`, `n3` folgen entsprechend der Orientierung.

E. Finde `nn3` in dem noch nicht als benutzt markierten Polygon, das `m` enthält. Unabhängig von der Orientierung ist `nn3` in diesem Polygon der zweite Punkt nach `m`.

F. Erzeuge das Bézierkontrollnetz für `m` nach (1.38) für den regulären Fall. Dazu werden im wesentlichen die Mittel der für die Bézierpunkte zuständigen Kontrollpunkte berechnet. Falls allerdings Quasikontrollpunkte für die Kontrollpunkte im irregulären Fall berechnet wurden oder schon Bézierkontrollnetze für die angrenzenden Flächenstücke existieren, verwende die Koordinaten dieser Punkte. Sollten für die vorher bestimmten Punkte schon Bézierkontrollnetze berechnet worden sein, dann verwende die Koordinaten des mittleren Bézierpunktes. Falls ein Quasikontrollpunkt für diese Punkte berechnet wurde, verwende die Koordinaten des Quasikontrollpunktes. Sonst verwende die Koordinaten des Kontrollpunktes.

In Schritt III–H wird zur Berechnung der Quasikontrollpunkte die Matrix $E - P^+P$ für die Gleichung (1.48) benötigt. Da diese Matrix nur von der Ordnung der Irregularität abhängt, genügt es aber, sie pro benötigter Ordnung nur einmal zu berechnen und sie für weitere Berechnungen zu speichern.

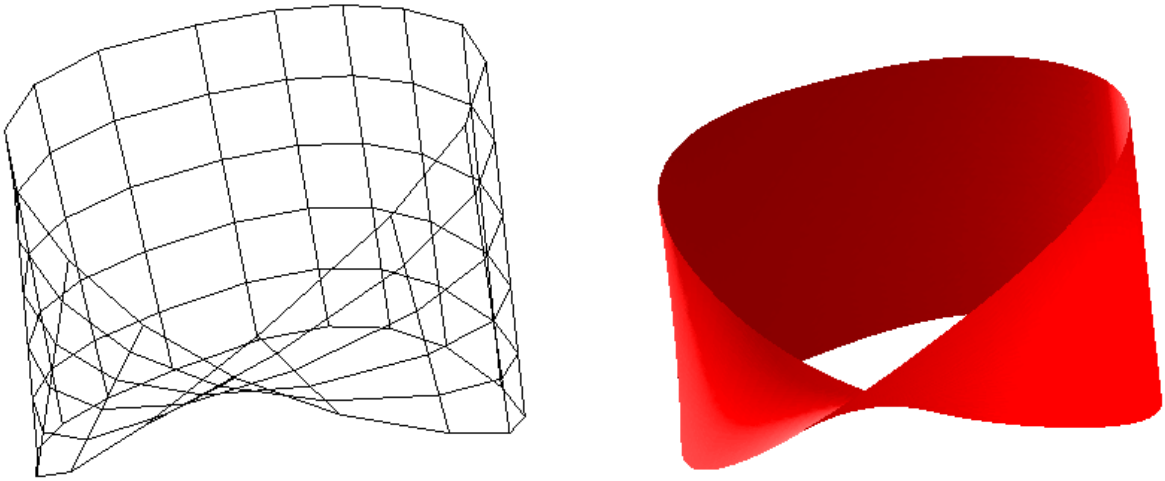


Abbildung 1.16: Möbiusband

Wurde die G–Spline–Fläche in biquadratische Bézierkontrollnetze umgewandelt, läßt sich die Fläche durch Standardalgorithmen für Bézierflächen darstellen. Auf den genauen Algorithmus zur Visualisierung gehen wir deshalb hier nicht näher ein.

Abbildung 1.16 zeigt ein Möbiusband, das durch ein reguläres Kontrollnetz beschrieben wird. Dieses Netz wurde allerdings schon mit dem im nächsten Abschnitt beschriebenen Doo–Sabin–Subdivision–Algorithmus aus einem vereinfachten Kontrollnetz erzeugt.

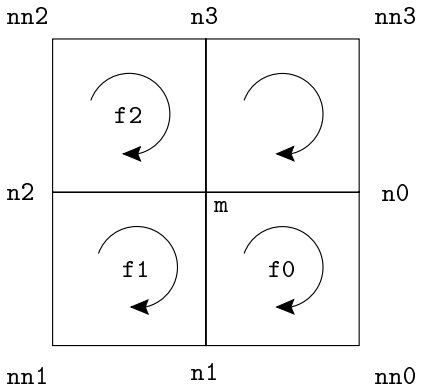


Abbildung 1.17: G–Spline–Kontrollpunkte im regulären Fall

1.9 Doo–Sabin–Subdivision

Liegen zwei irreguläre Polygone eines Kontrollnetzes direkt nebeneinander oder liegt nur ein reguläres Polygon zwischen ihnen, dann können wir an dieser Stelle den G–Spline–Algorithmus nicht einsetzen. Um ein solches Kontrollnetz doch verwenden zu können, müssen wir es zunächst verfeinern, indem wir die vorhandenen Polygone durch mehrere, kleinere Polygone ersetzen.

Eine Möglichkeit hierzu bietet der Doo–Sabin–Subdivision–Algorithmus (vgl. [Doo78]). Basierend auf den Verfeinerungstechniken für biquadratische, uniforme B–Spline–Flächen entwickelten Donald Doo und Malcolm Sabin einen Subdivision–Algorithmus, der zur Verfeinerung von Kontrollnetzen beliebiger Topologie verwendet werden kann. Die neuen Kontrollpunkte werden dabei als Mittel von vier durch das jeweilige Polygon bestimmten Punkten berechnet. Im folgenden sei der Kantenpunkt einer gegebenen Kante der Mittelpunkt dieser Kante und der Gitterpunkt eines Polygons sei das Mittel aller Punkte dieses Polygons.

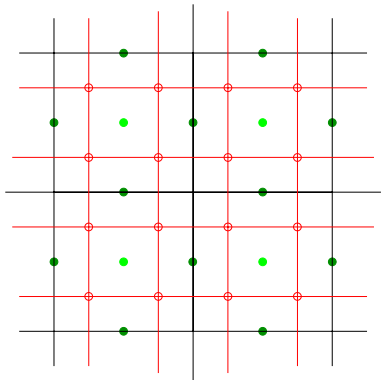


Abbildung 1.18: Doo–Sabin–Algorithmus

Beim Doo–Sabin–Algorithmus wird das alte Kontrollnetz vollständig durch ein neues ersetzt, wobei pro Kontrollpunkt und Polygon ein neuer Kontrollpunkt erzeugt wird, der das Mittel vom alten Kon-

trollpunkt, dem Gitterpunkt und den zwei Kantenpunkten ist. Diese Punkte werden dann zu einem neuen Kontrollnetz zusammengesetzt, so daß jedes ursprüngliche Polygon durch ein kleineres ersetzt wird und für jede Kante und jeden Kontrollpunkt ein neues Viereck entsteht. Mit diesen neuen Vierecken werden die verkleinerten Polygone miteinander verbunden. Dieser Prozeß ist in Abbildung 1.18 illustriert. Dabei sind die ausgefüllten Punkte in der Mitte der Quadrate die Gitterpunkte, die ausgefüllten Punkte auf den Linien sind die Kantenpunkte und die nicht ausgefüllten Punkte markieren die neuen Kontrollpunkte.

Algorithmus 1.3. `doo_sabin`
Doo–Sabin–Subdivision

I. Erzeuge eine Liste `pfl`, die zu jedem Kontrollpunkt die Polygone angibt, die diesen Punkt enthalten. Mit Hilfe von `pfl` können im folgenden die zu den Punkten gehörenden Polygone gefunden werden. Dazu bearbeite jedes Polygon `facet` des G–Spline–Kontrollnetzes einzeln und unterscheide für jeden Punkt eines solchen Polygons die beiden Fälle:

   A. Wenn der Punkt noch nicht in der Liste vorhanden ist, erzeuge ein neues Element in der Liste für diesen Punkt mit dem Polygon `facet`.

   B. Wenn der Punkt schon in der Liste vorhanden ist, füge das Polygon `facet` dem entsprechenden Element hinzu.

II. Erzeuge eine Liste `efl`, die zu jeder Kante die Polygone angibt, die diese Kante enthalten. Mit Hilfe von `efl` können im folgenden die zu den Kanten gehörenden Polygone gefunden werden. Dazu bearbeite jedes Polygon `facet` des G–Spline–Kontrollnetzes einzeln und unterscheide für jede Kante eines solchen Polygons die Fälle:

   A. Wenn die Kante noch nicht in der Liste ist, erzeuge ein neues Element für diese Kante und gebe `facet` als erstes Polygon für diese Kante an.

   B. Wenn schon ein Element für die Kante in der Liste vorhanden ist und bis jetzt nur ein Polygon für diese Kante aufgelistet wird, dann füge `facet` als zweites Polygon hinzu.

   C. Wenn schon ein Element für die Kante in der Liste vorhanden ist und bereits zwei Polygone für diese Kante aufgelistet werden, dann gib eine Warnung aus und ignoriere `facet` für diese Kante. Eigentlich ist in diesem Fall ein Fehler in dem Kontrollnetz vorhanden, da für eine Fläche eine Kante maximal in zwei Polygonen vorhanden sein darf. Es genügt aber dies einfach zu ignorieren und den Benutzer darüber zu informieren.

III. Erzeuge für jeden Kontrollpunkt `P` und jedes über `pfl` bestimmte Polygon `facet`, das diesen Punkt enthält, einen neuen Kontrollpunkt `Q` als Mittel der Kantenpunkte, des Gitterpunktes und `P`:

   A. Bestimme den Punkt `n` nach `P` und den Punkt `p` vor `P` in `facet`.

   B. Bestimme die Kantenpunkte als Mittel von `n` und `P` bzw. von `p` und `P`.

   C. Bestimme den Gitterpunkt als Mittel aller Punkte von `facet`.

   D. Der neue Kontrollpunkt `Q` ist das Mittel der beiden Kantenpunkte, des Gitterpunktes und `P`. Speichere `Q` als neuen Kontrollpunkt für `P` und `facet`.

IV. Verbinde für jedes Polygon `poly` die neuen, für dieses Polygon erzeugten Punkte:

   A. Erzeuge ein neues Polygon–Objekt `npoly`.

   B. Die für `poly` erzeugten neuen Kontrollpunkte bilden die Punkte von `npoly`.

   C. Füge `npoly` dem neuen Kontrollnetz hinzu.

V. Verbinde für jeden Kontrollpunkt `P` die für diesen Punkt erzeugten, neuen Kontrollpunkte:

   A. Erzeuge ein neues Polygon–Objekt `npoly`.

B. Die für `P` erzeugten, neuen Kontrollpunkte bilden die Punkte von `npoly`. Die neuen Punkte müssen entsprechend der Anordnung der zugehörigen Polygone miteinander verbunden werden.

C. Falls das soeben erzeugte Polygon nicht geschlossen ist, also an einem Rand des alten Kontrollnetzes liegt, dann lösche `npoly`. Sonst füge `npoly` dem neuen Kontrollnetz hinzu.

VI. Verbinde für jede Kante `E` die neuen Punkte, die zu den angrenzenden Polygonen gehören:

A. Erzeuge ein neues Polygon–Objekt `npoly`.

B. Finde die beiden Polygone, die die Kante `E` enthalten.

C. Die neuen Kontrollpunkte, die für die beiden Polygone und die Endpunkte der Kante `E` erzeugt wurden, bilden die Punkte für `npoly`.

D. Füge `npoly` dem neuen Kontrollnetz hinzu.

VII. Ersetze das alte Kontrollnetz durch das neue Kontrollnetz.

Zwei irreguläre, nebeneinander liegende Polygone des alten Kontrollnetzes werden durch Ausführen dieses Algorithmus' durch ein reguläres Polygon getrennt. Führt man den Algorithmus zweimal aus, sind sicher alle Irregularitäten durch wenigstens zwei reguläre Polygone getrennt und der G–Spline–Algorithmus kann auf das Kontrollnetz angewendet werden.

In Abbildung 1.19 wird ein einfacher Würfel gezeigt, auf den dann zweimal der Doo–Sabin–Algorithmus angewendet wurde. Das Bild oben zeigt das ursprüngliche Kontrollnetz, unterhalb wird dann das ursprüngliche Kontrollnetz mit dem ersten Doo–Sabin–Schritt und das Kontrollnetz nach zwei Doo–Sabin–Schritten gezeigt. Unten in der Mitte ist die hierdurch erzeugte G–Spline–Fläche dargestellt. Weitere Beispiele hierzu sind die Abbildungen 1.20, 1.21, 1.22, 1.23 und 1.24. Alle ursprünglichen Kontrollnetze dieser Beispiele wurden durch Zusammensetzen von Würfeln erzeugt, von denen einzelne um eine Achse verdreht wurden (siehe 4.3.1).

Neben dem Doo–Sabin–Subdivision–Algorithmus gibt es auch noch die Catmull–Clark–Subdivision. Dieser Subdivision–Algorithmus verallgemeinert die bikubische, uniforme B–Spline–Verfeinerung für beliebige Topologien. Wir werden diesen Algorithmus allerdings nicht implementieren. Für eine genaue Beschreibung sei auf [CC78] verwiesen.

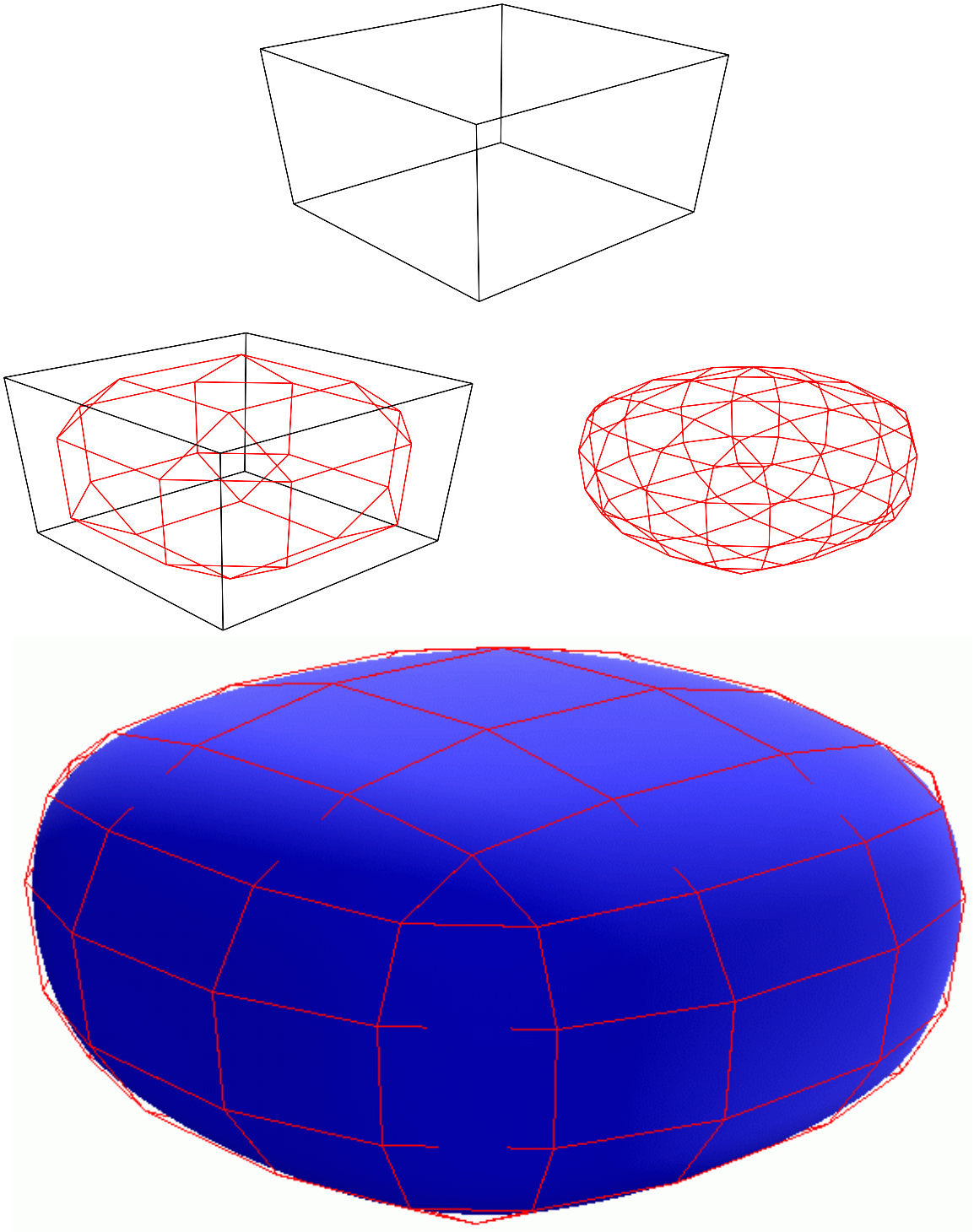


Abbildung 1.19: Doo–Sabin: Einfacher Würfel

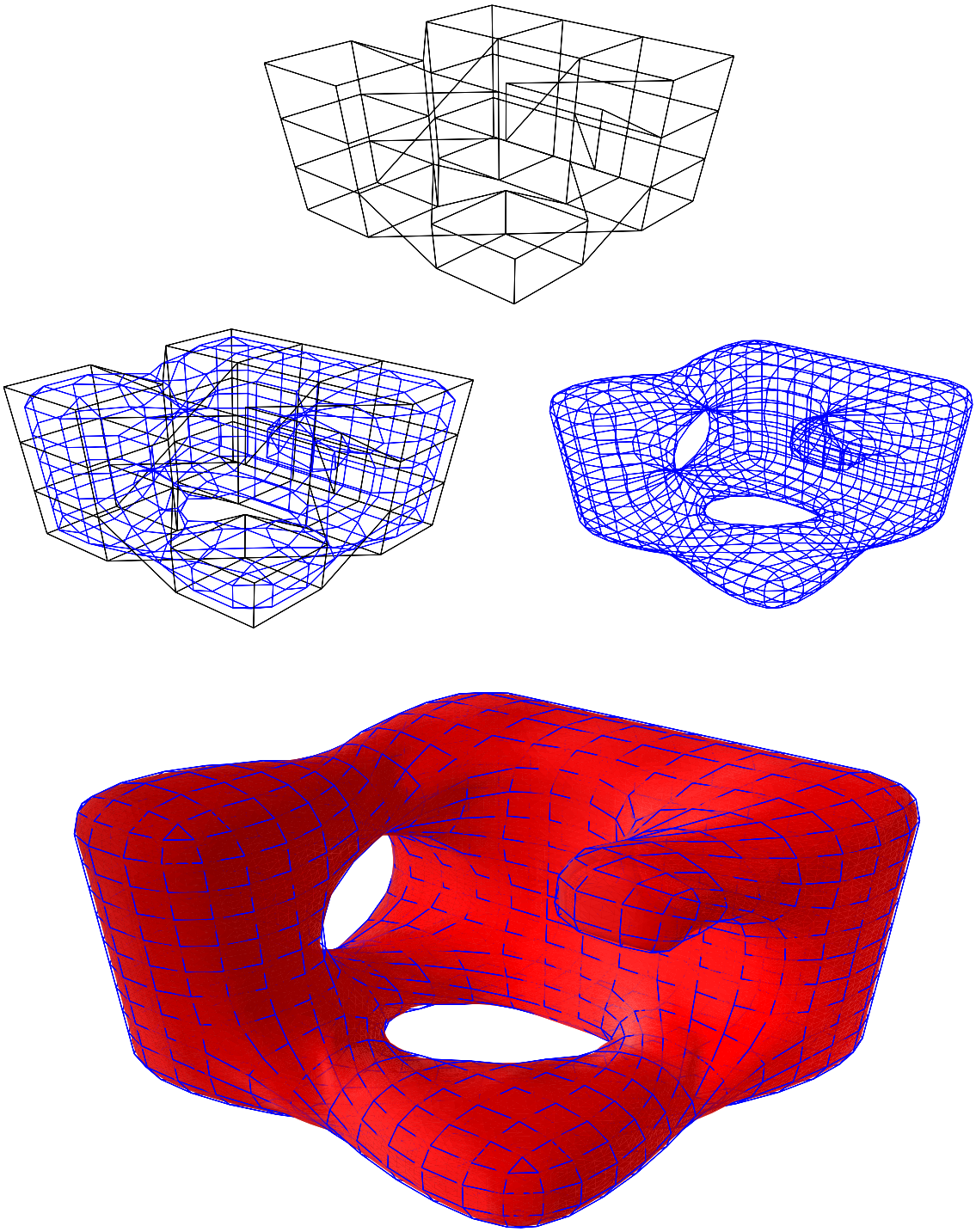


Abbildung 1.20: Doo–Sabin: Verdrehter Würfel

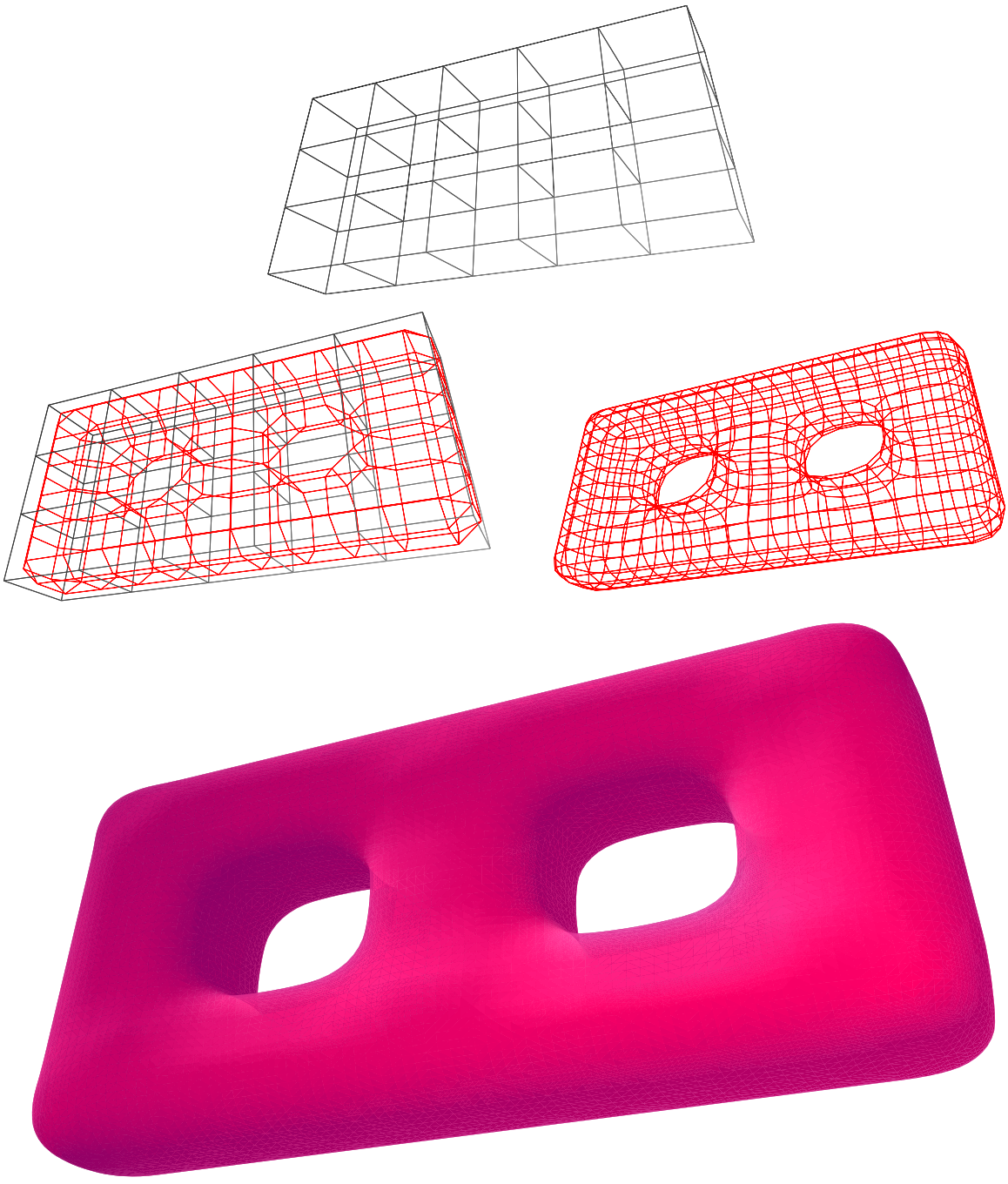


Abbildung 1.21: Doo–Sabin: Acht

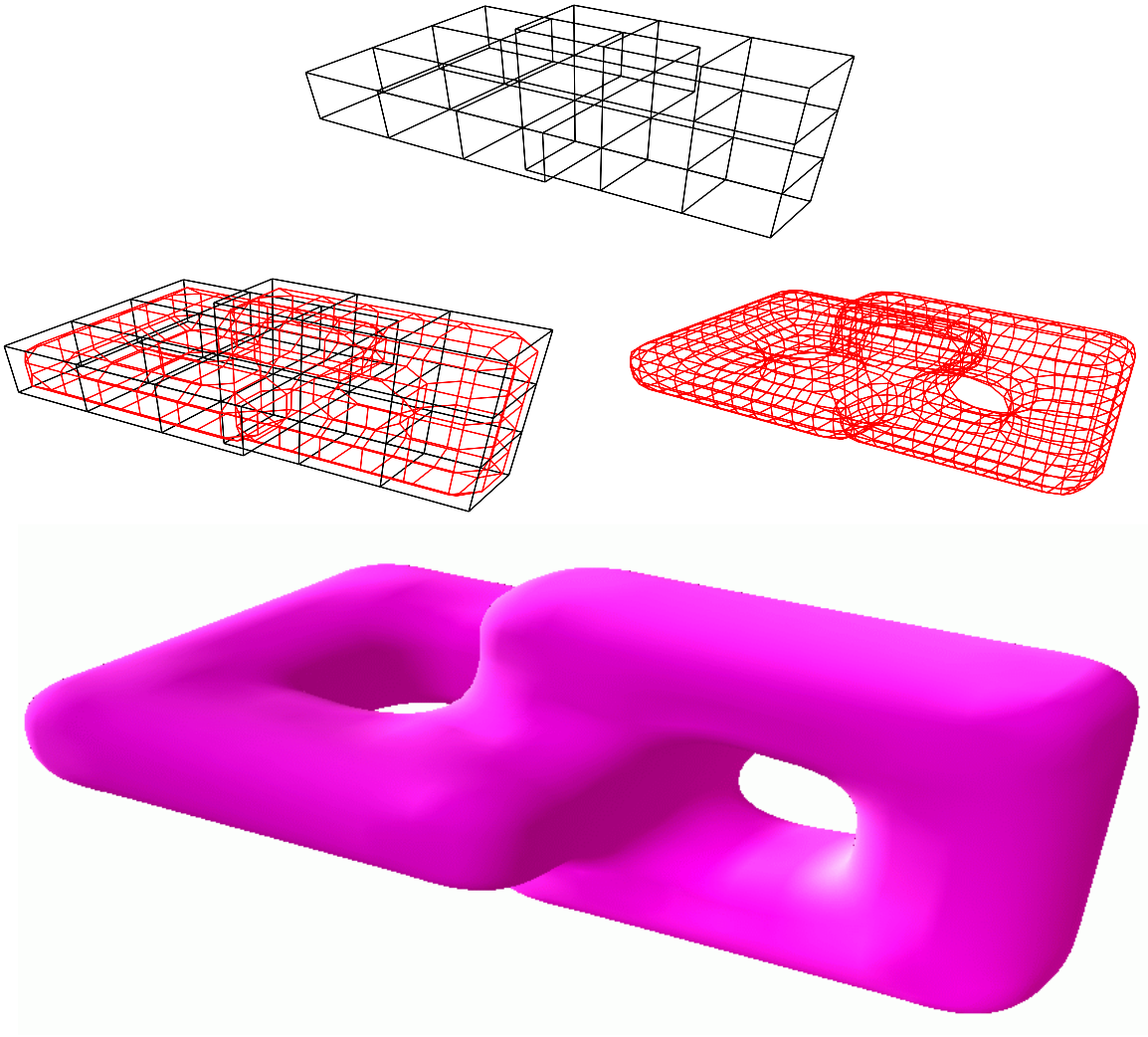


Abbildung 1.22: Doo–Sabin: Verdrehte Acht

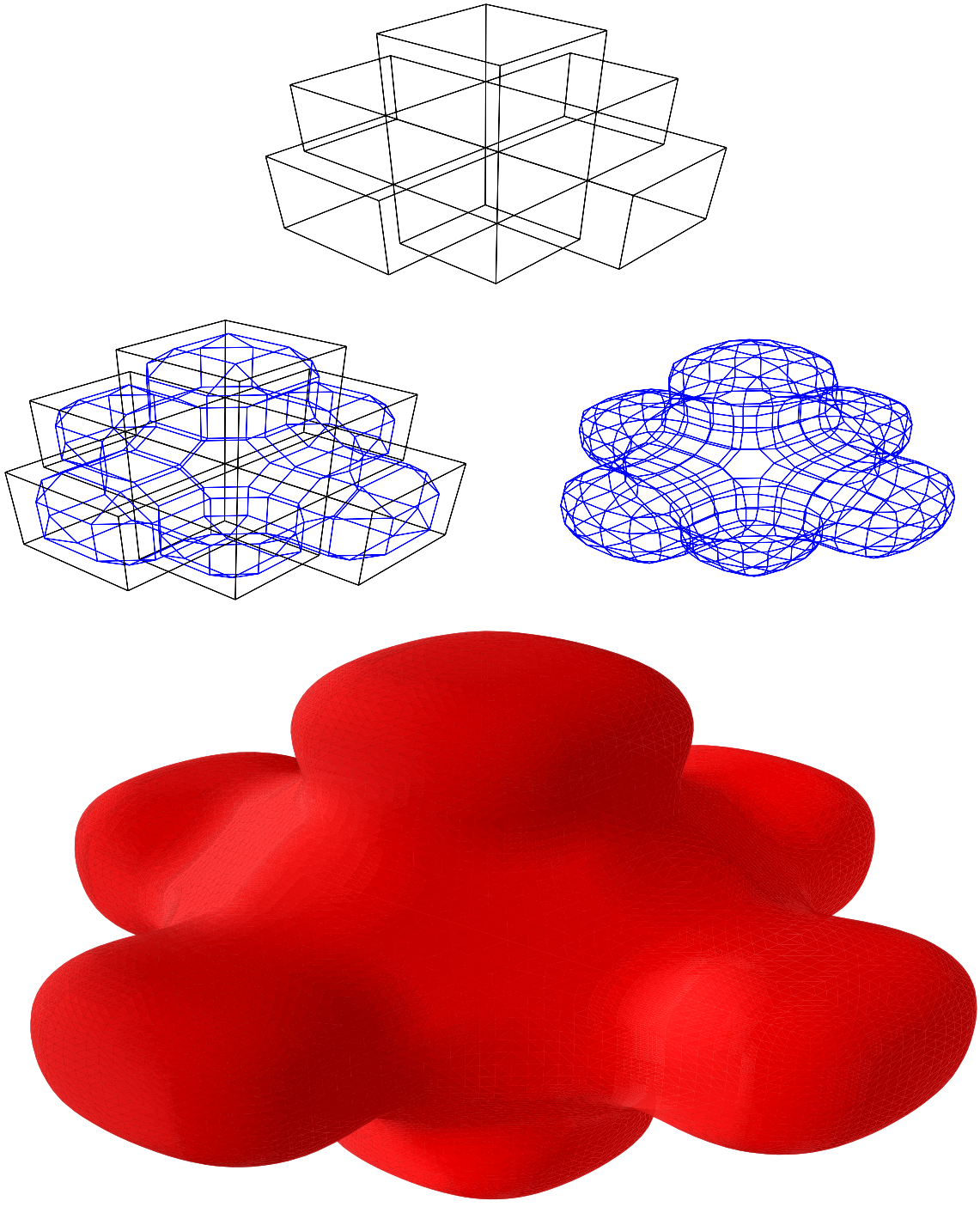


Abbildung 1.23: Doo–Sabin: Kreuz

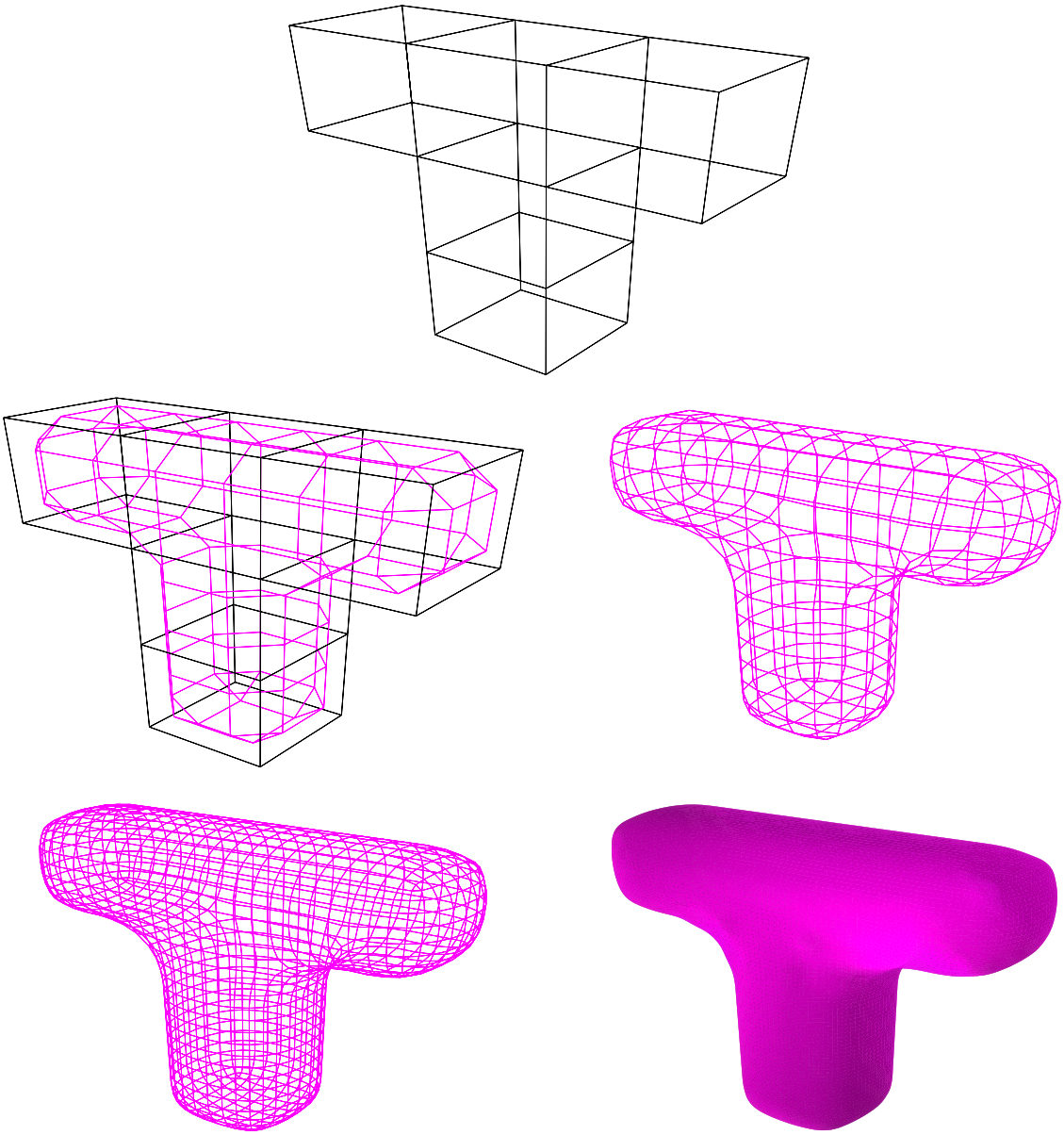


Abbildung 1.24: Doo–Sabin: T

Kapitel 2

Funktionen auf Flächen

In diesem Kapitel interessieren uns Funktionen auf den im vorherigen Kapitel vorgestellten, biquadratischen G–Spline–Flächen. Wir wollen diese Funktionen auch durch biquadratische Splines darstellen. Dazu betrachten wir zunächst zwei geometrisch glatt zusammengesetzte, biquadratische Splineflächen, auf denen jeweils eine Funktion gegeben ist, und suchen eine Bedingung, um diese beiden Funktionen stetig differenzierbar zusammenzusetzen. Hierzu stellen wir ein Ergebnis aus [Sey96] vor, welches eine mit der geometrischen Glattheit formal identische Bedingung liefert. Damit können wir Funktionen auf G–Spline–Flächen darstellen, indem wir zu jedem Kontrollpunkt der Fläche einen Funktionskontrollpunkt angeben. Die so erzeugten Funktionen lassen sich durch Farbwerte auf der Fläche, aber auch durch Stacheln, Netze, etc. über der Fläche darstellen. Mittels des bivariaten Romberg–Algorithmus' können wir dann verschiedene Oberflächenintegrale numerisch berechnen. Wir stellen auch einen Algorithmus zur Darstellung der Funktionen über Iso- bzw. Niveaulinien vor, mit dem man auch Reflektionslinien zeichnen kann. Zum Schluß untersuchen wir noch die Krümmung einer G–Spline–Fläche als spezielle Funktion.

2.1 Zusammensetzen von Funktionen auf Flächenstücken

Wie in Abschnitt 1.6 wollen wir zunächst zwei Bézierflächen p und q geometrisch glatt zusammensetzen. Die beiden Flächenstücke seien als Bézierflächenstücke mit den Kontrollpunkten $P_{j,k}$ und $Q_{j,k}$ gegeben,

$$p: \quad [0,1]^2 \mapsto \mathbb{R}^3, \quad p(u,v) = \sum_{j=0}^{\alpha} \sum_{k=0}^{\beta} P_{j,k} B_j^{\alpha}(u) B_k^{\beta}(v), \tag{2.1}$$

$$q: \quad [0,1]^2 \mapsto \mathbb{R}^3, \quad q(u,v) = \sum_{j=0}^{\alpha} \sum_{k=0}^{\beta} Q_{j,k} B_j^{\alpha}(u) B_k^{\beta}(v). \tag{2.2}$$

Wir nehmen an, daß p und q eine gemeinsame Randkurve $c : [0,1] \to \mathbb{R}^3$ besitzen. Diese sei o.B.d.A. gegeben durch

$$c(u) = p(u,0) = q(u,0) \quad \text{für } u \in [0,1]. \tag{2.3}$$

Weiter seien auf den beiden Bézierflächen $P := p([0,1]^2)$ und $Q := q([0,1]^2)$ zwei stetige differenzierbare Abbildungen

$$f_1 : P \to \mathbb{R}^d, \tag{2.4}$$

$$f_2 : Q \to \mathbb{R}^d \tag{2.5}$$

gegeben mit $d \in \mathbb{N}$. Wir folgen nun dem Ansatz aus [Sey96], um diese beiden Funktionen stetig differenzierbar zusammenzusetzen. Abbildung 2.1 zeigt eine schematische Darstellung dieser Situation.

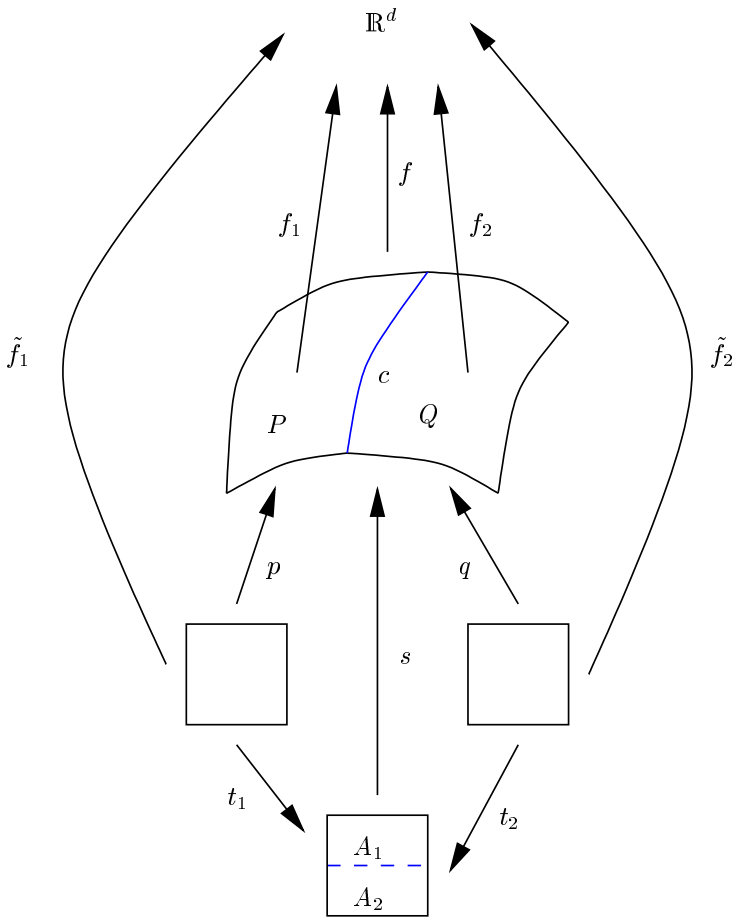


Abbildung 2.1: Zusammensetzen von Funktionen auf Flächenstücken

Wenn (1.34) für p und q erfüllt ist, erhalten wir eine längs c geometrisch glatt zusammengesetzte Fläche $S = P \cup Q$. Unter bestimmten Bedingungen an f_1 und f_2 erzeugen diese beiden Funktionen eine stetig differenzierbare Funktion $f : S \to \mathbb{R}^d$ mit

$$f(x) = \begin{cases} f_1(x) & \text{für } x \in P, \\ f_2(x) & \text{für } x \in Q. \end{cases} \tag{2.6}$$

Da P und Q die gemeinsame Randkurve c besitzen, muß

$$f_1(c(u)) = f_2(c(u)) \text{ für } u \in [0,1] \tag{2.7}$$

gelten, damit f wohldefiniert ist. Aus der Stetigkeit von f_1 und f_2 und aus (2.7) folgt, daß f längs c stetig ist.

Wir suchen nun noch eine Bedingung für die Differenzierbarkeit von f längs c. Dazu benötigen wir eine Parametrisierung $\tilde{f}$ von f, die differenzierbar ist. Zunächst werden wir eine stetig differenzierbare Parametrisierung $s : [0,1]^2 \to \mathbb{R}^3$ für S angeben. Diese erhalten wir durch das Kombinieren von p und q. Wir wählen dazu zwei stetig differenzierbare Transformationen

$$t_i : [0,1]^2 \to A_i \subset [0,1]^2, \quad i = 1,2 \tag{2.8}$$

mit $A_1 \cap A_2 = t_1([0,1],0) = t_2([0,1],0)$. Die Parametrisierung sei dann über

$$s(u,v) := \begin{cases} p(t_1^{-1}(u,v)) & \text{falls } (u,v) \in A_1, \\ q(t_2^{-1}(u,v)) & \text{falls } (u,v) \in A_2 \end{cases} \tag{2.9}$$

gegeben. Nach (2.3) ist dies wohldefiniert.

Um die Transformationen explizit zu bestimmen, wählen wir $t_1(u,v) = \left[u, \frac{1+v}{2}\right]^T$. Daraus folgt

$$t_2(u,0) = t_1(u,0) = \begin{bmatrix} u \\ \frac{1}{2} \end{bmatrix}. \tag{2.10}$$

Aus der stetigen Differenzierbarkeit von s folgt

$$\nabla\left(p\left(t_1^{-1}\left(u,\tfrac{1}{2}\right)\right)\right) = \nabla\left(q\left(t_2^{-1}\left(u,\tfrac{1}{2}\right)\right)\right), \tag{2.11}$$

$$\nabla p(u,0)\left(Dt_1\left(u,0\right)\right)^{-1} = \nabla q(u,0)\left(Dt_2\left(u,0\right)\right)^{-1}, \tag{2.12}$$

$$\begin{bmatrix} p_u(u,0) & 2p_v(u,0) \end{bmatrix} = \begin{bmatrix} q_u(u,0) & q_v(u,0) \end{bmatrix}\left(Dt_2\left(u,0\right)\right)^{-1}. \tag{2.13}$$

Aus (2.10) folgt

$$Dt_2\left(u,0\right) = \begin{bmatrix} 1 & \nu\left(u,0\right) \\ 0 & \mu\left(u,0\right) \end{bmatrix} \tag{2.14}$$

mit den Funktionen ν und μ für die partiellen Ableitungen von t_2 nach v. Damit gilt

$$\left(Dt_2\left(u,0\right)\right)^{-1} = \frac{1}{\mu\left(u,0\right)} \begin{bmatrix} \mu\left(u,0\right) & -\nu\left(u,0\right) \\ 0 & 1 \end{bmatrix}. \tag{2.15}$$

Dies in (2.13) eingesetzt ergibt

$$\mu\left(u,0\right)\begin{bmatrix} p_u(u,0) & 2p_v(u,0) \end{bmatrix} = \begin{bmatrix} \mu\left(u,0\right)q_u(u,0) & -\nu\left(u,0\right)q_u(u,0) + q_v(u,0) \end{bmatrix}. \tag{2.16}$$

Hieraus erhalten wir die Gleichungen

$$p_u(u,0) = q_u(u,0), \tag{2.17}$$

$$2\mu\left(u,0\right)p_v(u,0) = q_v(u,0) - \nu\left(u,0\right)q_u(u,0). \tag{2.18}$$

Aus (2.3) folgt sofort, daß (2.17) erfüllt ist. Es bleibt also zunächst die Gleichung (2.18). Setzt man (1.34) ein, ergibt dies

$$2\mu\left(u,0\right)p_v(u,0) = -\Phi(u)p_u(u,0) - \Psi(u)p_v(u,0) - \nu\left(u,0\right)p_u(u,0), \tag{2.19}$$

$$\left(2\mu\left(u,0\right) + \Psi(u)\right)p_v(u,0) = \left(-\Phi(u) - \nu\left(u,0\right)\right)p_u(u,0). \tag{2.20}$$

Hieraus erhält man die Bedingungen

$$\nu\left(u,0\right) = -\Phi(u), \tag{2.21}$$

$$\mu\left(u,0\right) = -\tfrac{1}{2}\Psi(u). \tag{2.22}$$

Dies bedeutet, daß

$$Dt_2\left(u,0\right) = \begin{bmatrix} 1 & -\Phi(u) \\ 0 & -\frac{1}{2}\Psi(u) \end{bmatrix}. \tag{2.23}$$

Diese Bedingung wird z.B. erfüllt von

$$t_2(u,v) = \begin{bmatrix} u - v\Phi(u) \\ -\frac{v}{2}\Psi(u) \end{bmatrix}. \tag{2.24}$$

Nun können wir eine Bedingung für die Differenzierbarkeit von $\tilde{f}$ herleiten. Dazu seien $\tilde{f}_i : [0,1]^2 \to \mathbb{R}^d$ für $i = 1, 2$ mit $\tilde{f}_1 = f_1 \circ p$ und $\tilde{f}_2 = f_2 \circ q$ definiert. Es sei weiter $\tilde{f} : [0,1]^2 \to \mathbb{R}^d$ gegeben durch

$$\tilde{f}(u,v) = \begin{cases} \tilde{f}_1\left(t_1^{-1}(u,v)\right) & \text{falls } (u,v) \in A_1, \\ \tilde{f}_2\left(t_2^{-1}(u,v)\right) & \text{falls } (u,v) \in A_2. \end{cases} \tag{2.25}$$

Aus der Stetigkeit von f folgt

$$\tilde{f}_1\left(t_1^{-1}\left(u, \tfrac{1}{2}\right)\right) = \tilde{f}_2\left(t_2^{-1}\left(u, \tfrac{1}{2}\right)\right), \tag{2.26}$$

$$\tilde{f}_1(u,0) = \tilde{f}_2(u,0) \tag{2.27}$$

für $u \in [0,1]$. Weiter soll f stetig differenzierbar sein, d.h. es muß gelten

$$Df_1(x) = Df_2(x) \text{ für } x \in c([0,1]), \tag{2.28}$$

$$D\tilde{f}_1\left(t_1^{-1}\left(u, \tfrac{1}{2}\right)\right) = D\tilde{f}_2\left(t_2^{-1}\left(u, \tfrac{1}{2}\right)\right), \tag{2.29}$$

$$\begin{bmatrix} \tilde{f_1}_{,u}(u,0) & 2\tilde{f_1}_{,v}(u,0) \end{bmatrix} = \begin{bmatrix} \tilde{f_2}_{,u}(u,0) & \tilde{f_2}_{,v}(u,0) \end{bmatrix} \left(Dt_2(u,0)\right)^{-1}. \tag{2.30}$$

Mit (2.23) folgt

$$\begin{bmatrix} \tilde{f_1}_{,u}(u,0) & 2\tilde{f_1}_{,v}(u,0) \end{bmatrix} = \begin{bmatrix} \tilde{f_2}_{,u}(u,0) & \tilde{f_2}_{,v}(u,0) \end{bmatrix} \frac{-2}{\Psi(u)} \begin{bmatrix} -\frac{1}{2}\Psi(u) & \Phi(u) \\ 0 & 1 \end{bmatrix}, \tag{2.31}$$

$$\begin{bmatrix} \tilde{f_1}_{,u}(u,0) & 2\tilde{f_1}_{,v}(u,0) \end{bmatrix} = \frac{-2}{\Psi(u)} \begin{bmatrix} -\frac{1}{2}\Psi(u)\tilde{f_2}_{,u}(u,0) & \Phi(u)\tilde{f_2}_{,u}(u,0) + \tilde{f_2}_{,v}(u,0) \end{bmatrix}. \tag{2.32}$$

Dies liefert die Gleichungen

$$\tilde{f_1}_{,u}(u,0) = \tilde{f_2}_{,u}(u,0), \tag{2.33}$$

$$-\Psi(u)\tilde{f_1}_{,v}(u,0) = \Phi(u)\tilde{f_2}_{,u}(u,0) + \tilde{f_2}_{,v}(u,0). \tag{2.34}$$

(2.33) ist wegen (2.27) erfüllt und aus (2.34) erhält man deshalb

$$\Phi\tilde{f_1}_{,u}(\cdot,0) + \Psi\tilde{f_1}_{,v}(\cdot,0) + \tilde{f_2}_{,v}(\cdot,0) \equiv 0. \tag{2.35}$$

Diese Bedingung stimmt formal mit der Bedingung (1.34) für die geometrische Glattheit von Flächen überein. Dabei ist zu beachten, daß Φ und Ψ dieselben Funktionen sind, die beim Zusammensetzen der Flächenstücke verwendet werden.

2.2 Darstellung von Funktionen durch Splines

Die formale Übereinstimmung von (2.35) und (1.34) legt es nahe, Funktionen auf einer Splinefläche wieder durch Splines darzustellen. Sei die biquadratische Splinefläche p durch (2.1) mit $\alpha = \beta = 2$ gegeben. Eine Funktion $f_1 : P \to \mathbb{R}^d$ auf dieser Fläche beschreiben wir als biquadratischen Spline in der Form

$$\tilde{f}_1 : [0,1]^2 \to \mathbb{R}^d, \quad \tilde{f}_1(u,v) = \sum_{j=0}^{2}\sum_{k=0}^{2} F_{j,k} B_j^2(u) B_k^2(v). \tag{2.36}$$

Dabei sind die $F_{j,k} \in \mathbb{R}^d$ $(j,k = 0,1,2)$ die Kontrollpunkte für die Funktion. Für einen Punkt $x = p(u_0, v_0)$ mit $u_0, v_0 \in [0,1]$ des Flächenstückes P gilt $f_1(x) = \tilde{f}_1(u_0, v_0)$, bzw. es gilt $\tilde{f}_1 = f_1 \circ p$. Man kann die Kontrollpunkte $F_{j,k}$ von $\tilde{f}_1$ auch so wählen, daß sie eine vorgegebene Funktion f approximieren.

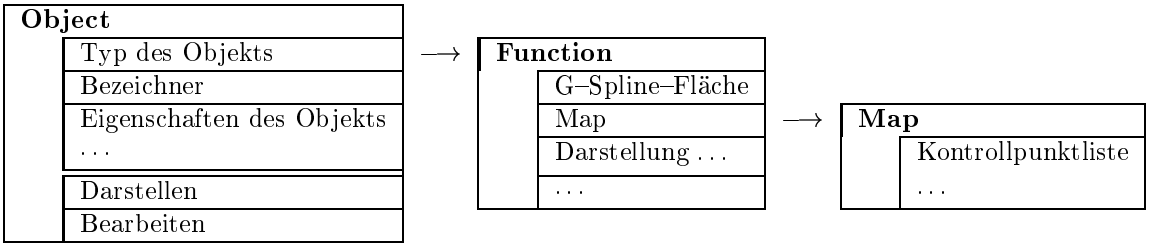


Abbildung 2.2: `Function` und `Map` Klassen

Sei analog eine Funktion $f_2 : Q \to \mathbb{R}^d$ auf dem Flächenstück Q durch

$$\tilde{f}_2 : [0,1]^2 \to \mathbb{R}^d, \quad \tilde{f}_2(u,v) = \sum_{j=0}^{2} \sum_{k=0}^{2} G_{j,k} B_j^2(u) B_k^2(v) \tag{2.37}$$

mit $G_{j,k} \in \mathbb{R}^d$ $(j,k = 0,1,2)$ gegeben. Um aus f_1 und f_2 eine stetig differenzierbare Funktion f zu erzeugen, müssen (2.7) und (2.35) erfüllt sein. Wir erhalten dabei die gleichen Bedingungen, die wir schon in Abschnitt 1.6 für glatte Flächen betrachtet haben. Allerdings sind Φ und Ψ bereits durch das Zusammensetzen der Flächenstücke bestimmt.

Ist die Fläche durch ein semi–reguläres Kontrollnetz gegeben, beschreiben wir eine Funktion auf dieser Fläche, indem wir zu jedem Kontrollpunkt der Fläche einen Kontrollpunkt für die Funktion angeben. Diese Form der Darstellung einer Funktion ist z.B. zur Darstellung von Meßwerten auf glatten Oberflächen geeignet.

Sowohl die Fläche als auch die Funktion werden dann durch biquadratische G–Splines dargestellt. Neben der Umwandlung des semi–regulären Kontrollnetzes in Bézierkontrollnetze durch den Algorithmus 1.2 muß auch das Kontrollnetz für die Funktion in Bézierkontrollnetze umgewandelt werden. Nachdem Φ und Ψ bereits durch die Fläche bestimmt werden, können hierfür die gleichen Formeln wie für die Flächenkontrollpunkte verwendet werden. Dazu müssen wir den Algorithmus 1.2 erweitern. Zusätzlich muß zu jedem Flächenkontrollpunkt der Funktionskontrollpunkt gefunden werden und das Bézierkontrollnetz für die Funktion analog zu dem für die Fläche berechnet werden. Ein Funktionsobjekt stellen wir dabei durch eine von `Object` abgeleitete Klasse `Function` dar. Sie enthält einen Zeiger auf die zugehörige G–Spline–Fläche und auf die `Map` Klasse. Die `Map` Klasse ist eine Liste, die jedem Kontrollpunkt der Fläche einen Kontrollpunkt der Funktion zuweist. Die Struktur ist grob in Abbildung 2.2 dargestellt. Details zur Implementierung des Algorithmus und der Klassen stehen in Kapitel 4 und im Anhang A. Die Struktur des Algorithmus' 1.2 ändert sich nicht.

Wird auf das Flächenkontrollnetz der Doo–Sabin–Algorithmus 1.3 angewendet, muß auch das Funktionskontrollnetz mit bearbeitet werden. Wie bei der Umwandlung in Bézierkontrollnetze müssen hier die gleichen Formeln auf die Kontrollpunkte der Funktion angewandt werden. Dazu wird der Algorithmus 1.3 so erweitert, daß er neben den Flächenkontrollpunkten auch gleich die Kontrollpunkte aller auf dieser Fläche definierten Funktionen in ein neues Kontrollnetz umwandelt.

Nachdem das Kontrollnetz der Funktion in Bézierkontrollnetze umgewandelt worden ist, haben wir mehrere Möglichkeiten, die Funktion darzustellen. Es bietet sich zunächst an, den Betrag des Funktionswertes in eine Farbe umzuwandeln und die Fläche selbst entsprechend einzufärben. Zu einem gegebenen Punkt x auf der Fläche berechnen wir den zugehörigen Farbwert $w(x)$ aus der über das Bézierkontrollnetz berechneten Funktion. Für skalarwertige Funktionen erhalten wir den Farbwert aus

$$w(x) = f(x) f_{\text{scale}} c_{\text{scale}} + c_{\text{delta}}. \tag{2.38}$$

Für vektorwertige Funktionen verwenden wir die euklidische Norm:

$$w(x) = \|f(x)\|_2 f_{\text{scale}} c_{\text{scale}} + c_{\text{delta}}. \tag{2.39}$$

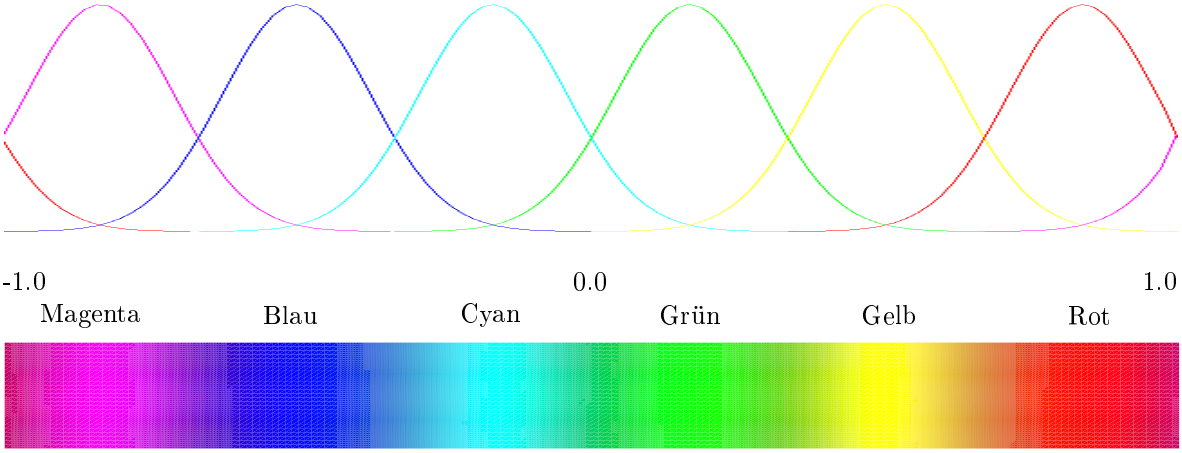


Abbildung 2.3: Farbschema zur Darstellung von Funktionen durch Farbwerte

f_{scale} ist dabei ein allgemeiner Skalierungsfaktor für die Funktion. c_{scale} und c_{delta} werden zur Skalierung und Translation der Farbwerte verwendet. Wie in Abbildung 2.3 gezeigt, bilden wir dann das Intervall $(-1, 1]$ auf die Farben ab. Funktionswerte in den Intervallen $(j, j+2]$ für ungerade $j \in \mathbb{Z}$ werden vorher auf das Intervall $(-1, 1]$ verschoben. Die Farbwerte werden durch die Funktionen

$$C_i(w) = e^{-32\left(\frac{k_i}{6} - w\right)^2} \tag{2.40}$$

festgelegt. Die einzelnen Parameter für die Farben sind dabei durch folgende Tabelle gegeben:

$$\begin{array}{lll} \text{Rot links:} & i = 1 & k_i = -7 \\ \text{Magenta links:} & i = 2 & k_i = -5 \\ \text{Blau:} & i = 3 & k_i = -3 \\ \text{Cyan:} & i = 4 & k_i = -1 \\ \text{Grün:} & i = 5 & k_i = 1 \\ \text{Gelb:} & i = 6 & k_i = 3 \\ \text{Rot rechts:} & i = 7 & k_i = 5 \\ \text{Magenta rechts:} & i = 8 & k_i = 7 \end{array} \tag{2.41}$$

Hierbei ist zu beachten, daß wir für Rot und Magenta eine Funktion am linken und am rechten Rand des Intervalls benötigen um einen kontinuierlichen Farbverlauf zu sichern. Der RGB–Farbwert wird schließlich durch

$$\begin{aligned} \begin{bmatrix} R \\ G \\ B \end{bmatrix} = & \begin{bmatrix} r \\ 0 \\ 0 \end{bmatrix} C_1(w) + \begin{bmatrix} r \\ 0 \\ b \end{bmatrix} C_2(w) + \begin{bmatrix} 0 \\ 0 \\ b \end{bmatrix} C_3(w) + \begin{bmatrix} 0 \\ g \\ b \end{bmatrix} C_4(w) + \\ & \begin{bmatrix} 0 \\ g \\ 0 \end{bmatrix} C_5(w) + \begin{bmatrix} r \\ g \\ 0 \end{bmatrix} C_6(w) + \begin{bmatrix} r \\ 0 \\ 0 \end{bmatrix} C_7(w) + \begin{bmatrix} r \\ 0 \\ b \end{bmatrix} C_8(w) \end{aligned} \tag{2.42}$$

berechnet. r, g, b geben dabei die maximalen Werte für den roten, grünen und blauen Farbkanal an. Normalerweise ist dies 1.0 für alle drei Farben. Da die C_i nur jeweils maximal drei Farbbereiche wirklich beeinflussen, müssen diese bei der Implementierung auch nur jeweils für drei benachbarte Bereiche berechnet werden (siehe Abbildung 2.3). Abbildung 2.5 zeigt eine Fläche, deren Kontrollpunkte über den Whitneyschen Regenschirm mit der Parametrisierung

$$(u, v) \mapsto \begin{bmatrix} uv \\ u \\ v^2 \end{bmatrix}, \quad u = -2 : 0.4 : 2,\ v = -2 : 0.4 : 2 \tag{2.43}$$

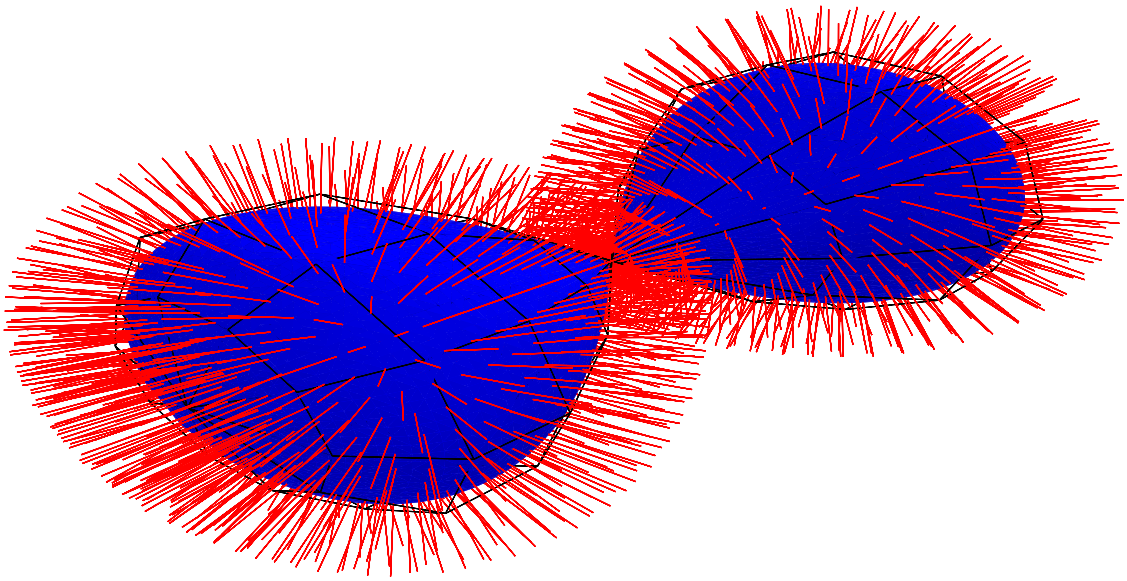


Abbildung 2.4: Einheitsnormalenfeld der Acht–Fläche

bestimmt wurden. Die Kontrollpunkte der über Farbwerte dargestellten Funktion wurden durch

$$(x, y, z) \mapsto \left(x^2 + y^2\right)^{0.2} \tag{2.44}$$

aus den Kontrollpunkten der Fläche berechnet.

Alternativ können wir die Funktionswerte auch durch Stacheln auf der Fläche darstellen. Dazu zeichnen wir eine Strecke vom einem Punkt x auf der Fläche längs des Einheitsnormalenvektors $n(x)$ mit der Länge $f(x)f_{\text{scale}}$ für skalarwertige bzw. $\|f(x)\|_2 f_{\text{scale}}$ für vektorwertige Funktionen. Aus den Stacheln können wir natürlich auch eine "Funktionsfläche" über der Fläche erzeugen. Diese kann dann als Gitternetz dargestellt werden oder durch die vorher beschriebene Methode eingefärbt werden. Eine eindimensionale Funktion $f : S \to \mathbb{R}^3$ kann man auch als Vektoren auf der Fläche darstellen.

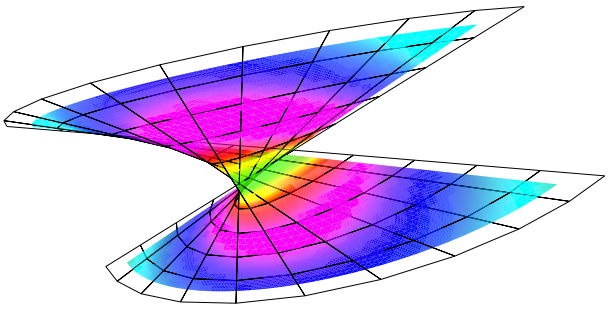


Abbildung 2.5: $(x, y, z) \mapsto \left(x^2 + y^2\right)^{0.2}$ auf dem Whitneyschen Regenschirm

Die verschiedenen Darstellungsarten lassen sich auch miteinander verknüpfen und sie können mittels eines einzigen Alogrithmus' berechnet werden. Zu jedem biquadratischen Bézierkontrollnetz der Funktion suchen wir zunächst das zugehörige Bézierkontrollnetz der Fläche. Wird schon beim Erzeugen der Bézierkontrollnetze hierauf Rücksicht genommen, können die Kontrollnetze in der gleichen Reihenfolge in jeweils einer Liste abgelegt werden, so daß wir nur noch beide Listen gleichzeitig durchlaufen müssen. Wir legen dann ein Gitter über den Parameterbereich und je nach ausgewählter Darstellungsart berechnen wir den Punkt auf der Fläche, die Flächennormale und den Funktionswert an den Gitterknoten. Die Stacheln, Vektoren und Gitternetze können direkt als Linien dargestellt werden. Für die farbigen Flächen unterteilen wir jedes Quadrat in zwei Dreiecke. Über die Farbwerte an den Ecken lassen sich dann einfach die Farbverläufe auf den Dreiecken bestimmen.

Mit Hilfe der Stacheln kann man z.B. das Einheitsnormalenfeld einer Fläche darstellen. Dazu verwenden wir einfach eine konstante Funktion auf dieser Fläche. Abbildung 2.4 zeigt das Einheitsnormalen-

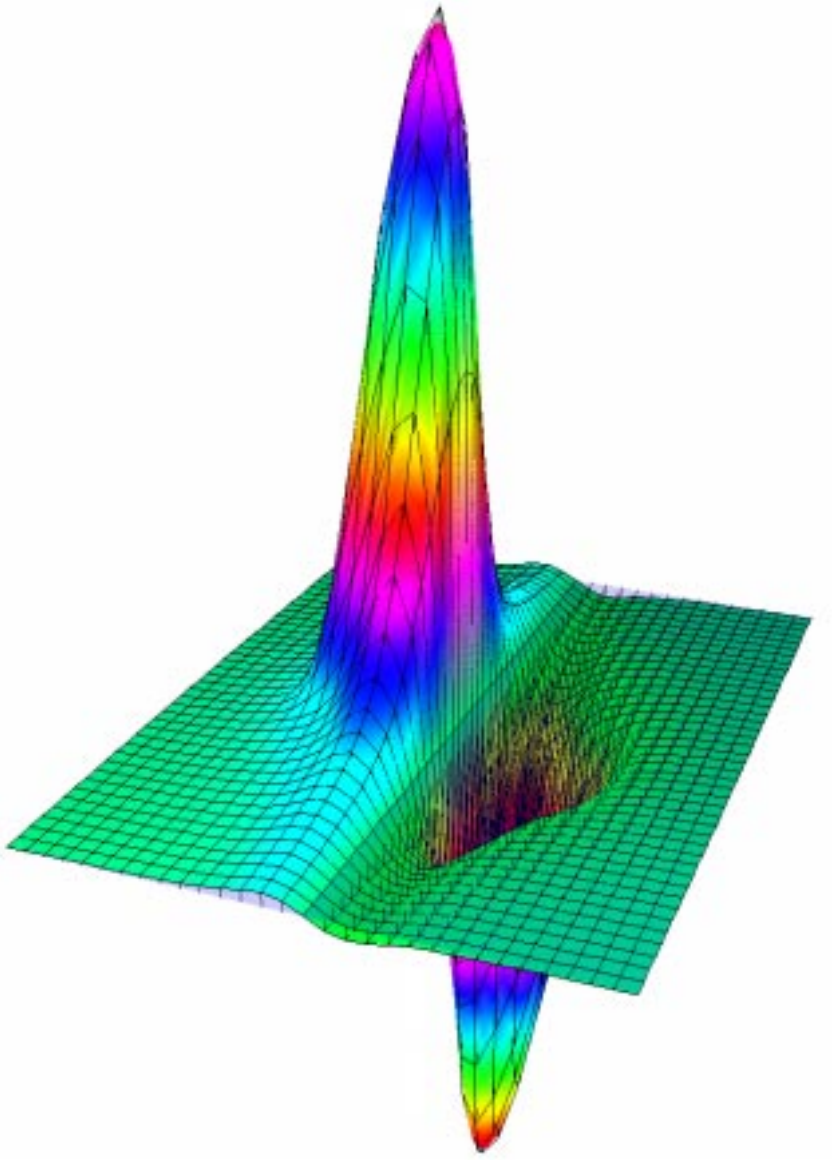


Abbildung 2.6: Dipolpotential

feld einer Acht–Fläche. Die Kontrollpunkte der Fläche wurden durch

$$(u,v) \mapsto \begin{bmatrix} \cos(u)\cos(v)\sin(v) \\ \sin(u)\cos(v)\sin(v) \\ \sin(v) \end{bmatrix} \quad \text{für } u = 0 : \frac{\pi}{5} : 2\pi,\ v = -\frac{\pi}{2} : \frac{\pi}{10} : \frac{\pi}{2} \tag{2.45}$$

berechnet. Die Punkte an der Nahtstelle für $u = 0$ und $u = 2\pi$ entlang der z–Achse wurden dabei miteinander identifiziert. Auch wurden die Punkte an den beiden "Enden" der Acht miteinander identifiziert, so daß dort jeweils eine Irregularität entsteht. Durch zweimaliges Anwenden des Doo–Sabin–Algorithmus' erhalten wir dann ein G–Spline–Kontrollnetz. Für den singulären Punkt bei $(0,0,0)$ haben wir allerdings die durch obige Parametrisierung erzeugten Kontrollpunkte für $v = 0$ beibehalten, da sonst der Doo–Sabin–Algorithmus die Fläche an dieser Stelle trennen würde.

In Abbildung 2.6 stellen wir näherungsweise das Potential eines statischen Dipols in der x–y–Ebene dar. Die Fläche ist ist dabei der durch die Kontrollpunkte $\left[u,v,0\right]^T$ mit $u = -2.2 : 0.4 : 2.2$, $v = -3 : 0.4 : 3$

bestimmte Teil der x–y–Ebene. Die Kontrollpunkte der Potentialfunktion wurden durch die Funktion

$$(x, y) \mapsto \frac{1}{x(x^2 + y^2)} \quad \text{für } x = -2.2 : 0.4 : 2.2,\ y = -3 : 0.4 : 3 \tag{2.46}$$

erzeugt. Die Funktion wurde durch Stacheln, ein Gitternetz und eine eingefärbte Fläche über der transparenten Ebene dargestellt.

2.3 Integration

2.3.1 Oberflächenintegrale

Wir wollen nun die über G–Spline–Kontrollpunkte gegebene Funktion $f : S \to \mathbb{R}$ auf der G–Spline–Fläche S integrieren. Dazu betrachten wir zunächst die allgemeine Definition.

Definition 2.1 (Oberflächenintegral von Skalarfeldern). *Es sei $S : \mathbb{R}^2 \supset K \to \mathbb{R}^3$ eine Fläche und f ein stetiges Skalarfeld auf $S(K)$. Dann ist durch*

$$\int_S f d\sigma := \int_K f(S(u, v)) \left\| \frac{\partial S}{\partial u} \times \frac{\partial S}{\partial v} \right\| d(u, v) \tag{2.47}$$

das Oberflächenintegral von f über S definiert (vgl. [Heu90]).

Wir nehmen an, daß für f und S die biquadratischen Bézierkontrollnetze bereits erzeugt wurden. Der Parameterbereich K von S sei ein aus kompakten Intervallen im $\mathbb{R}^2$ zusammengesetztes Gebiet. Damit können wir K in kompakte, zwei–dimensionale Intervalle K_i der Form $[a_i, a_i + 1] \times [b_i, b_i + 1]$ mit $a_i, b_i \in \mathbb{R}$ und $i \in I$ zerlegen. Jedes K_i ist der Parameterbereich eines Bézierflächenstücks. Dann gilt

$$\begin{aligned} \int_S f d\sigma &= \sum_{i \in I} \int_{S(K_i)} f d\sigma \\ &= \sum_{i \in I} \int_{K_i} f(S(a, b)) \left\| \frac{\partial S}{\partial a} \times \frac{\partial S}{\partial b} \right\| d(a, b). \end{aligned} \tag{2.48}$$

Wir betrachten jetzt speziell einen Parameterbereich K_i. Die Funktion f sei für K_i durch ein f_i in der Form von (2.36) mit $F^i_{j,k} \in \mathbb{R}$ für $j, k = 0 : 2$ gegeben,

$$f_i : [0, 1]^2 \to \mathbb{R}, \quad f_i(u, v) = \sum_{j=0}^{2} \sum_{k=0}^{2} F^i_{j,k} B^2_j(u) B^2_k(v). \tag{2.49}$$

S_i sei durch

$$S_i : [0, 1]^2 \to \mathbb{R}, \quad S_i(u, v) = \sum_{j=0}^{2} \sum_{k=0}^{2} P^i_{j,k} B^2_j(u) B^2_k(v) \tag{2.50}$$

mit $P^i_{j,k} \in \mathbb{R}^3$ gegeben. Beide Funktionen lassen sich durch eine einfache Translation $t_i : K_i \to [0, 1]^2, (a, b) \mapsto (a - a_i, b - b_i)$ von $[0, 1]$ auf K_i verschieben, so daß $S(a, b) = S_i(t_i(a, b))$ und $f(S(a, b)) =$

$f_i(t_i(a,b))$ für $(a,b) \in K_i$ gilt. Da $\det(Dt_i) = 1$, erhalten wir für das Integral über K_i

$$\begin{aligned}\iint_{K_i} f(S(a,b)) \left\| \frac{\partial S}{\partial a} \times \frac{\partial S}{\partial b} \right\| d(a,b) = \iint_{[0,1]^2} f_i(u,v) \left\| \frac{\partial S_i}{\partial u} \times \frac{\partial S_i}{\partial v} \right\| d(u,v) = \\ \iint_{[0,1]^2} \left(\sum_{j=0}^{2} \sum_{k=0}^{2} \tilde{F}^i_{j,k} B^2_j(u) B^2_k(v) \right) \\ \left\| \left(\sum_{j=0}^{2} \sum_{k=0}^{2} P^i_{j,k} \frac{dB^2_j(u)}{du} B^2_k(v) \right) \times \left(\sum_{j=0}^{2} \sum_{k=0}^{2} P^i_{j,k} B^2_j(u) \frac{dB^2_k(v)}{dv} \right) \right\| d(u,v).\end{aligned} \tag{2.51}$$

Für das Integral (2.48) müssen wir also für jedes biquadratische Bézierflächenstück der Fläche bzw. der Funktion das Integral über (2.51) berechnen.

Im mehrdimensionalen Fall können wir natürlich zunächst über die Norm integrieren. Für drei-dimensionale Vektorfelder wollen wir aber speziell ein Oberflächenintegral durch die folgende Definition einführen.

Definition 2.2 (Oberflächenintegral von Vektorfeldern). *Es sei* $S : \mathbb{R}^2 \supset K \to \mathbb{R}^3, (u,v) \mapsto \big[X(u,v), Y(u,v), Z(u,v)\big]^T$ *eine Fläche mit dem Parameterbereich* K *und*

$$f : S \to \mathbb{R}^3, \quad p \mapsto \begin{bmatrix} P(p) \\ Q(p) \\ R(p) \end{bmatrix} \tag{2.52}$$

ein stetiges drei-dimensionales Vektorfeld auf $S(K)$*. Dann definiert man*

$$\begin{aligned}\int_S P dy \wedge dz &:= \int_K P(S(u,v)) \frac{\partial(Y,Z)}{\partial(u,v)} \, d(u,v), \\ \int_S Q dz \wedge dx &:= \int_K Q(S(u,v)) \frac{\partial(Z,X)}{\partial(u,v)} \, d(u,v), \\ \int_S R dx \wedge dy &:= \int_K R(S(u,v)) \frac{\partial(X,Y)}{\partial(u,v)} \, d(u,v)\end{aligned} \tag{2.53}$$

und man nennt

$$\int_S P dy \wedge dz + Q dz \wedge dx + R dx \wedge dy := \int_S P dy \wedge dz + \int_S Q dz \wedge dx + \int_S R dx \wedge dy \tag{2.54}$$

das Oberflächenintegral von f *über* S*. (vgl. [Heu90])*

Dabei ist aber zu beachten, daß das Oberflächenintegral $\int_S P dy \wedge dz + Q dz \wedge dx + R dx \wedge dy$ auch von der Parameterdarstellung von S abhängt. Es ist aber invariant unter positivem Parameterwechsel und ändert nur das Vorzeichen bei negativem Parameterwechsel. Es gibt eine zweite Möglichkeit, dieses Oberflächenintegral darzustellen (siehe z.B. [Heu90]). Ist $n(u,v)$ der Einheitsnormalenvektor der Fläche S an $S(u,v)$, dann können wir das Oberflächenintegral auch als

$$\begin{aligned}\int_S P dy \wedge dz + Q dz \wedge dx + R dx \wedge dy &= \int_K \left\langle f(S(u,v)), \frac{\partial S}{\partial u} \times \frac{\partial S}{\partial v} \right\rangle d(u,v) \\ &= \int_K \langle f(S(u,v)), n(u,v) \rangle \left\| \frac{\partial S}{\partial u} \times \frac{\partial S}{\partial v} \right\| d(u,v) \\ &= \int_S \langle f, n \rangle \, d\sigma\end{aligned} \tag{2.55}$$

in der Form von (2.47) schreiben. Damit haben wir wieder ein skalares Oberflächenintegral, so daß wir für jedes Bézierflächenstück S_i mit $i \in I$ das Integral

$$\iint_{[0,1]^2} \langle f_i, n \rangle \, d(u,v) \tag{2.56}$$

berechnen können, um das Oberflächenintegral des Vektorfeldes f zu erhalten.

2.3.2 Bivariater Romberg–Algorithmus

Nachdem wir die beiden Formen der Oberflächenintegrale vorgestellt haben und diese auf eine gemeinsame Form eines bivariaten Integrals gebracht haben, werden wir eine Methode zur numerischen Berechnung solcher bivariaten Integrale vorstellen. Siehe hierzu auch [Höl98] und [KU98]. Zunächst führen wir dazu eine univariate Integrationsformel für

$$If := \int_a^b f(x)\,dx \tag{2.57}$$

ein. Zur numerischen Approximation dieses Integrals können wir f stückweise durch Polynome approximieren und die Integrale über diese Polynome aufsummieren. Dazu zerlegen wir $[a,b]$ durch die $k+1$ äquidistanten Stützstellen $x_0,\ldots,x_k$ in k Teilintervalle,

$$\begin{aligned} a = x_0 < x_1 < \cdots < x_{k-1} < x_k &= b, \\ x_j = a + j\frac{b-a}{k} \text{ für } j &= 0:k. \end{aligned} \tag{2.58}$$

Approximieren wir, wie in Abbildung 2.7 gezeigt, f auf jedem Teilintervall durch eine lineare Funktion, erhalten wir nach der Integration der linearen Funktionen die Trapezregel $T_k f$:

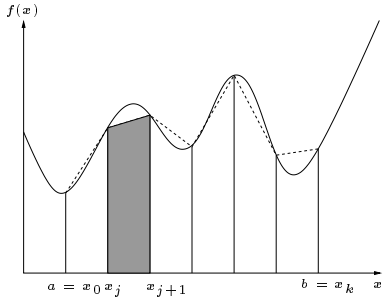


Abbildung 2.7: Trapezregel

$$If \approx T_k f := \frac{b-a}{k}\left(\frac{1}{2}f(x_0) + \sum_{j=1}^{k-1} f(x_j) + \frac{1}{2}f(x_k)\right). \tag{2.59}$$

Der Fehler der Trapezregel $T_k f$ zum exakten Wert läßt sich asymptotisch durch folgendes Theorem beschreiben.

Theorem 2.3 (Euler–Maclaurin–Summationsformel). *Sei $f \in C^{2m}([a,b])$. Dann gilt*

$$If - T_k f = \sum_{j=1}^{m-1} c_{2j}\left(f^{(2j-1)}(b) - f^{(2j-1)}(a)\right)h^{2j} + c_{2m} f^{(2m)}(t)(b-a)h^{2m} \tag{2.60}$$

für $t \in [a,b]$ und den von f unabhängigen Konstanten

$$c_{2j} = -\frac{p_{2j}(0)}{(2j)!}, \tag{2.61}$$

die über die Bernoulli Polynome p_j definiert werden:

$$\begin{aligned} p_0(x) &= 1, \\ p_j'(x) &= j p_{j-1}(x) \quad \textit{für } j \in \mathbb{N}. \end{aligned} \tag{2.62}$$

Beweis. Siehe [Höl98]. ∎

Nach (2.60) kann der Fehler der Trapezregel durch

$$If - T_k f = K_2 h^2 + K_4 h^4 + K_6 h^6 + K_8 h^8 + \cdots + K_{2m} h^{2m} \tag{2.63}$$

dargestellt werden. Dabei ist $m \in \mathbb{N}$ beliebig, solange $f \in C^{2m}([a,b])$. Die Konstanten K_{2j} hängen dabei nicht von der Schrittweite h, sondern nur von f selbst ab. Damit lassen sich einzelne Fehlerterme über die Richardson–Extrapolation eliminieren,

$$\left.\begin{aligned} T_i^0 &:= T_{2^i k} f, \\ T_i^{j+1} &:= \frac{4^{j+1} T_i^j - T_{i-1}^j}{4^{j+1}-1} \end{aligned}\right\} \quad \text{mit } j = 0:i \text{ für } i \in \mathbb{N}_0. \tag{2.64}$$

Mit der Trapezregel berechnen wir also zunächst T_i^0, wobei wir mit k Punkten anfangen und diese pro Schritt in i verdoppeln. Pro Schritt in j verschwindet in T_i^{j+1} jeweils ein Fehlerterm durch Kombination von T_i^j und T_{i-1}^j. Dieser Algorithmus wird auch als Romberg–Algorithmus bezeichnet. Er läßt sich auch durch das folgende Dreieckschema darstellen:

$$\begin{array}{ccccccc} T_1^0 & & & & & & \\ T_2^0 & T_1^1 & & & & & \\ T_3^0 & T_2^1 & T_1^2 & & & & \\ T_4^0 & T_3^1 & T_2^2 & T_1^3 & & & \\ T_5^0 & T_4^1 & T_3^2 & T_2^3 & T_1^4 & & \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{array}$$

Aus diesem Algorithmus werden wir ein numerisches Integrationsverfahren für bivariate Integrale

$$If := \iint_A f(x,y)\,d(x,y) \tag{2.65}$$

herleiten. Für unsere Anwendung genügt es anzunehmen, daß A ein zwei-dimensionales Intervall $[a,b]\times[c,d]$ in $\mathbb{R}^2$ ist. Wir unterteilen beide Intervalle $[a,b]$ und $[c,d]$ in jeweils k_x bzw. k_y Teilintervalle und wenden die Trapezregel auf beide Integrale getrennt an. Mit $h_x := \frac{b-a}{k_x}$ und $h_y := \frac{d-c}{k_y}$ ergibt dies die numerische Integrationsformel

$$\begin{aligned} If &\approx \int_a^b \underbrace{T_k f(x,\cdot)}_{=:F(x)}\,dx = \int_a^b h_y \left(\frac{1}{2}f(x,c) + \sum_{j=1}^{k_y-1} f(x,c+jh_y) + \frac{1}{2}f(x,d)\right) dx \approx T_k F \\ &= \frac{1}{2}h_x h_y \left(\frac{1}{2}f(a,c) + \sum_{j=1}^{k_y-1} f(a,c+jh_y) + \frac{1}{2}f(a,d)\right) + \\ &\quad \sum_{i=1}^{k_x-1} h_x h_y \left(\frac{1}{2}f(a+ih_x,c) + \sum_{j=1}^{k_y-1} f(a+ih_x,c+jh_y) + \frac{1}{2}f(a+ih_x,d)\right) + \\ &\quad \frac{1}{2}h_x h_y \left(\frac{1}{2}f(b,c) + \sum_{j=1}^{k_y-1} f(b,c+jh_y) + \frac{1}{2}f(b,d)\right). \end{aligned} \tag{2.66}$$

Damit erhalten wir eine bivariate Integrationsformel aus dem Tensorprodukt der univariaten Trapezregel. Wir legen dazu ein zwei–dimensionales Gitter mit den Schrittweiten h_x, h_y über das rechteckige Integrationsgebiet und multiplizieren die Funktionswerte an den Knoten mit bestimmten Faktoren. Dabei müssen wir zwischen Eckknoten, Randknoten und inneren Knoten unterscheiden. Die Faktoren aus der Formel (2.66) sind in Abbildung 2.8 dargestellt.

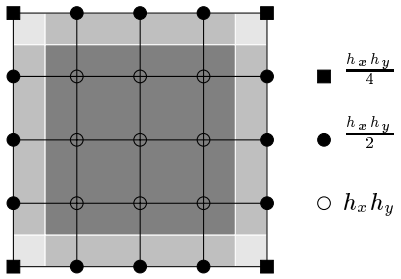


Abbildung 2.8: Bivariate Trapezregel

Den Fehler für (2.66) können wir durch zweimaliges Anwenden von Theorem 2.3 bestimmen. Danach gilt für $h = h_x = h_y$

$$\begin{aligned} If - \int_a^b F(x)\,dx = \sum_{j=1}^{m-1} c_{2j} \int_a^b \left(\frac{\partial^{(2j-1)} f}{\partial y^{2j-1}}(x,d) - \frac{\partial^{(2j-1)} f}{\partial y^{2j-1}}(x,c)\right) dx\, h^{2j} + \\ c_{2m} \int_a^b \frac{\partial^{(2m)} f}{\partial y^{2m}}(x,t_y)\,dx\,(d-c)h^{2m}, \end{aligned} \tag{2.67}$$

$$\int_a^b F(x)\,dx - T_k F = \sum_{j=1}^{m-1} c_{2j} \left(F^{(2j-1)}(b) - F^{(2j-1)}(a)\right) h^{2j} + c_{2m} F^{(2m)}(t_x)(b-a)h^{2m} \tag{2.68}$$

mit $t_x \in [a,b]$ und $t_y \in [c,d]$. Uns interessiert aber nur die allgemeine Form des Fehlers und dieser enthält wieder nur Terme mit h^{2j},

$$If - T_k F = If - \int_a^b F + \int_a^b F - T_k F = K_2 h^2 + K_4 h^4 + K_6 h^6 + \cdots + K_{2m} h^{2m}. \tag{2.69}$$

Dabei hängen die K_{2j} nur von f ab. Dies entspricht (2.63) und wir können damit den Romberg–Algorithmus (2.64) auch für die bivariate Trapezregel verwenden.

Nun können wir den allgemeinen Algorithmus zur Berechnung eines Integrals einer G–Spline–Funktion über einer G–Spline–Fläche der Form

$$\sum_{i \in I} \iint_{[0,1]^2} F_i \tag{2.70}$$

angeben. Wir gehen davon aus, daß die Fläche bereits über Bézierflächenstücke S_i für $i \in I$ gegeben ist. Die Funktion wird auf jedem Bézierflächenstück S_i durch F_i repräsentiert. F_i kann dabei wie oben beschrieben über ein Bézierkontrollnetz gegeben sein. Es kann aber auch nur von der Oberfläche selbst abhängen. Wir müssen nur jeweils die Trapezregel, bzw. die in der Trapezregel verwendete Funktion entsprechend wählen. Für die bisher vorgestellten Integrale geht dies aus den Formeln (2.51) und (2.56) hervor.

Algorithmus 2.1. `integrate`
Bivariater Romberg–Algorithmus

I. Setze das Ergebnis `value` der Integration auf 0.

II. Berechne das Integral für jedes Bézierkontrollnetz `bgs` der G–Spline–Fläche separat über den Romberg–Algorithmus:

   A. Initialisiere die Variablen für den Romberg–Algorithmus. Dabei ist `t` ein Array der Länge `max_iter`, welche die maximale Anzahl der Iterationen angibt. `n` gibt die Anzahl der Stützstellen in einer Gitterrichtung an und `i` wird als Iterationszähler auf 1 gesetzt.

   B. Berechne die Trapezregel für die Intervalllänge `1/(n-1)` pro Gitterrichtung (n^2 Stützstellen) und lege das Ergebnis in `t[0]` ab.

   C. Wiederhole Folgendes, solange die gewünschte Toleranz `prec` nicht erreicht wurde oder die maximale Anzahl der Iterationen erreicht wurde:

      1. Verdopple `n`.
      2. Berechne die Trapezregel für die Intervalllänge `1/(n-1)` pro Gitterrichtung (n^2 Stützstellen) und lege das Ergebnis in `t[i]` ab.
      3. Für `j = i:-1:1` berechne die Romberg–Extrapolation iterativ über `t[j-1] = (4^(j-i+1) * t[j] - t[j-1]) / (4^(j-i+1) - 1)`.
      4. Zur Bestimmung der Genauigkeit berechne den Betrag der Differenz zwischen dem besten Wert der Extrapolation in `t[0]` und dem besten Ergebnis der vorherigen Iteration.
      5. Erhöhe `i` um 1.

   D. Addiere das Ergebnis des Romberg–Algorithmus' zu `value`.

III. Das Ergebnis der Integration steht in `value`.

Es ist zunächst zu beachten, daß wir das Array `t` von rechts nach links füllen. Damit benötigen wir nur ein einziges, eindimensionales Array für die Extrapolation, in dem aus den alten Werten die neuen berechnet werden. Im ersten Schritt wird das Ergebnis der Trapezregel in ein noch nicht benutztes Element `t[i]` des Arrays abgelegt und die Extrapolationswerte von diesem Element aus bis `t[0]` eingetragen.

Die im Algorithmus verwendete Toleranz `prec` bezieht sich dabei auf das Integral über eine einzelne Bézierfläche. Durch das Aufsummieren der Ergebnisse verringert sich die Toleranz für das gesamte Integral. Bei M Bézierflächen ist sie nur noch $M * \texttt{prec}$.

Die verwendete Trapezregel entspricht natürlich der bivariaten Trapezregel (2.66), wobei im Algorithmus die Anzahl der Intervalle k durch die Anzahl der Stützstellen n ersetzt wurde. Es gilt $n = k + 1$. Je nach Integral kann f durch andere Funktionen ersetzt werden. So verwenden wir z.B. für das Oberflächenintegral von Vektorfeldern die Funktion $\langle f, n \rangle$. Spezielle Integrale stellen wir weiter unten vor.

Weiter ist auch zu beachten, daß wir nach dem ersten Auswerten der Trapezregel für die folgenden Auswertungen die Funktion nicht mehr an allen Stützpunkten berechnen müssen. Nachdem wir n jeweils verdoppeln, benötigen wir nur noch die Stützstellen $(i\frac{1}{n-1}, j\frac{1}{n-1})$, bei denen wenigstens eine der beiden Zahlen i und j ungerade ist. Für die restlichen Stützstellen können wir das Ergebnis der vorherigen Trapezregel mit $\frac{1}{4}$ multiplizieren. Die Punkte, an denen die Funktion noch ausgewertet werden muß, sind in Abbildung 2.9 gekennzeichnet. Von den n^2 Stützstellen wird die Funktion dann nur noch an $n^2 - \left(\frac{n}{2}\right)^2 = \frac{3}{4}n^2$ Stützstellen berechnet.

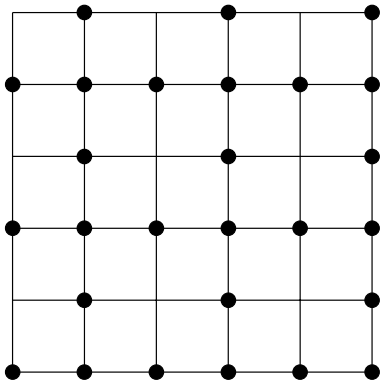


Abbildung 2.9: Funktionsauswertungen bei der bivariaten Trapezregel

Eine mögliche Verbesserung des Romberg–Algorithmus' wäre eine adaptive Anpassung an den Approximationsfehler. Wir berechnen dazu zunächst das Integral für ein zwei–dimensionales Integrationsintervall A und schätzen zusätzlich den Fehler des Ergebnisses ab. Ist dieser Fehler größer als eine vorgegebene Toleranz s, unterteilen wir das Intervall A in vier gleichgroße Teilintervalle und berechnen für diese Intervalle das Integral getrennt mit der Toleranz $\frac{s}{4}$. Dies wird rekursiv wiederholt, bis die Fehlerabschätzung kleiner als die Toleranz ist. Hierdurch wird die Anzahl der Stützstellen dem jeweiligen Approximationsfehler auf dem Teilintervall angepaßt. Nachdem wir aber schon das Integrationsgebiet am Anfang in die einzelnen quadratischen Bézierfläche unterteilen, haben wir den adaptiven Algorithmus nicht implementiert.

2.3.3 Flächeninhalt, Volumen, Schwerpunkt und Trägheitsmoment

Wir betrachten noch zwei spezielle Integrale für G–Spline–Flächen. Der Flächeninhalt $A(S)$ einer Fläche S ist definiert durch

$$A(S) := \int_S 1 \, d\sigma. \tag{2.71}$$

Für den Flächeninhalt der G–Spline–Fläche genügt es somit, das Integral

$$\iint_{[0,1]^2} \left\| \frac{\partial S_i}{\partial u} \times \frac{\partial S_i}{\partial v} \right\| d(u,v). \tag{2.72}$$

pro Bézierfläche $S_i : [0,1]^2 \to \mathbb{R}^3$ zu berechnen. Dies läßt sich durch eine einfache Änderung der Trapezregel im Romberg–Algorithmus implementieren.

Für eine geschlossene G–Spline–Fläche können wir auch das Volumen berechnen. Für das Volumenintegral einer Fläche S, welche das Volumen V einschließt, gilt nach dem Gauß'schen Integrationssatz, daß

$$V(S) = \int_V 1 \, dV = \int_V \operatorname{div} \begin{pmatrix} x \\ 0 \\ 0 \end{pmatrix} dV = \int_S \left\langle \begin{pmatrix} x \\ 0 \\ 0 \end{pmatrix}, n \right\rangle d\sigma, \tag{2.73}$$

Toleranz		Romberg–Algorithmus		Trapezregel	
`prec`	`96 * prec`	Ergebnis	Iterationen	Ergebnis	Iterationen
1e-02	0.96	0.6139044962	1	0.614231858	1
1e-03	0.96e-01	0.6139044962	1	0.614231858	1
1e-04	0.96e-02	0.6125966128	1.979166667	0.6138292174	1.8125
1e-05	0.96e-03	0.612233068	3.135416667	0.6122170234	3.875
1e-06	0.96e-04	0.6119397563	4.666666667	0.6119387735	5.614583333
1e-07	0.96e-05	0.611937454	6.208333333	0.6119354698	7.520833333
1e-08	0.96e-06	0.6119371768	7.677083333	0.6119371991	9.458333333

Abbildung 2.10: Volumenintegral für die Fläche aus der Abbildung 1.19 (einfacher Würfel)

wobei n der nach außen gerichtete Einheitsnormalenvektor von S ist. Analog erhält man für das Volumenintegral auch die Formeln

$$V(S) = \int_S \left\langle \begin{pmatrix} 0 \\ y \\ 0 \end{pmatrix}, n \right\rangle d\sigma = \int_S \left\langle \begin{pmatrix} 0 \\ 0 \\ z \end{pmatrix}, n \right\rangle d\sigma. \tag{2.74}$$

Daraus folgt

$$V(S) = \frac{1}{3} \int_S \left\langle \begin{pmatrix} x \\ y \\ z \end{pmatrix}, n \right\rangle d\sigma. \tag{2.75}$$

Dies entspricht dem Oberflächenintegral des Vektorfeldes $\left[x, y, z\right]^T$. Nachdem in dieser Formel alle drei Koordinaten vorkommen, ist sie auch numerisch stabiler. Für geschlossene G–Spline–Flächen erhalten wir das Volumen damit aus der Summe der Integrale

$$\frac{1}{3} \iint_{[0,1]^2} \langle S_i, N \rangle \, d(u, v) \tag{2.76}$$

pro Bézierfläche S_i, wobei N der nach außen gerichtete Normalenvektor der Fläche ist. Durch eine entsprechende Änderung der Trapezregel können wir so auch das Volumen berechnen. Dabei überprüfen wir allerdings nicht, ob die Fläche geschlossen ist und damit das Volumen überhaupt existiert. In wieweit das berechnete Integral definiert ist, bleibt dem Benutzer überlassen.

Die Tabelle in Abbildung 2.10 enthält die Ergebnisse der Volumenberechnung des einfachen Würfels aus Abbildung 1.19 für verschiedene Toleranzwerte. Zum Vergleich geben wir neben den Ergebnissen des Romberg–Algorithmus' auch die der bivariaten Trapezregel an. Die Iterationen sind die mittlere Anzahl der Iterationen, die für ein Flächenstück benötigt wurden, um die gewünschte Genauigkeit zu erreichen. Der Würfel besteht aus 96 Flächenstücken, d.h. die gesamte Toleranz ist 96 mal größer als die Toleranz für die einzelnen Flächenstücke. Man sieht an der Tabelle, daß die mittlere Anzahl der Iterationen für die Trapezregel schneller als für den Romerg–Algorithmus mit dem Verkleinern der Toleranzwerte steigt.

Mit Hilfe der vorgestellten Integrationsalgorithmen können wir den Schwerpunkt einer Fläche S berechnen. Sei dazu $\rho : S \to \mathbb{R}$ die Flächendichte von S. Dann ist die Masse von S

$$m(S) := \int_S \rho d\sigma. \tag{2.77}$$

Der Schwerpunkt $c(S)$ ist

$$c(S) := \frac{1}{m(S)} \begin{pmatrix} \int_S x\rho \, d\sigma \\ \int_S y\rho \, d\sigma \\ \int_S z\rho \, d\sigma \end{pmatrix}. \tag{2.78}$$

Objekt Abbildung	Flächenstück `prec`	Volumen Fläche	[Iter.] [Iter.]	Schwerpunktintegrale x y z	[Iter.] [Iter.] [Iter.]	Schwerpkt. x y z
einfacher	96	0.61	[3.1354]	1.78	[3.5625]	0.5
Würfel	1.04167e-05	3.57	[4.25]	1.78	[3.5625]	0.5
1.19				1.78	[3.5625]	0.5
Verdrehter	992	13.44	[4.1069]	65.43	[4.5847]	1.38
Würfel	1.00805e-06	47.54	[4.2480]	66.01	[4.5796]	1.39
1.20				63.15	[4.4516]	1.33
Acht	800	11.23	[3.9888]	100.36	[4.6563]	2.50
1.21	1.25e-06	40.15	[3.975]	60.22	[4.3975]	1.50
				20.07	[3.8625]	0.50
Verdrehte	928	13.04	[4.8114]	113.48	[5.3179]	2.50
Acht	1.07759e-06	45.39	[4.6789]	68.09	[5.0129]	1.50
1.22				68.09	[5.0065]	1.50
Kreuz	480	5.84	[3.7125]	30.37	[4.0583]	1.50
1.23	1.72414e-06	20.25	[3.95]	30.37	[4.0375]	1.50
				30.37	[4.0583]	1.50
T	1408	4.01	[3.6235]	24.15	[3.8274]	1.50
1.24	7.10227e-07	16.10	[3.5866]	30.35	[3.9517]	1.89
				8.05	[3.1719]	0.50

Abbildung 2.11: Integrale und Flächenschwerpunkte verschiedener Objekte

Für $\rho \equiv 1$ zeigt die Tabelle in Abbildung 2.11 das Volumen, den Flächeninhalt, der hier physikalisch der Masse der Fläche entspricht, die obigen drei Integrale für den Schwerpunkt und die Koordinaten des Schwerpunktes für die im Kapitel 1 vorgestellten Flächen. Die Toleranzen wurden so gewählt, daß die gesamte Toleranz in etwa 0.001 beträgt. Die in eckigen Klammern angegebenen Werte entsprechen der mittleren Anzahl der Iterationen pro Flächenstück. Es sei auch darauf hingewiesen, daß der Integrand für die Schwerpunktintegrale durch eine G–Spline–Funktion angegeben wurde.

Wir können ebenfalls das Trägheitsmoment $J(S)$ einer Fläche S mit der Dichte ρ bzgl. einer Rotationsachse berechnen,

$$J(S) := \int_S r^2 \rho \, d\sigma. \tag{2.79}$$

$r : S \to \mathbb{R}$ weist jedem Punkt der Fläche seinen senkrechten Abstand zu der Rotationsachse zu. Dabei geben wir $r^2\rho$ wieder als G–Spline–Funktion auf der Fläche an. Für das Möbiusband aus Abbildung 1.16 mit der Dichte $\rho \equiv 1$ ist das Trägheitsmoment um die Mittelachse gleich 57.98 bei einem Flächeninhalt von 14.24. Je weiter die einzelnen Masseteilchen der Fläche von der Rotationsachse entfernt sind, um so stärker wirken sie sich auf die Rotation aus. Die Funktion $r^2\rho$ ist für das Möbiusband in Abbildung 2.12 dargestellt. Nachdem das Möbiusband nicht orientierbar ist, eignet sich die reine Farbdarstellung der Funktion am besten.

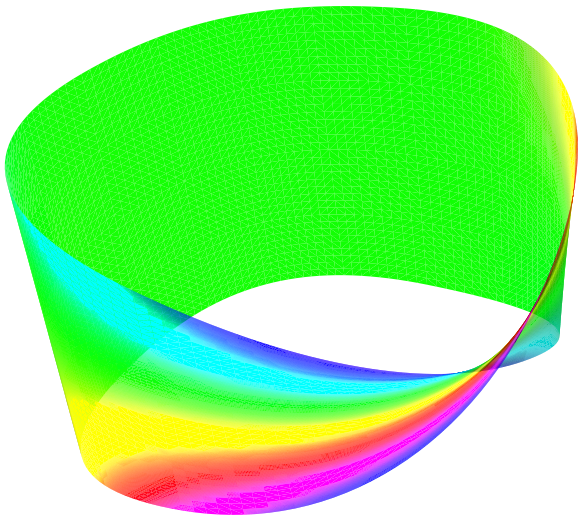


Abbildung 2.12: Trägheitsmoment eines Möbiusbandes

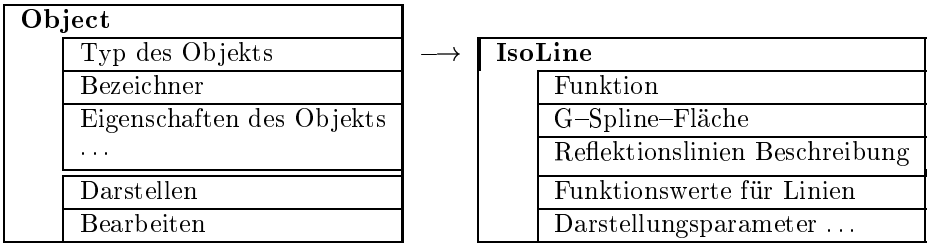


Abbildung 2.13: `IsoLine` Klasse

2.4 Isolinien

Neben der Darstellung durch Farben bzw. als Gitter oder Fläche über der Fläche können wir G–Spline–Funktionen auch durch Isolinien oder Niveaulinien darstellen. Dazu zeichnen wir auf der Fläche eine Linie, entlang der die Funktion den gleichen Wert hat. Die Werte, für die wir solche Linien zeichnen, können entweder äquidistant sein, oder wir können eine Liste spezieller Werte angeben.

Im Prinzip ist der Algorithmus für diese Darstellung ein verallgemeinerter Plotter für Contourlinien einer Funktion $f : \mathbb{R}^2 \to \mathbb{R}$. Für die Isolinien leiten wir von `Object` eine neue Klasse `IsoLine` ab, die Zeiger auf die Fläche und die Funktion besitzt und die Darstellung der Isolinien genauer beschreibt, wie in Abbildung 2.13 dargestellt. Wir gehen davon aus, daß die Fläche und die Funktion bereits durch biquadratische Bézierkontrollnetze beschrieben werden. Wieder bearbeiten wir jedes Bézierkontrollnetz einzeln. Der Algorithmus `isolines` zeichnet die Isolinien für das Bézierkontrollnetz `fb` der Funktion, auf der durch das Bézierkontrollnetz `b` gegebenen Fläche, für den Parameterbereich $[\texttt{u0}, \texttt{u1}] \times [\texttt{v0}, \texttt{v1}] \subset [0,1]^2$. `isolines` wird rekursiv aufgerufen, um Unterschiede in der Dichte der Linien auszugleichen. Dabei sollte aber die maximale Rekursionstiefe möglichst klein gewählt werden. Der erste Aufruf erfolgt natürlich für den Parameterbereich $[0,1]^2$.

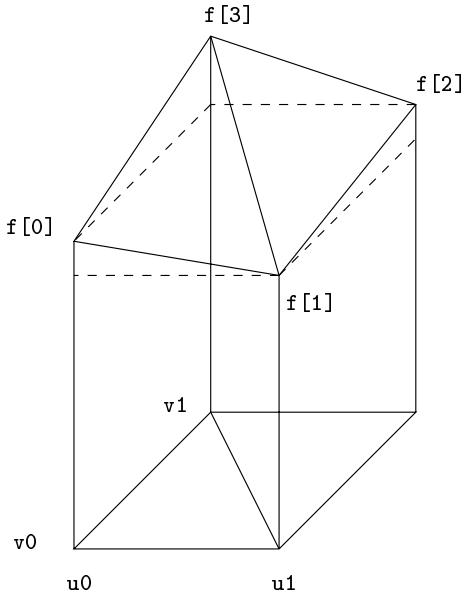


Abbildung 2.14: Isolinien Funktionsauswertung

Algorithmus 2.2. `isolines`
Isolinien Plotter

I. Falls die Funktion schon wenigstens einmal rekursiv aufgerufen wurde, d.h. falls die Rekursionstiefe `level` größer als 0 ist, dann überprüfe, ob in dem ausgewählten Quadrat im Parameterbereich Linien gezeichnet werden können:

   A. Berechne die Funktionswerte `f[0:3]` an den vier Ecken des Parameterbereiches (siehe Abbildung 2.14).

   B. Überprüfe, ob durch jede Seite des Quadrats maximal eine Isolinie geht, dann zeichne die Linien für das Quadrat mittels `handle_quad`:

      1. Falls äquidistante Isolinien gezeichnet werden sollen, verschiebe und skaliere zunächst die Funktionswerte: `f[0:3] = (f[0:3] - base) / step`. Dabei ist `base` ein Funktionswert, für den Isolinien gezeichnet werden sollen und `step` der Abstand zwischen den Funktionswerten der Isolinien. Damit wird das Problem auf das von Isolinien für ganzzahlige Funktionswerte reduziert.

2. Berechne die Differenz zwischen den Funktionswerten an den Ecken des Quadrats.
3. Sind alle Differenzen 0, dann nimm an, die Funktion ist konstant und kehre vom rekursiven Funktionsaufruf zurück. Wenn die Unterteilung am Anfang nicht fein genug gewählt wurde, können hierbei allerdings Details verloren gehen.
4. Sind alle Beträge der Differenzen kleiner als `mindist`, dann geht durch jede Seite des Quadrats maximal eine Isolinie. Rufe für dieses Quadrat `handle_quad` zum Zeichnen der Isolinien auf und kehre dann vom rekursiven Funktionsaufruf zurück. Für äquidistante Isolinien ist `mindist` gleich 1, für eine Liste von Werten ist `mindist` die kleinste Differenz zweier direkt aufeinanderfolgender Werte. Dieser Wert muß vor dem Aufruf von `isolines` bestimmt werden.

C. Falls die maximale Rekursionstiefe `maxlevel` erreicht wurde, unterteile das verbleibende Quadrat im Parameterbereich mittels bilinearer Interpolation in Teilquadrate:

1. Berechne folgende ganzzahlige Werte für die Iteration über die Teilquadrate:
```
uimax = abs(max(f[1] - f[0], f[2] - f[3]) / mindist) + 1
vimax = abs(max(f[3] - f[0], f[2] - f[1]) / mindist) + 1
uskip = (uimax / maxlines) + 1
vskip = (vimax / maxlines) + 1
```
wobei `maxlines` die maximal erlaubte Anzahl von Isolinien pro Quadrat festlegt.
2. Zerlege den Parameterbereich in Teilquadrate über eine durch obige Werte bestimmte Iteration und rufe für jedes Teilquadrat `handle_quad` auf.
`uimax` und `vimax` geben die maximale Anzahl der Teilintervalle in u– bzw. v–Richtung an. `uksip` und `vksip` geben an, wieviele der Teilintervalle pro Iterationsschritt in u– bzw. v–Richtung übersprungen werden.
3. Kehre vom rekursiven Funktionsaufruf zurück.

II. Zerlege den quadratischen Parameterbereich $[\texttt{u0},\texttt{u1}] \times [\texttt{v0},\texttt{v1}] \subset [0,1]^2$ in $\texttt{resolution}^2$ gleichgroße Teilquadrate und rufe `isolines` rekursiv für jedes dieser Quadrate mit einer um 1 erhöhten Rekursionstiefe auf.

Wie in Abbildung 2.15 gezeigt, unterteilen wir den Parameterbereich in Quadrate solange, bis nur noch maximal eine Linie durch jede Seite des Quadrats geht. Haben wir vorher die maximale Rekursionstiefe erreicht, dann legen wir über das Quadrat im Parameterbereich noch einmal ein Gitter, damit wir wieder Quadrate erhalten, durch deren Seiten maximal eine Linie geht. Das Gitter wird durch die Variablen aus Schritt refit:KuF:bilinf bestimmt. Es gibt zusätzlich den Parameter `maxlines` zur Begrenzung der Anzahl der Linien, die pro Quadrat des letzten Rekursionsschritts gezeichnet werden. Damit wird verhindert, daß hier ein zu feines Gitter erzeugt wird. Die Funktionswerte auf diesem Gitter werden durch bilineare Interpolation approximiert. Vor allem bei äquidistanten Isolinien für Funktionen, bei denen die Werte in einzelnen Bereichen stark ansteigen oder abfallen, aber sich sonst eher gering verändern, verbessert dies das Verhalten des Algorithmus'.

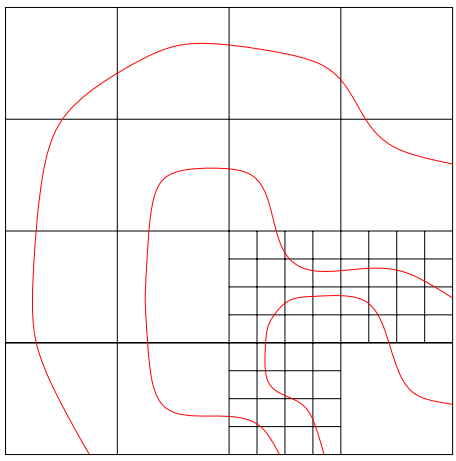


Abbildung 2.15: `isolines` Algorithmus

Die Interpolationsformel für die bilineare Interpolation von f auf dem Quadrat $[u_0,u_1] \times [v_0,v_1]$ bei Schritt I–C im Algorithmus lautet

$$\begin{aligned} p(u,v) = & \frac{v_1 - v}{v_1 - v_0} \left(\frac{u_1 - u}{u_1 - u_0} f(u_0,v_0) + \frac{u - u_0}{u_1 - u_0} f(u_1,v_0) \right) + \\ & \frac{v - v_0}{v_1 - v_0} \left(\frac{u_1 - u}{u_1 - u_0} f(u_0,v_1) + \frac{u - u_0}{u_1 - u_0} f(u_1,v_1) \right). \end{aligned} \tag{2.80}$$

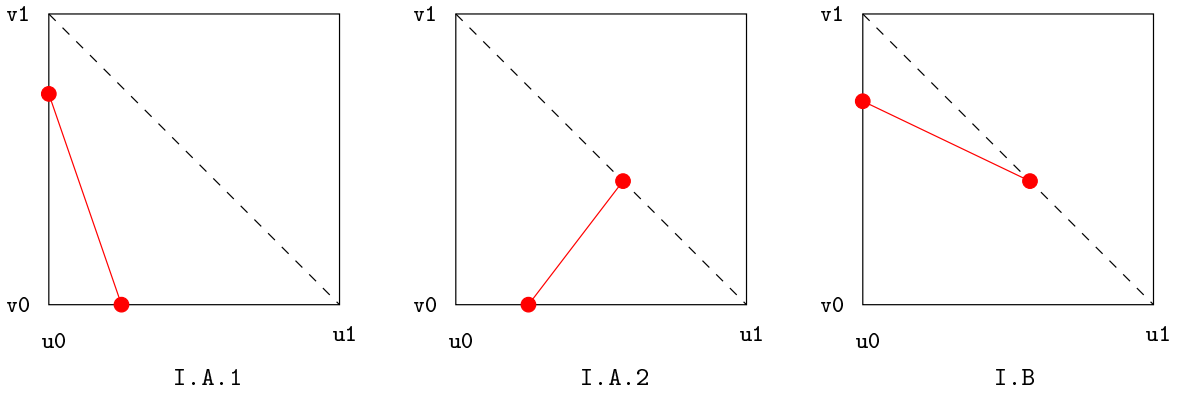


Abbildung 2.16: `handle_quad` Algorithmus

Die Linien für ein einzelnes Quadrat werden schließlich durch den Algorithmus `handle_quad` gezeichnet. Hier wird einfach überprüft, von welcher Seite des Quadrats zu welcher Seite wir eine Linie zeichnen müssen. Dazu wird das Quadrat zusätzlich in zwei Dreiecke aufgeteilt und der Schnittpunkt mit der Diagonalen berücksichtigt. Optional kann die Linie auch noch in der Farbe für den für sie vorgesehenen Funktionswert gezeichnet werden. Außerdem kann man die Linien längs der Einheitsnormalen der Fläche um einen festen Wert verschieben, um sicherzustellen, daß sie nicht von der Fläche überdeckt werden.

Algorithmus 2.3. `handle_quad`
Isolinien für ein einzelnes Teilquadrat zeichnen

I. Zeichne die Linie für das Dreieck `(u0,v0)`, `(u1,v0)`, `(u0,v1)`:

   A. Überprüfe, ob eine Isolinie durch die Strecke von `(u0,v0)` nach `(u1,v0)` geht. Wenn ja, dann unterscheide folgende zwei Fälle:

      1. Geht eine Isolinie zum gleichen Funktionswert durch die Strecke von `(u0,v0)` nach `(u0,v1)`, dann zeichne die Verbindungsstrecke.
      2. Geht eine Isolinie zum gleichen Funktionswert durch die Strecke von `(u1,v0)` nach `(u0,v1)`, dann zeichne die Verbindungsstrecke.

   B. Überprüfe, ob eine Isolinie durch die Strecke von `(u0,v0)` nach `(u0,v1)` geht. Geht eine Isolinie zum gleichen Funktionswert auch durch die Strecke von `(u1,v0)` nach `(u0,v1)`, dann zeichne die Verbindungsstrecke.

II. Zeichne die Linie für das Dreieck `(u1,v0)`, `(u1,v1)`, `(u0,v1)`:

   A. Überprüfe, ob eine Isolinie durch die Strecke von `(u1,v1)` nach `(u0,v1)` geht. Wenn ja, dann unterscheide folgende zwei Fälle:

      1. Geht eine Isolinie zum gleichen Funktionswert durch die Strecke von `(u1,v0)` nach `(u1,v1)`, dann zeichne die Verbindungsstrecke.
      2. Geht eine Isolinie zum gleichen Funktionswert durch die Strecke von `(u1,v0)` nach `(u0,v1)`, dann zeichne die Verbindungsstrecke.

   B. Überprüfe, ob eine Isolinie durch die Strecke von `(u1,v0)` nach `(u1,v1)` geht. Geht eine Isolinie zum gleichen Funktionswert auch durch die Strecke von `(u1,v0)` nach `(u0,v1)`, dann zeichne die Verbindungsstrecke.

Wir zerlegen zunächst das Quadrat in zwei Dreiecke über die Diagonale von `(u1,v0)` nach `(u0,v1)`. In beiden Dreiecken überprüfen wir dann, von welcher Seite zu welcher Seite wir eventuell eine Linie

ziehen müssen. Diese Linie wird auf dem Bézierflächenstück der Fläche, welches zum Bézierkontrollnetz der Funktion gehört, zu den entsprechenden Parameterwerten gezeichnet. Falls eine Isolinie durch eine der Seiten im Parameterbereich geht, wird der genaue Parameterwert durch lineare Interpolation auf der Seite berechnet. In Abbildung 2.16 sind die drei Fälle für das Dreieck `(u0,v0)`, `(u1,v0)`, `(u0,v1)` dargestellt.

Der Test, ob durch eine Seite eine Isolinie geht, hängt von der Weise ab, wie die Funktionswerte für die Isolinien gegeben wurden. Verwenden wir äquidistante Linien, wird `handle_quad` bereits mit den auf ganzzahlige Werte verschobenen und skalierten Funktionswerten aufgerufen. Wir müssen dann nur noch überprüfen, ob zwischen zwei Funktionswerten eine ganze Zahl liegt. Für eine Liste von Funktionswerten ist dies nicht ganz so einfach. Hierfür müssen wir für jedes Element dieser Liste überprüfen, ob es zwischen den zwei Funktionswerten an den Ecken liegt. Wenn wir die Liste der Funktionswerte sortieren, können wir dabei ein paar Vergleiche einsparen.

Mit Hilfe des Isolinien–Algorithmus' können wir auch Contourplots erstellen. Dazu verwenden wir für die Kontrollpunkte der Fläche ein ebenes Gitter. Auf diesem Gitter geben wir die Funktionskontrollpunkte für den Contourplot an. Zeichnen wir die Isolinien für diese Funktion, erhalten wir einen Contourplot. So können wir die Funktionskontrollpunkte z.B. über die Parametrisierung des Affensattels

$$(u, v) \mapsto \begin{bmatrix} u \\ v \\ u^3 - 3uv^2 \end{bmatrix} \tag{2.81}$$

mit $u = -1 : 0.2 : 1$, $v = -1 : 0.2 : 1$ festlegen. Daraus ergibt sich der Contourplot aus Abbildung 2.17. In dieser Abbildung haben wir zusätzlich die Funktion über Farbwerte auf der Ebene dargestellt. Wir können auch die Höhenfunktion $(u, v) \mapsto (u^3 - 3uv^2)$ als Isolinien auf dem Affensattel darstellen. In Abbildung 2.18 haben wir die Isolinien etwas oberhalb und unterhalb der Fläche gezeichnet.

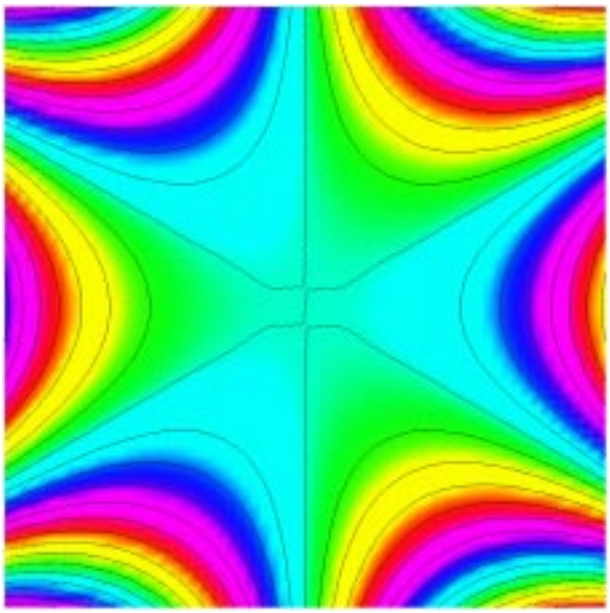


Abbildung 2.17: Contourplot des Affensattels

In Abbildung 2.19 stellen wir die Funktion $(x, y) \to \sin(x) + \cos(x)$ auf der in Abbildung 1.22 vorgestellten verdrehten Acht dar. Der Ursprung liegt dabei "links hinten" in der Ecke des entsprechenden erzeugenden Würfels. Ein Contourplot auf einer Ebene ist für diese Funktion nicht mehr möglich.

Neben den Isolinien für Funktionen können wir sie auch für die Fläche selbst zeichnen, d.h. wir verwenden die Flächenfunktion selbst für Isolinien. In diesem Fall markieren die Isolinien die Kurven auf der Fläche mit konstantem Abstand vom Ursprung. Abbildung 2.20 zeigt dies für das in Abbildung 1.23 vorgestellte Kreuz. Der Ursprung wurde dabei in den Mittelpunkt des Kreuzes gelegt.

Interessanter sind aber Reflektionslinien. Dazu definieren wir zunächst eine Ebene im Raum über einen Punkt P und zwei linear unabhängige Vektoren p_1 und p_2. Auf dieser Ebene seien nun parallele Lichtquellen im gleichen Abstand angebracht. In unserem einfachen Modell sind die Lichtquellen unendlich lang und besitzen keine Breite. Weiter nehmen wir an, daß die Intensität der Lichtstrahlen unabhängig von der Entfernung immer gleich groß sei. p_1 gebe die Richtung der Lichtquellen an und p_2 sei der Abstand zwischen den parallelen Lichtquellen. Das Licht von diesen Lichtquellen reflektieren wir an der Fläche S für einen vorgegebenen Beobachtungspunkt E. Diese Situation ist schematisch in Abbildung 2.21 dargestellt.

Sei Q ein beliebiger Punkt auf S. Die Strecke $q = E - Q$ legt den Ausfallswinkel für einen Reflekti-

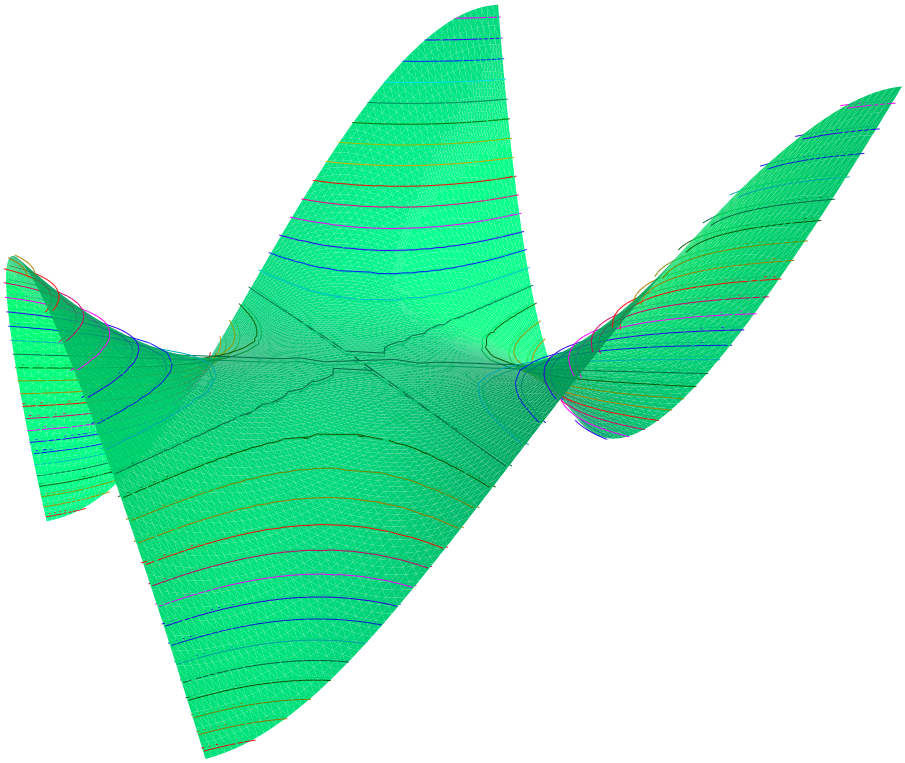


Abbildung 2.18: Höhenlinien des Affensattels

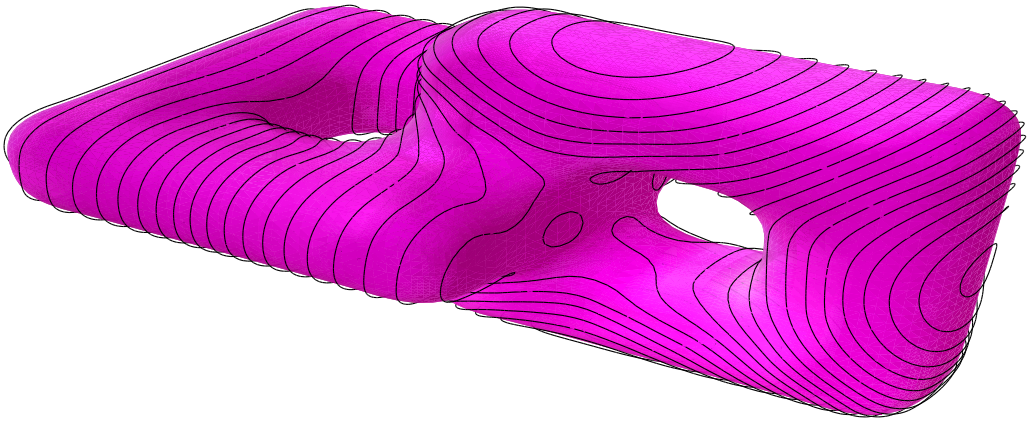


Abbildung 2.19: Isolinien auf der verdrehten Acht

onsstrahl fest. Da der Einfallswinkel genauso groß wie der Ausfallswinkel sein muß, erhalten wir die Richtung r des Reflektionsstrahls aus

$$r = q - 2\left(q - \frac{\langle q, n\rangle}{\|N\|}N\right) = 2\frac{\langle q, n\rangle}{\|N\|}N - q, \tag{2.82}$$

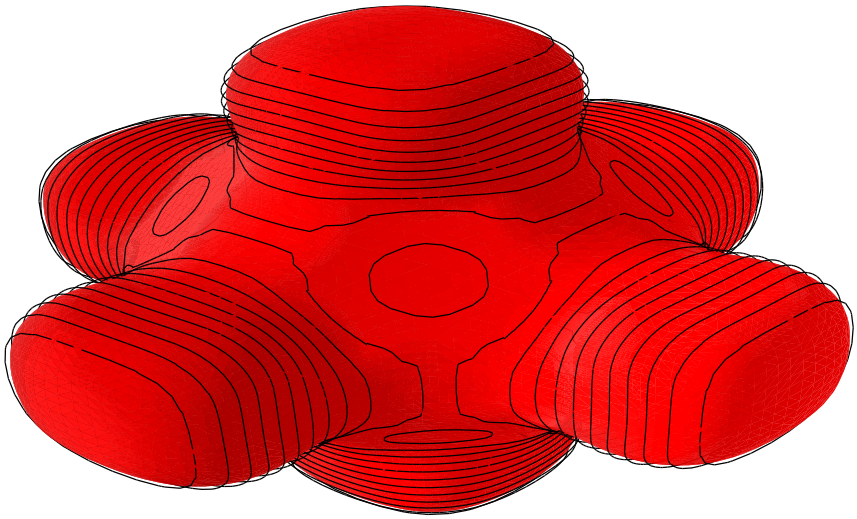


Abbildung 2.20: Isolinien des Kreuzes

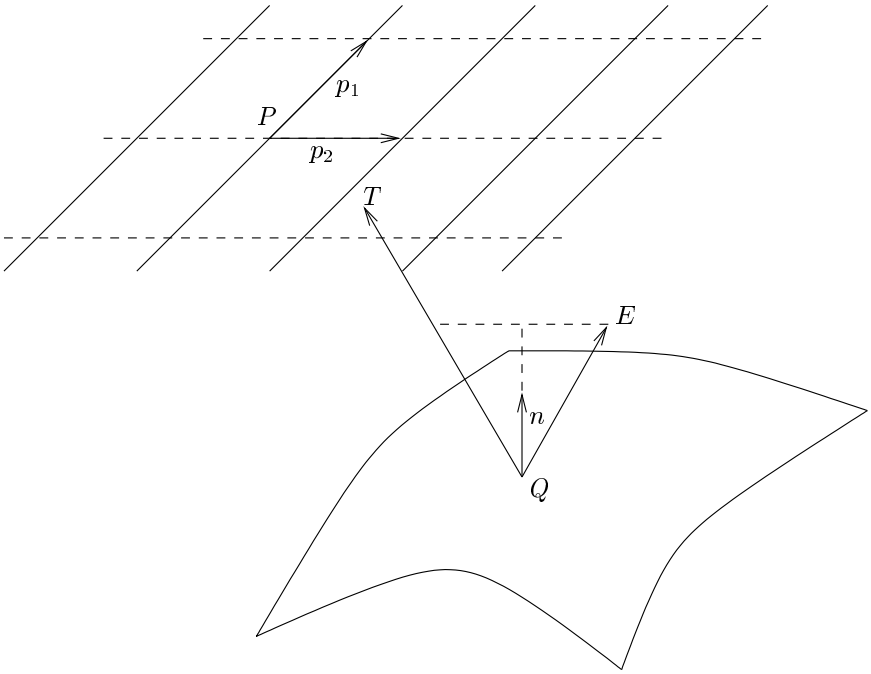


Abbildung 2.21: Reflektionslinien

wobei N der Normalenvektor von S bei Q ist. Für den Schnittpunkt T des Reflektionsstrahls mit der Ebene erhalten wir die lineare Gleichung

$$P + p_1 t_1 + p_2 t_2 = T = Q + rs \tag{2.83}$$

mit den Unbekannten t_1, t_2, s. In Matrixform ergibt dies

$$\begin{bmatrix} p_1 & p_2 & -r \end{bmatrix} \begin{bmatrix} t_1 \\ t_2 \\ s \end{bmatrix} = Q - P. \tag{2.84}$$

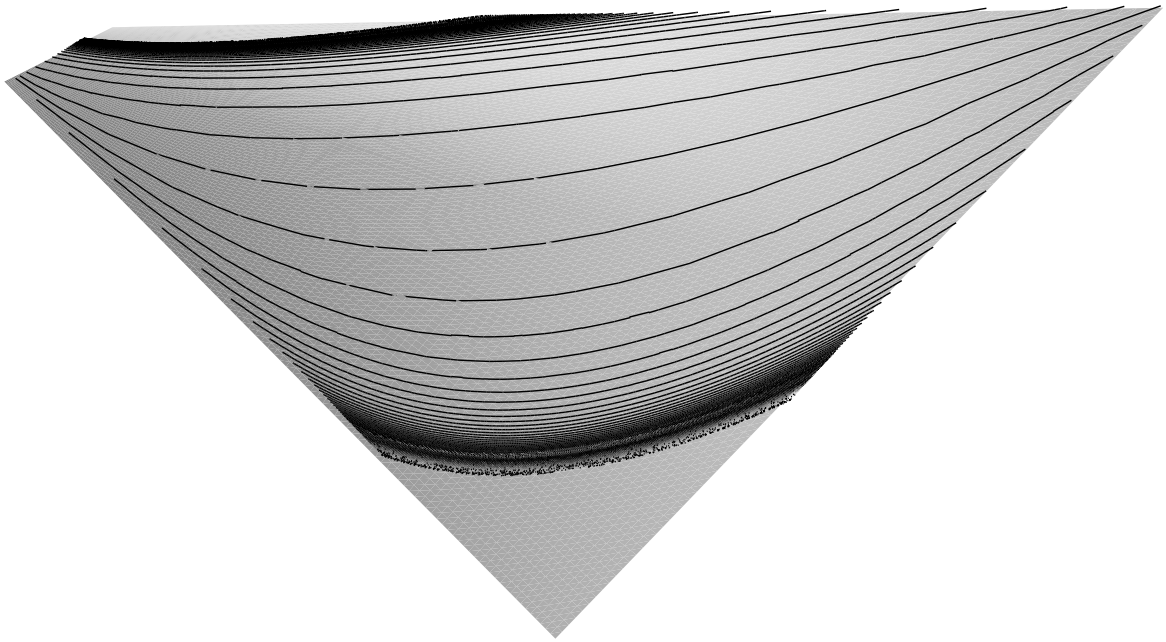


Abbildung 2.22: Reflektionslinien auf einem hyperbolischen Paraboloid

Numerisch können wir dieses System über einen Lösungsalgorithmus für lineare Gleichungssysteme lösen. Uns interessiert allerdings nicht der Schnittpunkt mit der Ebene, sondern ob der Reflektionsstrahl eine Lichtquelle auf der Ebene trifft. Dies ist der Fall, wenn t_2 ganzzahlig ist. Ist s negativ, dann findet die Reflektion "hinter" der Ebene statt. Um diesen Fall auszuschließen, setzen wir die Reflektionsfunktion auf 0, wenn s negativ ist.

Die Reflektionslinien können wir über den Isolinien–Algorithmus zeichnen, indem wir t_2 in Abhängigkeit von s als Funktionswert verwenden. Auch wenn es kein realistisches Reflektionsmodell ist, kann man über diese Reflektionslinien sehr schön kleine Unebenheiten, etc. in der Fläche erkennen. Wichtig ist allerdings dabei eine sinnvolle Wahl der Ebene und des Beobachtungspunktes. Es genügt auch meistens nicht, die Fläche nur mit einer Einstellung zu untersuchen.

Abbildung 2.22 zeigt die Reflektionslinien für ein Fläche, deren Kontrollpunkte über das hyperbolische Paraboloid

$$(u,v) \mapsto \begin{bmatrix} u \\ v \\ uv \end{bmatrix} \quad \text{mit } u = -1 : 0.2 : 1,\ v = -1 : 0.2 : 1 \tag{2.85}$$

bestimmt wurden. Die Reflektionsebene wurde dabei parallel zur x–y–Ebene über der Fläche gewählt. Nachdem die Reflektionsebene unendlich groß ist, erhalten wir am Rand sehr viele Reflektionslinien. Um den Aufwand für das Zeichnen dieser Linien über den Algorithmus `isolines` zu verkleinern, sollte sowohl `resolution`, als auch `maxlevel` und `maxlines` möglichst klein gewählt werden. Für das hyperbolische Paraboloid wurde `resolution` auf 3, `maxlevel` auf 6 und `maxlines` auf 2 gesetzt.

Abbildung 2.23 zeigt die Reflektionslinien für Irregularitäten der Ordnung 3, 4 und 5. Die Irregularitäten liegen alle um den Punkt $\left[1,1,1\right]^T$ und die Reflektionsebene ist durch den Punkt $\left[4,4,4\right]^T$, den Lichtquellenrichtungsvektor $\left[1,0,-1\right]^T$ und den Abstandsvektor $\left[-1,1,1\right]^T$ gegeben. Für die Irregularitäten der Ordnung 3 und 5 wurde `resolution` auf 6, `maxlevel` auf 3 und`maxlines` auf 2 gesetzt. Für die Irregularität der Ordnung 6 war `resolution` = 10, `maxlevel` = 2 und `maxlines` = 2.

Für die in Abbildung 1.21 vorgestellte Acht zeigt Abbildung 2.24 die Reflektionslinien für eine zur x–y–Ebene parallelen Ebene über der Acht. Aufgrund der Irregularitäten sind diese sehr unregelmäßig.

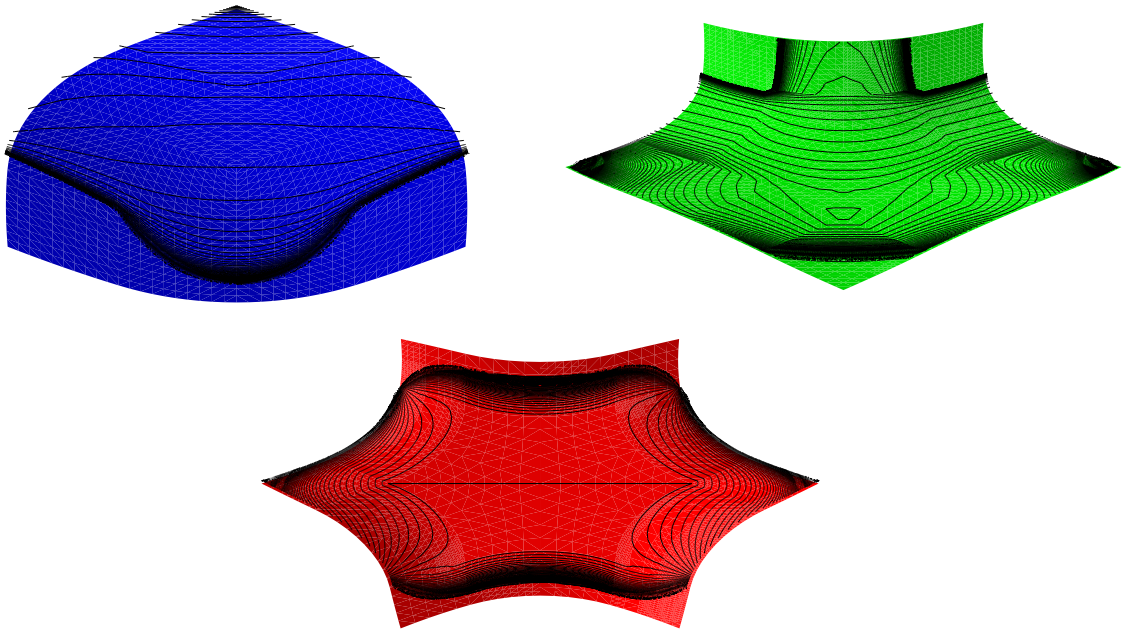


Abbildung 2.23: Reflektionslinien für Irregularitäten der Ordnung 3, 5 und 6

Auch ist es notwendig `maxlines` und `maxlevel` des Isolinien–Algorithmus' stark zu beschränken, um unnötigen Rechenaufwand am Rand des Reflektionsgebietes zu vermeiden. `maxlevel` wurde für diese Abbildung auf 1, `maxlines` auf 6 und `resolution` auf 15 gesetzt.

2.5 Krümmung

Als spezielle Funktionen betrachten wir abschließend die Krümmung von Flächen. Wir geben hier nur eine kurzen Überblick aus der Differentialgeometrie. Für eine umfassende Einführung sei beispielsweise auf [Car93] verwiesen, aus dem folgender kurzer Überblick zusammengestellt wurde.

Es sei $S : K \to \mathbb{R}^3$ eine reguläre Fläche und $p \in S$ ein Punkt auf dieser Fläche. Wir bezeichnen die Menge der Tangentenvektoren von allen Kurven in S durch p als Tangentialebene $T_p(S)$ an S in p. Das natürliche innere Produkt in $\mathbb{R}^3 \supset S$ induziert auf jedem $T_p(S)$ ein inneres Produkt $\langle \cdot, \cdot \rangle_p$. Dieses wird als eine symmetrische Bilinearform $I_p : T_p(S) \to \mathbb{R}$ mit $I_p(v) := \langle v, v \rangle_p$ dargestellt. Wir bezeichnen I_p als erste Fundamentalform von S in p.

Für ein $v \in T_p(S)$ existiert nach Definition eine Kurve $c : (-\epsilon, \epsilon) \to S$ auf S für $\epsilon \in \mathbb{R}_+$ mit $c(t) = S(u(t), v(t))$, so daß $c(0) = p$ und $c'(0) = v$ gilt. Dies liefert die folgende Form für I_p:

$$\begin{aligned} I_p(v) &= \langle c', c' \rangle_p \\ &= \langle S_u u' + S_v v', S_u u' + S_v v' \rangle_p \\ &= E(u')^2 + 2Fu'v' + G(v')^2 \end{aligned} \tag{2.86}$$

mit

$$E = \langle S_u, S_u \rangle_p, \quad F = \langle S_u, S_v \rangle_p, \quad G = \langle S_v, S_v \rangle_p, \tag{2.87}$$

wobei alle Funktionen bei $t = 0$ ausgewertet werden.

Ist $c : [a, b] \to \mathbb{R}^3$ eine nach der Bogenlänge $s \in [a, b]$ parametrisierte Kurve, dann ist $k(s) := |c''(s)|$ die Krümmung von c in s. Hieraus leiten wir die Krümmung für die Fläche S ab. Sei $n : S \to \mathbb{R}^3$,

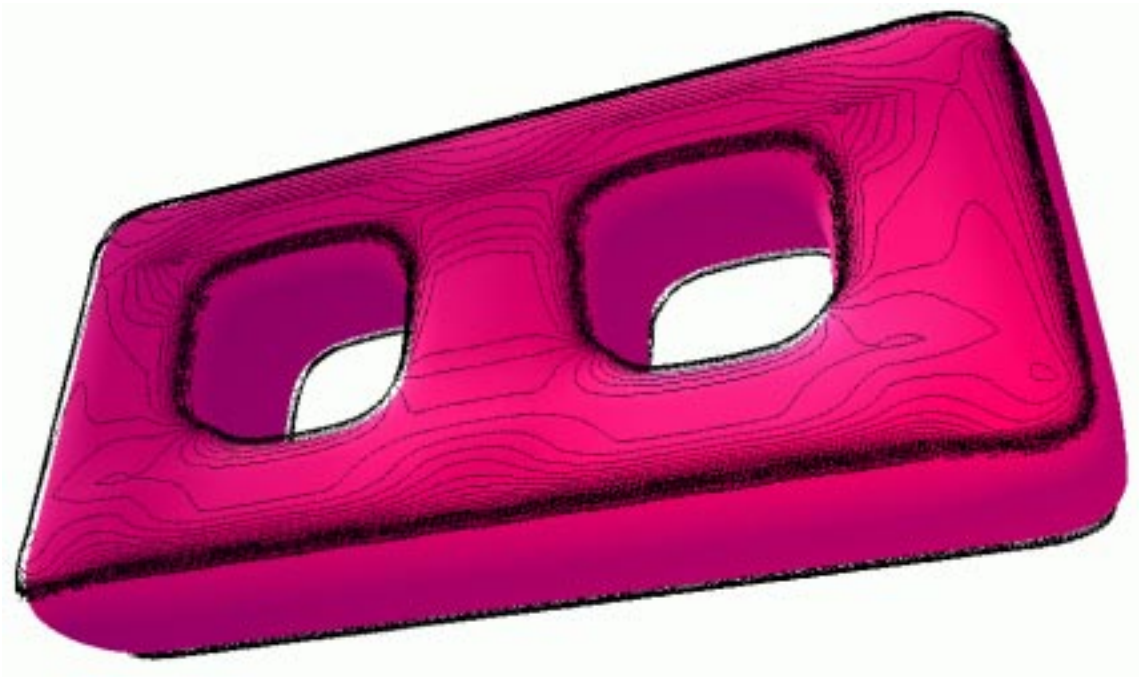


Abbildung 2.24: Reflektionslinien auf der Acht

$p \mapsto \frac{S_u \times S_v}{\|S_u \times S_v\|}$ die Abbildung, die jedem Punkt p von S den Einheitsnormalenvektor zuweist. n gibt so für jeden Punkt und damit für die ganze Fläche eine Orientierung an. Die Werte der Abbildung $n : S \to \mathbb{R}^3$ liegen alle in der Einheitssphäre $S^2 := \{(x, y, z) \in \mathbb{R}^3 : x^2 + y^2 + z^2 = 1\}$. Somit können wir die Gauß–Abbildung von S als $n : S \to S^2$ definieren. Diese Abbildung ist für orientierbare Flächen differenzierbar und das Differential dn_p von n ist eine lineare Abbildung von $T_p(S)$ in den Tangentenraum $T_{n(p)}(S^2)$. Nachdem beide Tangentenräume parallele Ebenen sind, kann man dn_p auch als lineare Abbildung auf $T_p(S)$ interpretieren. Weiter können wir hiermit die zweite Fundamentalform von S bei p als $II_p(v) := -\langle dn_p(v), v\rangle$ definieren.

Wir betrachten wieder eine reguläre, nach Bogenlänge s parametrisierte Kurve c, die ganz in S verläuft und es gelte $c(0) = p$. k sei die Krümmung von c in p und n_c sei der Normalenvektor an c. Dann ist $k_c = k\langle n_c, n\rangle$ die Normalkrümmung von c in p. Wir betrachten nun alle Kurven in S durch p. Die maximale und minimale Normalkrümmung heißen Hauptkrümmungen von S bei p. Weiter definieren wir die Gauß'sche Krümmung als Determinante von dn_p und der negative Wert der halben Spur von dn_p heißt die mittlere Krümmung.

Zur Vereinfachung der Notation stehen im folgenden die Funktionen für ihren Wert im Punkt p. Es gilt

$$c' = S_u u' + S_v v', \tag{2.88}$$

$$dn(c') = n'(u(t), v(t)) = n_u u' + n_v v'. \tag{2.89}$$

Da n_u und n_v zu $T_p(S)$ gehören, können wir sie wie folgt darstellen

$$\begin{aligned} n_u &= a_{11} S_u + a_{21} S_v, \\ n_v &= a_{12} S_u + a_{22} S_v. \end{aligned} \tag{2.90}$$

Damit erhalten wir die Darstellung

$$dn(c') = (a_{11} u' + a_{12} v')S_u + (a_{21} u' + a_{22} v')S_v. \tag{2.91}$$

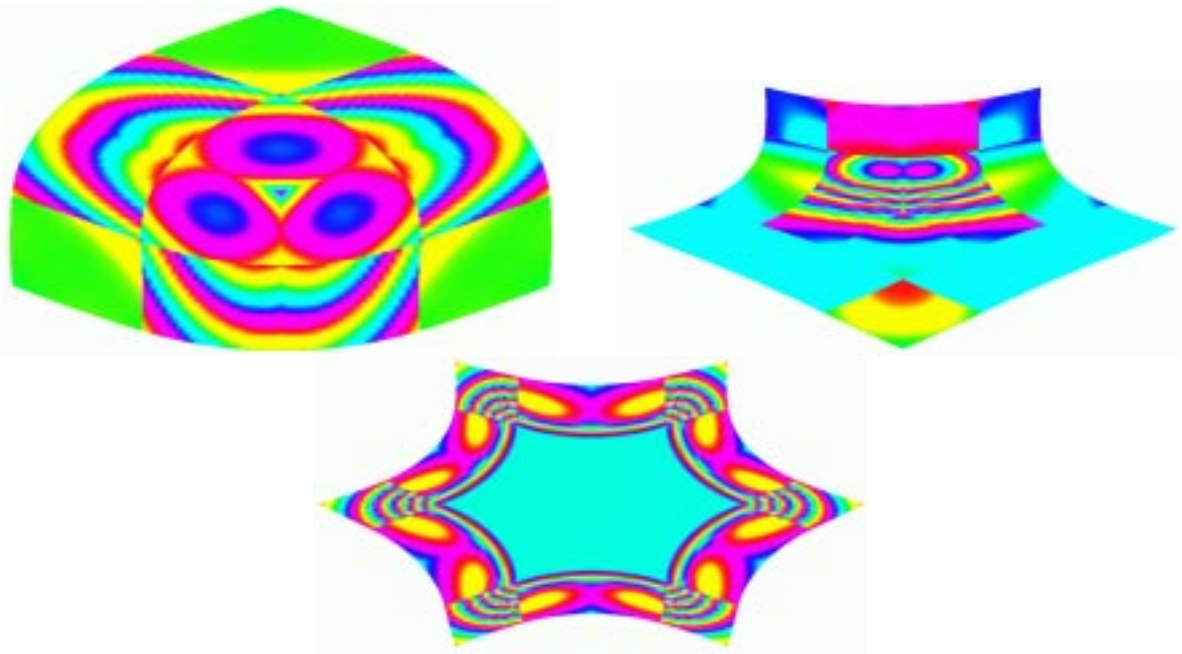


Abbildung 2.25: Gauß–Krümmung für Irregularitäten der Ordnung 3, 5 und 6

Die zweite Fundamentalform können wir schreiben als

$$\begin{aligned} II_p(c') &= -\langle dn(c'), c'\rangle \\ &= -\langle n_u u' + n_v v', S_u u' + S_v v'\rangle \\ &= e(u')^2 + 2fu'v' + g(v')^2. \end{aligned} \tag{2.92}$$

Wegen $\langle n, S_u\rangle = \langle n, S_v\rangle = 0$ gilt

$$\begin{aligned} e &= -\langle n_u, S_u\rangle = \langle n, S_{uu}\rangle = \frac{1}{\sqrt{EG-F^2}} \det \begin{bmatrix} S_{uu} \\ S_u \\ S_v \end{bmatrix}, \\ f &= -\langle n_v, S_u\rangle = \langle n, S_{uv}\rangle = \frac{1}{\sqrt{EG-F^2}} \det \begin{bmatrix} S_{uv} \\ S_u \\ S_v \end{bmatrix} = \langle n, S_{vu}\rangle = -\langle n_u, S_v\rangle, \\ g &= -\langle n_v, S_v\rangle = \langle n, S_{vv}\rangle = \frac{1}{\sqrt{EG-F^2}} \det \begin{bmatrix} S_{vv} \\ S_u \\ S_v \end{bmatrix}. \end{aligned} \tag{2.93}$$

Aus (2.90) können wir nun Gleichungen für die a_{ij} aufstellen,

$$\begin{aligned} -f &= a_{11}F + a_{21}G, \qquad -e = a_{11}E + a_{21}F, \\ -f &= a_{12}E + a_{22}F, \qquad -g = a_{12}F + a_{22}G. \end{aligned} \tag{2.94}$$

Hieraus erhalten wir

$$\begin{aligned} a_{11} &= \frac{fF - eG}{EG - F^2}, \qquad a_{21} = \frac{eF - fE}{EG - F^2}, \\ a_{12} &= \frac{gF - fG}{EG - F^2}, \qquad a_{22} = \frac{fF - gE}{EG - F^2}. \end{aligned} \tag{2.95}$$

Dies ergibt folgende Gleichung für die Gauß–Krümmung,

$$K = \det(a_{ij}) = \frac{eg - f^2}{EG - F^2}. \tag{2.96}$$

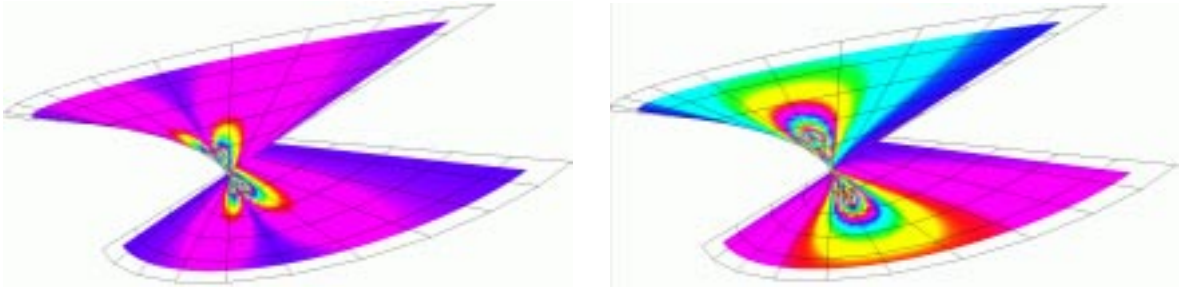


Abbildung 2.26: Gauß'sche (links) und mittlere (rechts) Krümmung des Whitneyschen Regenschirms

Für die mittlere Krümmung erhalten wir

$$H = -\frac{1}{2}(a_{11} + a_{22}) = \frac{1}{2}\frac{eG - 2fF + gE}{EG - F^2}. \tag{2.97}$$

Nachdem k_1 und k_2 Eigenwerte von dn_p sind, gilt außerdem

$$H = \frac{k_1 + k_2}{2}, \tag{2.98}$$

$$K = k_1 k_2. \tag{2.99}$$

Somit erfüllen die beiden Hauptkrümmungen die quadratische Gleichung

$${k_i}^2 - 2Hk_i + K = 0 \tag{2.100}$$

und deshalb gilt

$$k_i = H \pm \sqrt{H^2 - K}. \tag{2.101}$$

Berechnen wir E, F, G, e, f, g über (2.87) und (2.93), dann erhalten wir über obige Formeln die verschiedenen Krümmungen zu einem Punkt p. Damit können wir die Krümmungsfunktionen einfach implementieren, indem wir die Auswertung der Bézierkontrollnetze für die Funktion durch die Berechnung von E, F, G, e, f, g und der entsprechenden Krümmung für das Bézierkontrollnetz der Fläche ersetzen.

Wir stoßen hier allerdings auch an die Grenzen der biquadratischen G–Splines. Biquadratische Splineflächen können nur tangential stetig zusammengesetzt werden. Für die Krümmungen benötigen wir aber auch die zweiten Ableitungen zur Berechnung von e, f und g. Deshalb müssen die Krümmungsfunktionen nicht mehr stetig sein. Besonders wird dies an den irregulären Stellen des Kontrollnetzes sichtbar.

In Abbildung 2.25 stellen wir die Gauß–Krümmung in der Umgebung von Irregularitäten der Ordnung 3, 5 und 6 dar. Nachdem die Formeln für biquadratische G–Splines an den Irregularitäten nur aus den Bedingungen für tangentiales Zusammensetzen der Flächenstücke erzeugt wurden, ist die Krümmung natürlich nicht stetig. Für die Stetigkeit der Krümmung benötigen wir Splineflächen höherer Ordnung.

Für die über die Parametrisierung des Whitneyschen Regenschirms erzeugte Fläche aus Abbildung 2.5 zeigen wir in Abbildung 2.26 die Gauß'sche und die mittlere Krümmung.

Abbildung 2.27 zeigt die Gauß'sche Krümmung der in Abbildung 1.24 vorgestellten T–Fläche. Neben der Farbdarstellung stellen wir sie hier auch als farbige Fläche über dem T dar.

Abbildung 2.28 zeigt die Gauß'sche und die mittlere Krümmung der Fläche, deren Kontrollpunkte über die Parametrisierung (2.81) des Affensattels festgelegt wurden. Besonders bei der mittleren Krümmung

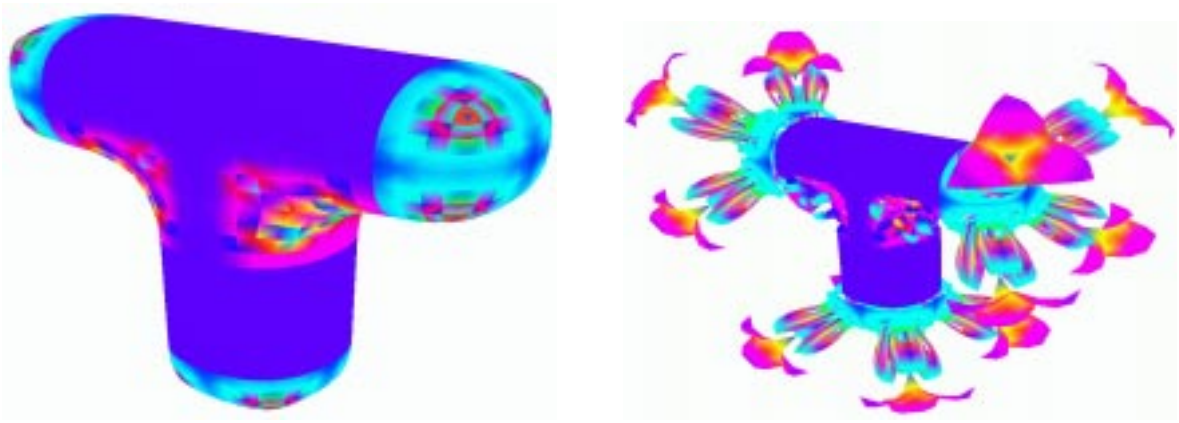


Abbildung 2.27: Gauß'sche Krümmung des T

erkennt man, daß die Funktion nicht mehr stetig ist. Die exakten Formeln für die Krümmungen des Affensattels sind

$$K = \frac{-36(u^2+v^2)}{(1+9u^4+18u^2v^2+9v^4)^2}, \quad H = \frac{-27u^5+54u^3v^2+81uv^4}{(1+9u^4+18u^2v^2+9v^4)^{\frac{3}{2}}} \tag{2.102}$$

(vgl. [Gra94]). Hierüber können wir auch Funktionskontrollpunkte für die Flächenkontrollpunkte mit $u = -1 : 0.1 : 1$, $v = -1 : 0.1 : 1$ festlegen. Diese Funktionen sind in Abbildung 2.29 dargestellt.

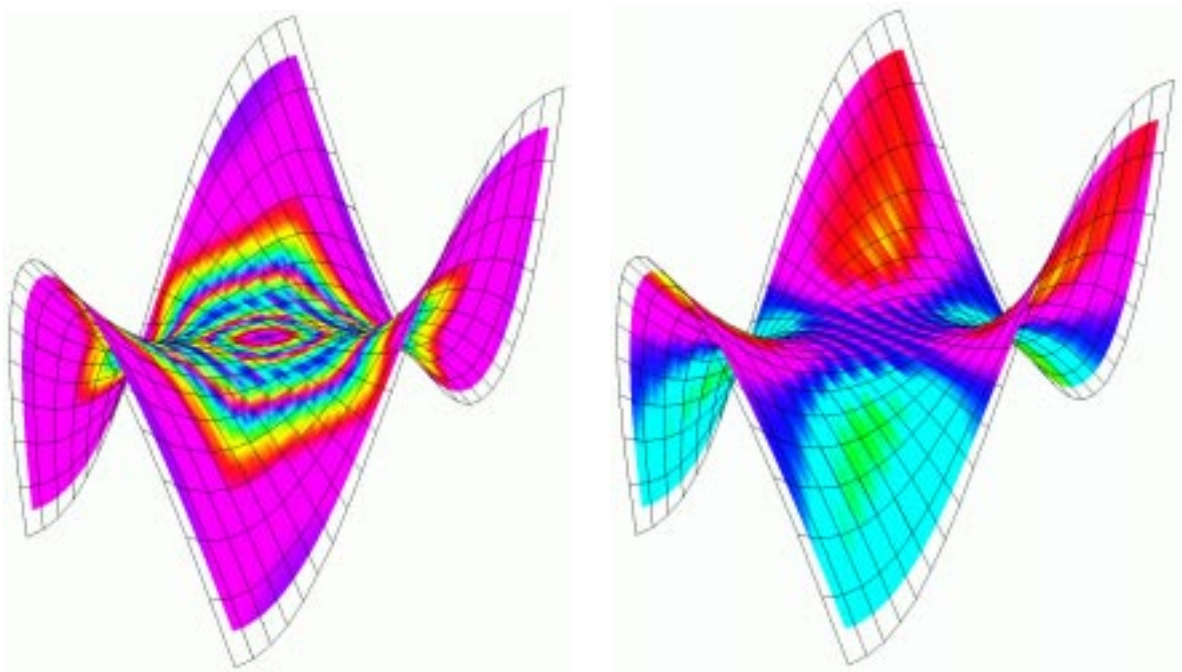


Abbildung 2.28: Gauß'sche (links) und mittlere (rechts) Krümmung des Affensattels

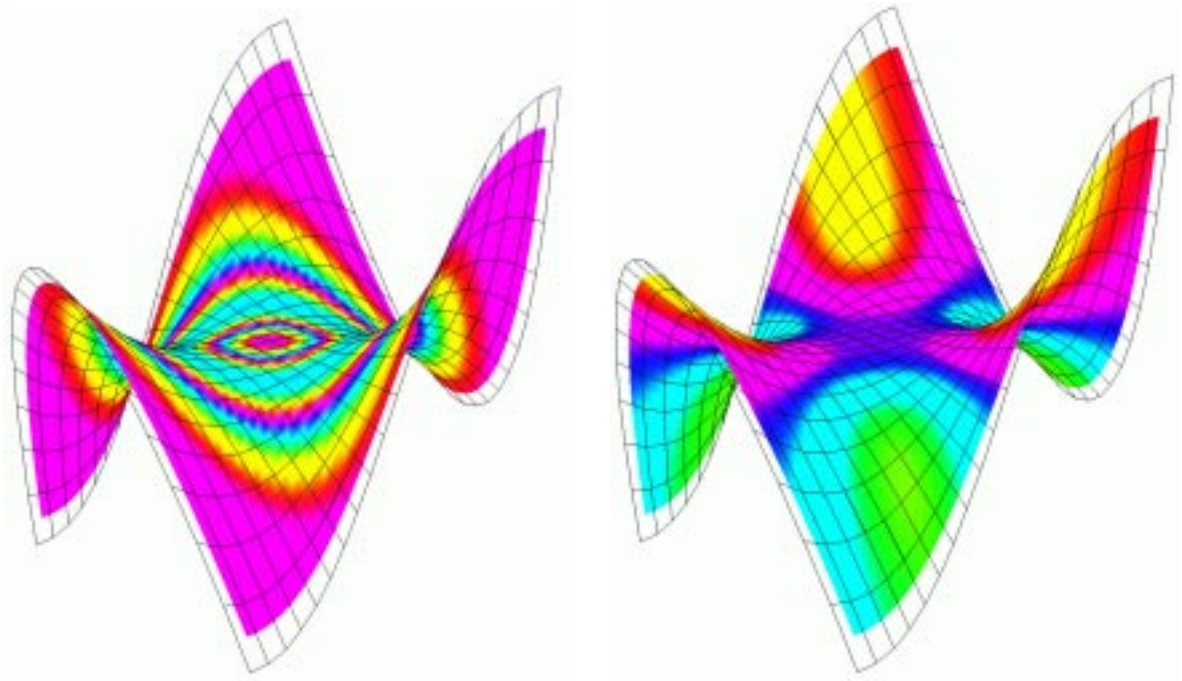


Abbildung 2.29: Gauß'sche (links) und mittlere (rechts) Krümmung des Affensattels als G–Spline–Funktion

Kapitel 3

Getrimmte Flächen

Getrimmte Flächen sind Flächen, aus denen Teile herausgeschnitten wurden. Auf diese Weise können wir z.B. Löcher für Durchdringungen anderer Körper in eine Fläche schneiden. Um biquadratische G–Spline–Flächen zu trimmen, schneiden wir aus den einzelnen Flächenstücken Teile aus. Diese Teile können durch Trimmkurven im Parameterbereich bestimmt werden, aber auch direkt im Raum der Fläche angegeben werden. Wir stellen hier zunächst eine Methode zur Behandlung solcher getrimmter Flächen dar und betrachten dann noch kurz die Auswirkungen auf Funktionen auf getrimmten Flächen und die Integrationsmethoden.

3.1 Trimming

Wir können Flächen sowohl durch die Beschreibung von Gebieten im Parameterbereich als auch durch die Angabe von Volumina im Raum der Fläche trimmen. Haben wir dann eine Testfunktion, mit der wir überprüfen können, ob die Parameter bzw. die Bildpunkte innerhalb oder außerhalb dieser Bereiche liegen, können wir die getrimmten Flächen darstellen.

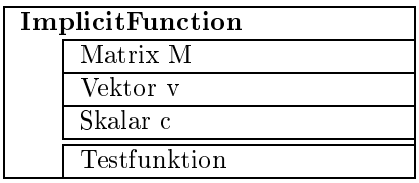


Abbildung 3.1: `ImplicitFunction` Klasse

Eine einfache, leicht zu implementierende Methode die Fläche im Raum mit Hilfe von Volumina zu trimmen, besteht in der impliziten Beschreibung der Volumina. Wir beschränken uns dazu auf die quadratische Form

$$x^T M x + v^T x < c, \tag{3.1}$$

wobei die 3×3 Matrix M, der drei–dimensionale Vektor v und $c \in \mathbb{R}$ beliebig gewählt werden können. Der so beschriebene Körper wird durch die quadratische Fläche $x^T M x + v^T x - c = 0$ begrenzt, wobei für quadratische Flächen M eine symmetrische Matrix sein muß. Der Test, ob ein Punkt x der Fläche in dem Volumen liegt, besteht einfach darin, die Gleichung (3.1) zu überprüfen. Eine solche quadratische, implizite Trimmfunktion beschreiben wir durch die in Abbildung 3.1 dargestellte Klasse `ImplicitFunction`. Man kann auch beliebige implizite Funktionen, die Volumina im $\mathbb{R}^3$ beschreiben, verwenden. Dabei würde sich nur (3.1) im Algorithmus ändern.

Im Parameterbereich geben wir die Trimmbereiche über Trimmkurven an. Diese schneiden Teile des Parameterbereiches aus, die wir auf die Fläche übertragen. Eine solche Trimmkurve beschreiben wir durch die Klasse `TrimCurve` aus Abbildung 3.2. Eine Trimmkurve beschreibt einen Trimmbereich für ein einzelnes Bézierflächenstück der G–Spline–Fläche. Die Kurve kann entweder ein Polygonzug oder eine geometrisch glatte, quadratische Splinekurve sein. Der Parameterbereich sei wie üblich $[0,1]^2$. Die Trimmkurve kann diesen Bereich aber verlassen.

TrimCurve	
	Kontrollpunkt der G–Spline–Fläche
	Liste der Trimmkurvenkontrollpunkte
	Ordnung der Trimmkurve
	Koordinaten der Polygonzugecken
	Process Funktion
	Testfunktion

Abbildung 3.2: `TrimCurve` Klasse

Für einen Polygonzug legen die Kontrollpunkte die Ecken fest. Zwei aufeinanderfolgende Kontrollpunkte geben jeweils eine Linie des Polygonzuges an. Zusätzlich wird der erste und der letzte Punkt in der Kontrollpunktliste durch eine Linie miteinander verbunden. Der von diesem Polygonzug eingeschlossene Bereich wird aus dem Parameterbereich ausgeschnitten. Wir gehen dabei davon aus, daß der Polygonzug immer im Uhrzeigersinn durchlaufen wird und alles, was rechts vom Polygonzug liegt, ausgeschnitten wird. Einen Testalgorithmus, mit dem wir bestimmen können, ob ein Punkt im Parameterbereich ausgeschnitten wird, werden wir weiter unten vorstellen.

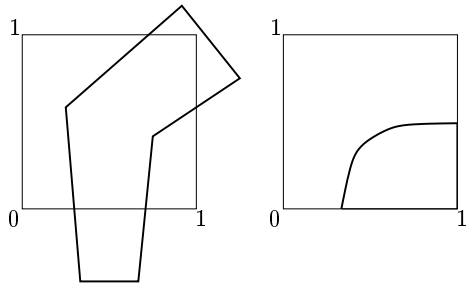


Abbildung 3.3: Trimmkurven

Geben die Kontrollpunkte eine quadratische Splinekurve an, dann gibt es pro Kontrollpunkt p jeweils ein quadratisches Bézierkurvensegment. Die für dieses Segment notwendigen drei Bézierpunkte werden aus dem Kontrollpunkt p und dem Kontrollpunkt vor und nach p über die Bedingungen der geometrischen Stetigkeit berechnet. Ist der erste und letzte Kontrollpunkt identisch, dann ist die Kurve geschlossen. Ist die Kurve nicht geschlossen, dann erweitern wir die Kurve zunächst so, daß Anfangs- und Endpunkt sicher außerhalb des Parameterbereiches liegen und fügen Teile des Randes von $[0,1]^2$ hinzu, damit eine geschlossene Kurve entsteht. Die so erzeugte quadratische Kurve wandeln wir in einen Polygonzug um. Der eigentliche Testalgorithmus ist dann mit dem für die Polygonzüge identisch. Allgemein kann man natürlich eine Splinekurve von beliebigem Grad in einen Polygonzug umwandeln.

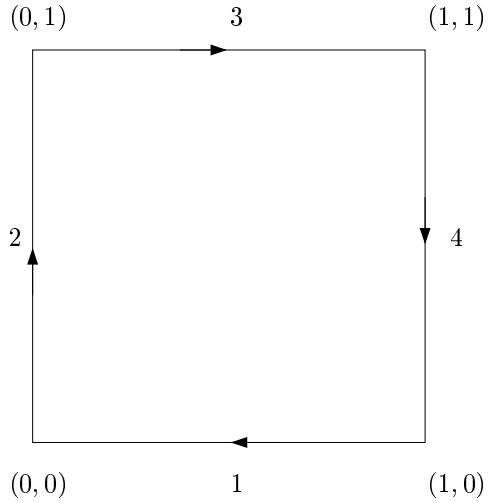


Abbildung 3.4: Erweiterung der Trimmkurven

Abbildung 3.3 zeigt zwei Beispiele von Trimmkurven im Parameterbereich. Wir berücksichtigen dabei noch nicht den Umlaufsinn und mehrere Trimmkurven in einem Parameterbereich. Die linke Trimmkurve ist ein Polygonzug. Dieser kann auch außerhalb des Parameterbereiches fortgesetzt werden. Dies entspricht einer linearen Trimmkurve im Parameterbereich, die entsprechende Teile des Randes von $[0,1]$ enthält. Die rechte Trimmkurve wird durch eine Bézierkurve bestimmt. Sie ist nicht geschlossen und wird durch den Rand von $[0,1]^2$ erweitert. Wie dies genau geschieht wird im Algorithmus zur Umwandlung in einen Polygonzug beschrieben. Liegt der Anfangs- oder Endpunkt innerhalb des Parameterbereiches, verlängern wir zunächst die Kurve in Richtung der Tangente an den entsprechenden Punkt, bis wir den Parameterbereich verlassen haben. Danach verbinden wir den Eintritts- und Austrittspunkt der Kurve aus dem Parameterbereich über den Rand. Wir gehen dabei im Uhrzeigersinn vom Austrittspunkt zum Eintrittspunkt.

Die Umwandlung dieser Kurven in Polygonzüge und die Erweiterung durch den Rand des Parameterbereiches wird durch den Algorithmus `process_quadratic_trimcurve` behandelt. Er wandelt eine zwei–dimensionale quadratische Splinekurve mit `n` Kontrollpunkten `b[0]`, ..., `b[n-1]` in einen Polygonzug $(\texttt{x[0]}, \texttt{y[0]})^T, \ldots, (\texttt{x[N-1]}, \texttt{y[N-1]})^T$ um. Dabei ist zu beachten, daß wir aus den Kontrollpunkten zunächst die Bézierpunkte für eine geometrisch glatte, quadratische Kurve berechnen. Für jedes quadratische Segment der Trimmkurve benötigen wir $\frac{1}{\texttt{step}} + 1$ Linien, wenn `step` die Schrittweite

zur Berechnung der quadratischen Kurve ist. Wenn wir die Kurve mit dem Rand von $[0,1]^2$ erweitern müssen, kommen noch maximal vier Ecken hinzu. Für die Punkte, an denen wir das Einheitsquadrat für die Erweiterung verlassen und eintreten, haben wir nochmal zwei zusätzliche Punkte für den Polygonzug. Insgesamt ist N also nicht größer als $\frac{\mathtt{n}}{\mathtt{step}} + \mathtt{n} + 7$.

Algorithmus 3.1. `process_quadratic_trimcurve`
Quadratische Trimmkurve in einen Polygonzug umwandeln

I. Initialisiere die Arrays `x` und y für $\frac{\mathtt{n}}{\mathtt{step}} + \mathtt{n} + 7$ Punkte des Polygonzuges.

II. Berechne für jeweils drei aufeinanderfolgende Kontrollpunkte `b0`, `b2`, `b4` der Trimmkurve den Polygonzug:

   A. Die Bézierpunkte der quadratischen Kurve sind `b1`, `b2` und `b3`, wobei `b1` das Mittel von `b0` und `b2` ist und `b3` das Mittel von `b2` und `b4`.

   B. Berechne den Polygonzug der Bézierkurve nach Definition 1.4. Dazu unterteile das Intervall $[0,1]$ mit der Schrittweite `step` und hänge die Koordinaten der neuen Punkte an die Arrays `x` und `y` an.

   C. Bestimme die nächsten drei Kontrollpunkte unter Berücksichtigung möglicher Erweiterungen durch den Rand des Parameterbereiches:

      1. Zunächst wird `b0` der Punkt `b2` und `b2` der Punkt `b4` zugewiesen. `b4` wird der nächste, noch nicht benutzte Punkt aus der Kontrollpunktliste.
      2. Existiert `b4`, dann kann die Umwandlung fortgesetzt werden.
      3. Existiert `b4` nicht, dann überprüfe, ob die Kurve geschlossen ist. Dazu muß der erste und der letzte Kontrollpunkt (also hier `b2` und der erste Punkt in der Kontrollpunktliste) gleich sein. Ist dies der Fall, dann setze `b4` auf den zweiten Kontrollpunkt in der Liste und wandle diese letzte Kurve in einen Polygonzug um. Ansonsten erweitere die Kurve mit dem Rand des Parameterbereiches, da die Kurve nicht geschlossen ist:
         i. Liegt der zuletzt berechnete Punkt noch innerhalb des Parameterbereiches, dann füge einen neuen Punkt hinzu, indem zum letzten Punkt der Vektor, der die letzte Linie des Polygonzuges repräsentiert, hinzuaddiert wird, bis der Parameterbereich verlassen wird.
         ii. Berechne den Punkt (`xout`, `yout`), an dem der Polygonzug den Parameterbereich verläßt. Dazu suche zunächst die letzte Linie im Polygonzug, bei der ein Punkt innerhalb und ein Punkt außerhalb des Parameterbereiches liegt. Danach überprüfe, welche der vier Randlinien einen Schnittpunkt mit dieser Linie hat. Berechne den Schnittpunkt und setze `lout` auf die Nummer der Linie (vgl. Abbildung 3.4).
         iii. Erzeuge den Eintrittspunkt (`xin` , `yin`) auf der Randlinie `lin` analog zu (`xout`, `yout`) und `lout`.
         iv. Sind `lin` und `lout` gleich und liegt (`xin` , `yin`) vor (`xout` , `yout`) auf dem Rand, wenn dieser im Uhrzeigersinn durchlaufen wird, dann verbinde beide Punkte. In diesem Fall ist der Polygonzug nun vollständig.
         v. Sind `lin` und `lout` gleich und ist der Umlaufsinn gegen den Uhrzeigersinn, dann verschiebe `lout` um eine Linie im Uhrzeigersinn und füge den übersprungenen Eckpunkt zum Polygonzug hinzu. Der Polygonzug ist aber noch nicht vollständig, sondern muß im folgenden Punkt erweitert werden.
         vi. Solange `lin` nicht `lout` ist, verschiebe `lout` um eine Linie im Uhrzeigersinn und füge den übersprungenen Eckpunkt zum Polygonzug hinzu.

Nun fehlt noch der oben angesprochene Testalgorithmus für einen Polygonzug. Diese Testfunktion muß überprüfen, ob ein vorgegebener Punkt (`u`, `v`) im Parameterbereich durch die Trimmkurve ausgeschnitten wird. Die Trimmkurve ist dabei als Polygonzug über die beiden Arrays `x` und `y` gegeben.

Der Umlaufsinn, mit dem die Trimmkurve durchlaufen wird, gibt normalerweise an, welcher Bereich innerhalb des Trimmbereiches liegt und welcher außerhalb. Wir nehmen im folgenden immer an, daß der Polygonzug im Uhrzeigersinn durchlaufen wird und alles, was rechts von dem Polygonzug liegt, ausgeschnitten wird. Gebiete mit entgegengesetztem Umlaufsinn können durch die Kombination mehrerer Trimmkurven erzeugt werden. Wir werden dies weiter unten genauer betrachten.

Zum Testen, ob ein Punkt im Trimmbereich liegt, zählen wir im Algorithmus `trimcurve_test`, wie oft eine Linie von $(\mathtt{u},\mathtt{v})$ in Richtung $(1,0)$ den Polygonzug schneidet. Ist die Anzahl der Schnitte gerade, sind wir außerhalb des Trimmbereiches, sonst innerhalb. Die Richtung ist dabei beliebig. Ein paar Beispiele hierfür zeigt Abbildung 3.5. Ist der Punkt innerhalb des Trimmbereiches, muß die Linie wenigstens einmal den Rand schneiden. Tritt die Linie an einer anderen Stelle wieder in den Trimmbereich ein, dann muß sie diesen auch wieder verlassen, da er beschränkt ist. Damit ist die Anzahl der Schnitte ungerade. Liegt der Punkt dagegen außerhalb des Trimmbereichen, dann gibt es für jeden Eintrittspunkt in den Trimmbereich auch wieder einen Austrittspunkt und so ist die Anzahl der Schnitte gerade. Es gibt allerdings Fälle, wie für den nicht ausgefüllten Punkt in Abbildung 3.5, bei denen dieses Verfahren nicht funktioniert. Durch zusätzliche Tests, etwa das Zählen der Schnitte mit Linien in andere Richtungen, lassen sich solche Fälle weitere reduzieren, wir gehen hierauf aber nicht näher ein. Für unsere Zwecke genügt die einfache Variante, wobei es manchmal nötig sein kann die Auflösung der Testgitter und der Trimmkurven anzupassen.

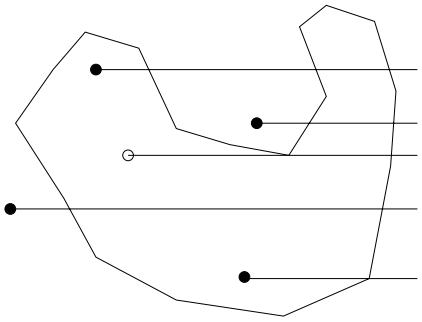


Abbildung 3.5: Trimmkurventest

Algorithmus 3.2. `trimcurve_test`
Testfunktion für eine lineare Trimmkurve

I. Bestimme die Anzahl der Segmente des Polygonzuges aus `x`, `y`, die den Strahl $(\mathtt{u},\mathtt{v})+t(1,0)$, $t>0$ schneiden. Überprüfe dazu für jede Linie des Polygonzuges Folgendes:

   A. Ein Schnittpunkt mit der Linie von (x_1,y_1) nach (x_2,y_2) liegt genau dann vor, wenn folgende Bedingung erfüllt ist (vgl. Abbildung 3.6):

$$\begin{aligned} \mathtt{v} &> \min\{y_1,y_2\} && \wedge \\ \mathtt{v} &\le \max\{y_1,y_2\} + \mathtt{Epsilon} && \wedge \\ \mathtt{u} &\le \max\{x_1,x_2\} + \mathtt{Epsilon} && \wedge \\ |y_1-y_2| &\ge \mathtt{Epsilon} && \wedge \\ \Big(|x_1-x_2| < \mathtt{Epsilon} \vee \mathtt{u} &< (x_2-x_1)\frac{\mathtt{v}-y_1}{y_2-y_1}+x_1\Big). \end{aligned} \tag{3.2}$$

   `Epsilon` gleicht dabei numerische Fehler aus. Obige Bedingung wurde so gewählt, daß ein Schnittpunkt der direkt auf einem der Endpunkte der Linien liegt, nur einmal gezählt wird. Liegt ein Liniensegment vollständig auf dem Strahl, wird es nicht berücksichtigt.

II. Ist die Anzahl der Schnitte gerade, dann liegt $(\mathtt{u},\mathtt{v})$ außerhalb des Trimmbereiches, sonst liegt er im Trimmbereich.

Wir beschreiben nun den Algorithmus zum Zeichnen getrimmter G–Spline–Flächen. Die Umwandlung in Bézierkontrollnetze ändert sich dabei nicht. Die impliziten Funktionen zum Trimmen im Raum der Fläche geben wir allgemein für die Fläche an. Für die Trimmkurven müssen wir dagegen auch das Bézierflächenstück festlegen. Nachdem wir nur biquadratische Flächenstücke haben, genügt es für die Trimmkurven einen G–Spline–Kontrollpunkt anzugeben. Dieser bestimmt dann das Flächenstück. Im Algorithmus zeichnen wir jedes Bézierflächenstück einzeln. Wir verwenden dazu die oben vorgestellten

Testfunktionen. Zusätzlich müssen wir allerdings berücksichtigen, daß mehrere Trimmkurven und implizite Funktionen für ein Flächenstück angegeben werden können. Dabei gehen wir davon aus, daß implizite Funktionen immer Teile aus der Fläche ausschneiden, auch wenn die Volumina sich überlappen.

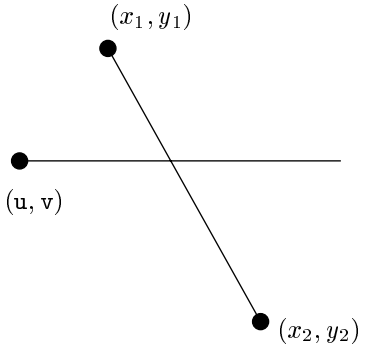


Abbildung 3.6: Schnittbedingung (3.2)

Bei Trimmkurven können sich dagegen auch "Inseln" bilden. Liegt ein Punkt im Parameterbereich innerhalb einer geraden Anzahl von Trimmbereichen, dann wird er nicht ausgeschnitten. Liegt er in einer ungeraden Anzahl von Trimmbereichen wird er ausgeschnitten (vgl. Abbildung 3.8). Damit entstehen keine Probleme, wenn sich die Trimmkurven schneiden oder der Umlaufsinn der Trimmkurven nicht zusammenpaßt. Eigentlich ignorieren wir den Umlaufsinn einfach. Deshalb nehmen wir auch wie oben beschrieben immer an, daß die Trimmkurven im Uhrzeigersinn durchlaufen werden. Trimmkurven, die gegen den Uhrzeigersinn durchlaufen werden, lassen sich trotzdem modellieren. Soll z.B. alles, was außerhalb eines Kreises im Parameterbereich liegt, weggeschnitten werden, genügt es den Rand von $[0,1]^2$ zusätzlich zum Kreis als Trimmkurve anzugeben.

Algorithmus 3.3. `trimmed_bezier`
Zeichnen eines getrimmten Bézierflächenstückes

I. Lege über das Intervall $[\mathtt{u0}, \mathtt{u1}] \times [\mathtt{v0}, \mathtt{v1}]$ ein Gitter mit $\mathtt{resolution}^2$ Punkten und prüfe, ob die Gitterknoten innerhalb oder außerhalb der Trimmbereiche liegen:

   A. Initialisiere ein zwei–dimensionales `boolean` Array `b0` für die $\mathtt{resolution}^2$ Gitterpunkte mit `false`. Es wird `true` für ausgeschnittene und `false` für die anderen Gitterpunkte sein.

   B. Prüfe für jeden Gitterpunkt mit den Koordinaten $(\mathtt{u}, \mathtt{v}) = \left(\mathtt{u0} + \mathtt{ui} * \frac{\mathtt{u1}-\mathtt{u0}}{\mathtt{resolution}-1}\right.$, $\mathtt{v0} + \left.\mathtt{vi} * \frac{\mathtt{v1}-\mathtt{v0}}{\mathtt{resolution}-1}\right)$, wobei `ui` und `vi` die Position im `b0` Array festlegen und jeweils von 0 bis $\mathtt{resolution} - 1$ hochgezählt werden, ob sie in einem Trimmbereich liegen:

      1. Zähle die Anzahl der Trimmbereiche, die die Koordinaten $(\mathtt{u}, \mathtt{v})$ des Knotens enthalten, mit Hilfe des Algorithmus' `trimmcurve_test`. Die zu dem Bézierflächenstück gehörenden Trimmkurven müssen vorher bekannt sein.
      2. Ist die Anzahl ungerade, dann setze `b0` für diesen Gitterpunkt auf `true`.
      3. Überprüfe, ob der zu $(\mathtt{u}, \mathtt{v})$ gehörende Punkt auf dem Bézierflächenstück wenigstens eine der impliziten Trimmfunktionen erfüllt. Ist dies der Fall, setze `b0` für diesen Gitterpunkt auf `true`.

II. Bearbeite nun jedes Quadrat im Parameterbereich, das durch die zwei Gitterknoten mit den Indizes `[ui,vi]` und `[ui+1,vi+1]` bzgl. `b0` bestimmt ist, um das zugehörige Teil der Fläche zu zeichnen:

   A. Wenn wenigstens eine der Ecken des Quadrats laut `b0` in einem Trimmbereich liegt:

      1. Ist die maximale Rekursionstiefe für den Trimmalgorithmus noch nicht erreicht, rufe `trimmed_bezier` rekursiv für dieses Quadrat mit der um 1 erhöhten Rekursionstiefe auf.
      2. Sonst überprüfe, ob eines der durch die Ecken des Quadrats bestimmten Dreiecke außerhalb des Trimmbereiches liegt und zeichne das zugehörige Teil der Fläche als Dreieck.

   B. Liegt keine der Ecken in einem Trimmbereich, dann zeichne den zu diesem Quadrat gehörenden Teil der Fläche durch zwei Dreiecke.

`trimmed_bezier` wird mit einer Liste der Trimmkurven aufgerufen, die zu dem jeweiligen Bézierflächenstück gehören. Diese Liste kann über den zu jeder Trimmkurve angegebenen Kontrollpunkt der

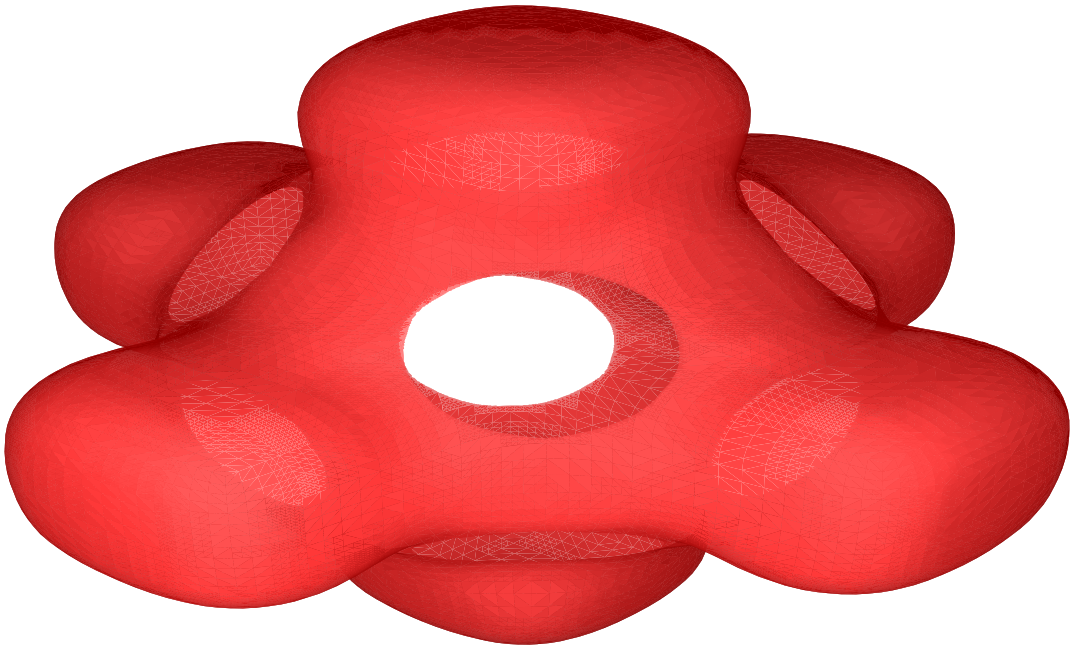


Abbildung 3.7: Getrimmtes Kreuz

G–Spline–Fläche bestimmt werden. Beim ersten Aufruf wählen wir normalerweise das Einheitsquadrat als Parameterbereich und die Rekursionstiefe ist 0. Das Trimming wird im Algorithmus vor allem durch I. implementiert. Dieser Teil kann in ähnlicher Form auch in den anderen Algorithmen für getrimmte Flächen verwendet werden.

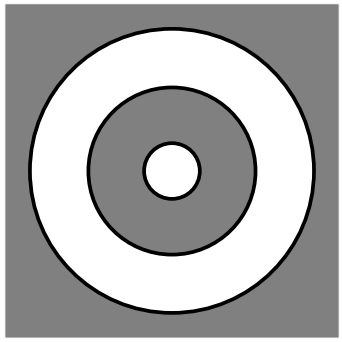


Abbildung 3.8: Überlappende Trimmbereiche

Bei `trimmed_bezier` ist zu berücksichtigen, daß die Auflösung für das Testgitter groß genug gewählt wird. Ansonsten ist es möglich, daß kleine Trimmbereiche ausgelassen werden. Auf der anderen Seite sollte aber vor allem die maximale Rekursionstiefe klein sein, da sonst der Algorithmus am Rand der Trimmbereiche sehr lange arbeitet. Die maximale Rekursionstiefe sollte der benötigten Grafikauflösung für die Darstellung angepaßt sein.

Die Trimmkurven können wir nicht für Flächen verwenden, auf die der Doo–Sabin–Subdivision–Algorithmus angewendet wurde. In unserer Implementierung des Doo–Sabin–Subdivision–Algorithmus' wird das alte Kontrollnetz vollständig durch ein neues ersetzt. Da die Trimmkurven an die alten Kontrollpunkte gebunden sind, verlieren wir hierbei alle Trimmkurven. Wir gehen in dieser Arbeit nicht näher auf dieses Problem ein, es wäre allerdings sinnvoll eine Übertragung der Trimmkurven auf die neuen Kontrollpunkte zu entwickeln. In diesem Zusammenhang sollte man auch eine Methode einführen, Trimmkurven über mehrere Parameterbereiche anzugeben. Implizite Funktionen können dagegen auch für Flächen zusammen mit der Doo–Sabin–Subdivision verwendet werden.

Abbildung 3.7 zeigt das Kreuz aus Abbildung 1.23, aus dem über eine implizite Funktion eine Kugel ausgeschnitten wurde. In Abbildung 3.9 zeigen wir den Würfel aus Abbildung 1.19, bei dem drei zu den Koordinatenachsen parallele Zylinder ausgeschnitten wurden.

Abbildung 3.10 stellt eine G–Spline–Fläche mit einer Irregularität der Ordnung 5 dar, bei der jeder Kontrollpunkt an der Irregularität mit jeweils drei quadratischen, nicht geschlossenen Trimmkurven und einem Trimmpolygon versehen wurde.

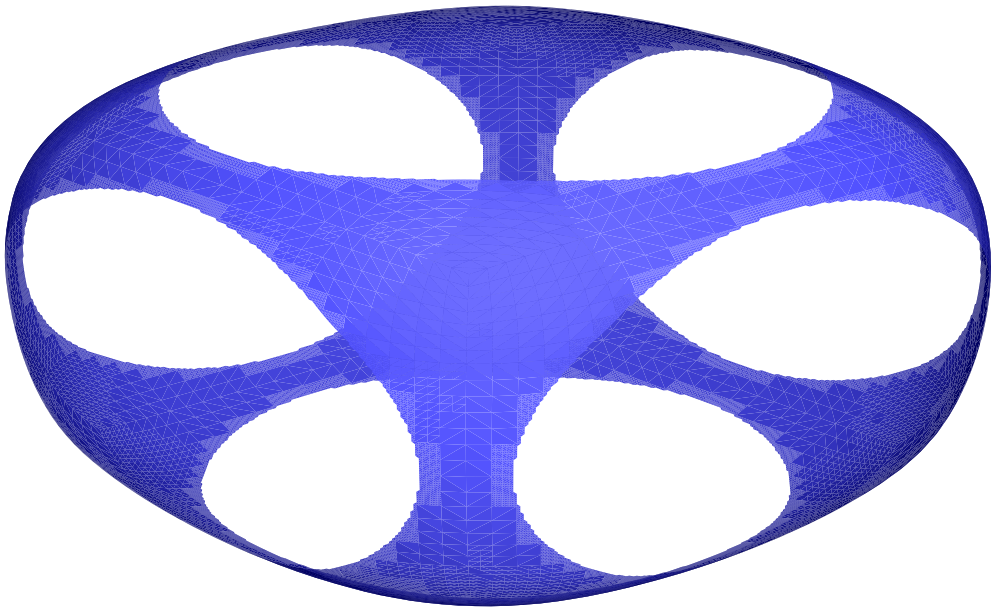


Abbildung 3.9: Getrimmter Würfel

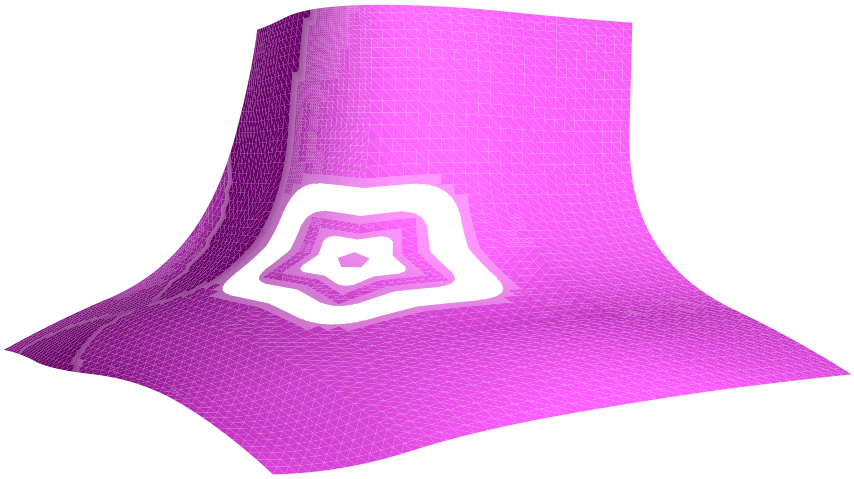


Abbildung 3.10: Getrimmte G–Spline–Fläche mit einer Irregularität der Ordnung 5

3.2 Funktionen auf getrimmten Flächen

Auch andere G–Spline–Algorithmen lassen sich nach dem Beispiel von `trimmed_bezier` für getrimmte Flächen modifizieren. Dabei bleiben die Tests gleich und in der nachfolgenden Bearbeitung können wir den Algorithmus für nicht getrimmte Flächen so modifizieren, daß er die Testergebnisse berücksichtigt. Wir betrachten dies im folgenden für die Darstellung und Integration von Funktionen auf getrimmten Flächen.

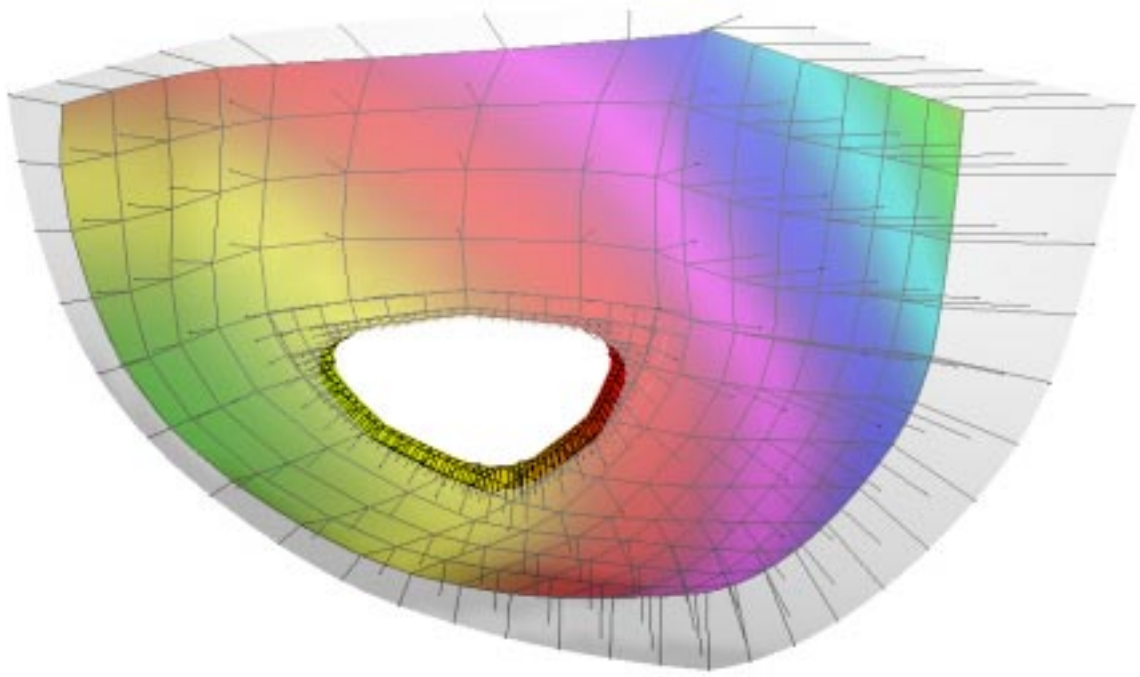


Abbildung 3.11: Funktion auf einer getrimmten Fläche mit einer Irregularität der Ordnung 3

3.2.1 Darstellung getrimmter Funktionen

Der Algorithmus zur Darstellung von Funktionen auf getrimmten Flächen entspricht im Prinzip dem zur Darstellung der Fläche selbst. Nur zeichnen wir anstatt der Fläche die Funktion in Abhängigkeit von den ausgewählten Darstellungsarten. Pro Punkt der Fläche müssen wir zusätzlich überprüfen, ob er durch eine Trimmkurve im Parameterbereich oder durch eine implizite Funktion im $\mathbb{R}^3$ ausgeschnitten wird.

Bei Isolinien gehen wir nicht ganz so direkt vor. Für Quadrate im Parameterbereich der Fläche, die in keinem Trimmbereich liegen, können wir den Algorithmus `isolines` mit entsprechenden Argumenten aufrufen. Wir verwenden also die Tests aus dem Algorithmus `trimmed_bezier`, um den Parameterbereich in Quadrate zu zerlegen, die entweder im Trimmbereich oder außerhalb des Trimmbereiches liegen und rufen dann entsprechend `isolines` auf.

Algorithmus 3.4. `trimmed_isolines`
Zeichnen von Isolinien auf einem getrimmten Bézierflächenstück

I. Lege über das Intervall $[\texttt{u0}, \texttt{u1}] \times [\texttt{v0}, \texttt{v1}]$ ein Gitter mit $\texttt{resolution}^2$ Punkten und prüfe, ob die Gitterknoten innerhalb oder außerhalb der Trimmbereiche liegen:

   A. Initialisiere ein zwei–dimensionales `boolean` Array `b0` für die $\texttt{resolution}^2$ Gitterpunkte mit `false`. Es wird `true` für ausgeschnittene und `false` für die anderen Gitterpunkte sein.

   B. Prüfe für jeden Gitterpunkt mit den Koordinaten $(\texttt{u}, \texttt{v}) = \left(\texttt{u0} + \texttt{ui} * \frac{\texttt{u1}-\texttt{u0}}{\texttt{resolution}-1}\ ,\ \texttt{v0} + \texttt{vi} * \frac{\texttt{v1}-\texttt{v0}}{\texttt{resolution}-1}\right)$, wobei `ui` und `vi` die Position im `b0` Array festlegen und jeweils von 0 bis $\texttt{resolution} - 1$ hochgezählt werden, ob sie in einem Trimmbereich liegen:

      1. Zähle die Anzahl der Trimmbereiche, die die Koordinaten $(\texttt{u}, \texttt{v})$ des Knotens enthalten, mit Hilfe des Algorithmus' `trimmcurve_test`. Die zu dem Bézierflächenstück gehörenden Trimmkurven müssen vorher bekannt sein.
      2. Ist die Anzahl ungerade, dann setze `b0` für diesen Gitterpunkt auf `true`.

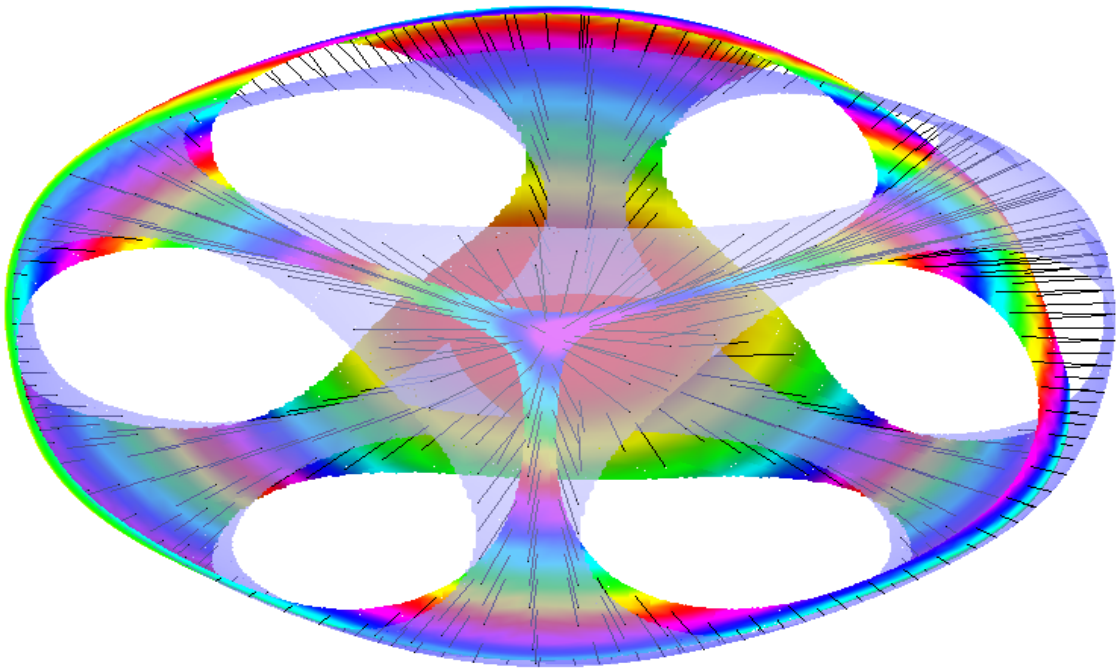


Abbildung 3.12: Funktion auf dem getrimmten Würfel

3. Überprüfe, ob der zu (u, v) gehörende Punkt auf dem Bézierflächenstück wenigstens eine der impliziten Trimmfunktionen erfüllt. Ist dies der Fall, setze `b0` für diesen Gitterpunkt auf `true`.

II. Bearbeite jedes Quadrat im Parameterbereich, das durch die zwei Gitterknoten mit den Indizes `[ui,vi]` und `[ui+1,vi+1]` bzgl. `b0` bestimmt ist, um die zugehörigen Isolinien zu zeichnen:

A. Liegt wenigstens eine der Ecken des Quadrats laut `b0` in einem Trimmbereich, dann rufe `trimmed_isolines` rekursiv für dieses Quadrat auf. Falls dies die maximale Rekursionstiefe überschreiten würde, tue nichts.

B. Liegt keine der Ecken in einem Trimmbereich, dann rufe `isolines` für dieses Quadrat auf, um die Isolinien zu zeichnen.

`trimmed_isolines` wird mit einer Liste der Trimmkurven aufgerufen, die zu dem jeweiligen Bézierflächenstück gehören. Beim ersten Aufruf wählen wir normalerweise das Einheitsquadrat als Parameterbereich und die Rekursionstiefe ist 0. Der erste Teil entspricht natürlich genau dem schon bekannten Testalgorithmus für das Trimming. Nur im zweiten Teil wird `isolines` mit den entsprechenden Parametern aufgerufen, um die Isolinien zu zeichnen. Obiger Algorithmus führt nur die Unterteilung des Parameterbereiches durch, bis ein gesamtes Quadrat außerhalb oder innerhalb der Trimmbereiche liegt und verwendet dann den Algorithmus für nicht getrimmte Flächen, der mit einer entsprechend großen Rekursionstiefe aufgerufen wird.

Sobald eines der Quadrate vollständig außerhalb des Trimmbereiches liegt, werden über `isolines` die zugehörigen Isolinien gezeichnet. Liegt ein Quadrat teilweise in einem Trimmbereich, dann wird der Algorithmus rekursiv aufgerufen. Wurde im Algorithmus `trimmed_isolines` die maximale Rekursionstiefe erreicht, werden die Quadrate, bei denen wenigstens ein Punkt des Parameterbereiches im Trimmbereich liegt, nicht weiter behandelt. Dabei verlieren wir die Isolinien auf den Dreiecken am Rand der Trimmbereiche. Bei feinen Auflösungen ist dies allerdings zu vernachlässigen. Ansonsten könnte man hier noch eine spezielle Version von `handle_quad` verwenden, die nur ein Dreieck berücksichtigt.

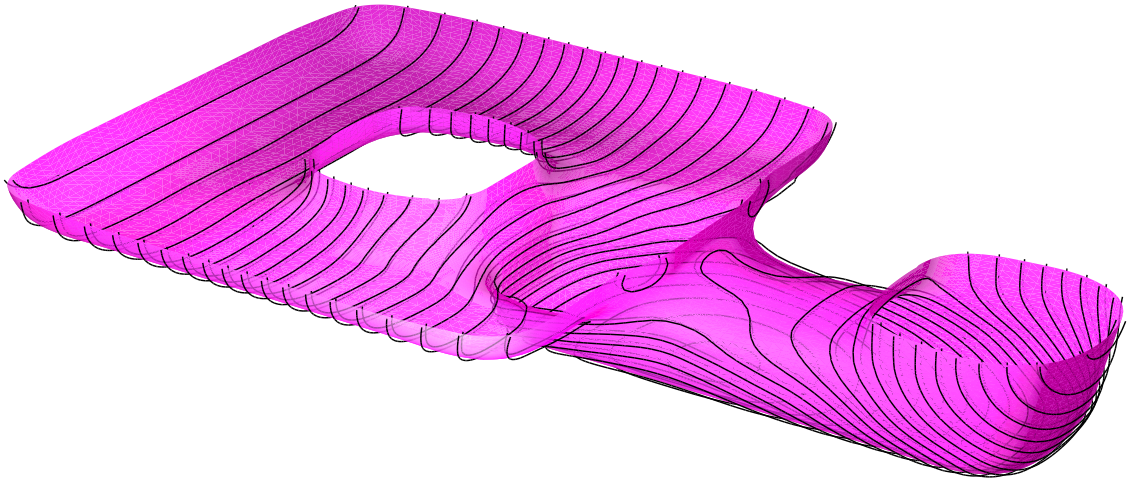


Abbildung 3.13: Isolinien auf der aufgeschnittenen, verdrehten Acht

Abbildung 3.11 zeigt eine G–Spline–Fläche mit einer Irregularität der Ordnung 3. An den drei Kontrollpunkten dieser Irregularität wurde ein kreisähnlicher Trimmbereich über quadratische Trimmkurven ausgeschnitten. Die Fläche selbst ist transparent und hinter ihr stellen wir eine Funktion auf dieser Fläche durch eine farbige Fläche, Stacheln und ein Gitter dar. In Abbildung 3.12 wird eine Funktion auf dem transparenten getrimmten Würfel auf die gleiche Weise dargestellt.

In Abbildung 3.13 ist die verdrehte Acht mit Isolinien aus der Abbildung 2.19 zu sehen. Wir haben sie durch eine implizite Funktion aufgeschnitten. Die Isolinien wurden sowohl etwas oberhalb, als auch etwas unterhalb der Fläche gezeichnet.

3.2.2 Integration getrimmter Funktionen

Eine einfache Möglichkeit, eine Funktion über einer getrimmten Fläche zu integrieren, besteht darin, sie auf den Trimmbereichen auf 0 zu setzen und dann den normalen Romberg–Algorithmus anzuwenden. Wir müssen hier nur die Funktionsberechnung um die Tests aus Algorithmus `trimmed_bezier` erweitern. Für die Berechnung des Flächeninhaltes können wir genauso vorgehen. Volumenberechnungen für getrimmte Körper sind nicht möglich. Die Methode funktioniert sowohl für Trimmkurven als auch für implizite Funktionen. Allerdings ist die so entstandene Funktion nicht mehr differenzierbar und damit ist Theorem 2.3 nicht mehr anwendbar. Dies bedeutet, daß der Romberg–Algorithmus eigentlich nicht mehr angewendet werden sollte und wir eine wesentlich schlechtere Konvergenzordnung der Verfahrens erhalten (vgl. [Höl98]). Im Rahmen dieser Arbeit beschränken wir uns allerdings auf diese einfache Variation des Romberg–Algorithmus'.

In Abbildung 3.14 berechnen wir die gleichen Integrale wie in Abbildung 2.11 bis auf das Volumen für den getrimmten Würfel aus Abbildung 3.9 und für das getrimmte Kreuz aus Abbildung 3.7. Die mittlere Anzahl der Iterationen für die Schwerpunktintegrale ist beim getrimmten Würfel kleiner als beim einfachen Würfel. Dies liegt an den großen Bereichen, die wir ausgeschnitten haben und für die dann nur noch eine Iteration notwendig ist. Bei allen anderen Integralen benötigen wir jedoch mehr Iterationen.

In [Höl98] werden zwei weitere Methoden für die Integration angegeben. So könnte man an den Rändern der Trimmbereiche die bereits berechneten Polygonzüge verwenden und dort über die entstandenen Dreiecke integrieren. Eine weitere Möglichkeit besteht darin, an den Rändern das Integrationsgebiet auf Rechtecke zu transformieren. Dazu legen wird über das gesamte Integrationsgebiet ein Rechteckgitter, so daß sich an den Rändern die Kurven als Funktion von u oder v darstellen lassen.

Objekt Abbildung	Flächenstücke `prec`	Fläche [Iter.]	Schwerpunktintegrale x [Iter.] y [Iter.] z [Iter.]	Schwerpkt. x y z
einfacher Würfel 3.9	96 1.04167e-05	1.22 [3.875]	0.61 [2.9583] 0.61 [2.9583] 0.61 [2.9583]	0.5 0.5 0.5
Kreuz 3.7	480 1.72414e-06	17.71 [4.225]	26.56 [4.6625] 26.56 [4.6417] 26.56 [4.6208]	1.50 1.50 1.50

Abbildung 3.14: Integrale und Flächenschwerpunkte verschiedener getrimmter Objekte

Dann besteht die Transformation nur in der Skalierung einer der beiden Parameter. Beide Methoden funktionieren allerdings nur für Trimmkurven und nicht für implizite Trimmfunktionen.

Kapitel 4

Implementierung

In diesem Kapitel stellen wir das Programm vor, in dem alle vorher besprochenen Algorithmen implementiert wurden. Dazu beschreiben wir zunächst den Programmablauf und gehen etwas auf die verwendeten Entwicklungswerkzeuge und Systemarchitekturen ein. Weiter erklären wir die Benutzung des Programms und das Dateiformat ODL. Schließlich stellen wir noch zwei spezielle Dateiformate vor, aus denen man einfach ODL–Dateien erzeugen kann.

4.1 LiLit Programm–Dokumentation

Die in dieser Arbeit vorgestellten Algorithmen haben wir in dem Programm LiLit[1] realisiert. Es handelt sich dabei um einen Prototyp, den wir zur Implementierung der wesentlichen Algorithmen verwendet haben. Es wurde dabei weniger auf grafische Benutzeroberflächen, Fehlerbehandlung und erweiterte Funktionalität Rücksicht genommen. LiLit ist größtenteils objektorientiert in ANSI C++ geschrieben worden. Aufgrund der Größe von LiLit werden wir hier nur die grobe Struktur des Programms vorstellen.

LiLit liest zunächst die zu bearbeitenden Objekte aus einer Datei im ODL–Format (siehe unten) ein. Der hierzu notwendige Parser wurde mit bison++ erstellt. Der Parser verwendet einen mit flex++ erzeugten lexikographischen Scanner. Er wandelt die ODL–Darstellung in die interne Darstellung der Objekte durch die Klasse `World` um. Diese Klasse enthält eine Liste aller in der Datei beschriebenen Objekte, eine Liste der Objekte, die angezeigt werden sollen und eine Liste der zu berechnenden Integrale. Weiter enthält `World` noch allgemeine Informationen zum Rendern und Bearbeiten der gesamten Szene. Die Objekte werden alle durch eine von der abstrakten Klasse `Object` abgeleiteten Klasse dargestellt.

Im nächsten Schritt bearbeitet LiLit allgemein alle Objekte, die angezeigt bzw. integriert werden sollen. Zusätzlich müssen wir auch Objekte bearbeiten, die indirekt von diesen Objekten benutzt werden. In diesem Schritt werden die G–Spline–Kontrollnetze zusammen mit eventuell vorhandenen Funktionskontrollnetzen in Bézierkontrollnetze umgewandelt. Als Ergebnis erhalten wir zu den Objekten eine einfachere Beschreibung, die zum Integrieren und Rendern verwendet werden kann.

Sollen Oberflächenintegrale berechnet werden, werden anschließend die zugehörigen Integrationsmethoden aufgerufen. Die Ergebnisse werden so gespeichert, daß sie im folgenden Schritt einfach angezeigt werden können.

[1] LiLit wurde von dem hebräischen **לילית** für "die Nächtliche" abgeleitet, welches wohl vom akkadischen *lilitu* für "dämonischer Wind" abstammt. Bei den Akkadern war Lilit eine Dämonin der Nacht, der Ursprung der Succubi. Manchmal gilt sie auch als erste Frau des Adams. Nach der Übersetzung von Samuel Kramer ist die älteste, bekannte Referenz in der Gilgamesch Dichtung zu finden.

Schließlich werden alle Objekte, die angezeigt werden sollen, mit Hilfe von OpenGL und GLUT gerendert. Für die Bézierkontrollnetze der Flächen können wir die Standardfunktionen verwenden. Zur Darstellung der Funktionen auf den Flächen müssen wir die Bézierkontrollnetze zunächst entsprechend auswerten und die Normalen, Flächen, etc. durch OpenGL–Primitives rendern. Bei Isolinien werden die Linien sogar erst jetzt berechnet. Wir speichern die OpenGL–Primitives in Display–Listen, um die Szene schnell neu anzeigen zu können. Nachdem alle Primitives erzeugt wurden, kann der Benutzer im interaktiven Modus von LiLit den Beobachtungspunkt verändern.

Neben dem direkten Anzeigen der Objekte in einem X11–Fenster kann man die Szene auch in einer PostScript–Datei speichern. Wir benutzen hierzu den Feedback–Buffer von OpenGL. Die grundlegende Idee hierfür stammt aus der GLP–Bibliothek von Michael Sweet. OpenGL transformiert zunächst die drei–dimensionalen Grafikobjekte in zwei–dimensionale Objekte. Erst im nächsten Schritt werden diese Objekte für die Grafikausgabe gerastert. Wir können OpenGL nun dazu veranlassen, nur die zwei–dimensionalen Primitives aus dem ersten Schritt zu erzeugen und diese im Feedback–Buffer abzulegen. Der Buffer enthält nur Punkte, Linien und Polygone. Er kann auch noch Bitmaps enthalten, die wir aber für LiLit nicht verwenden und damit ignorieren können. Die Punkte und Linien lassen sich direkt in PostScript übertragen. Die Polygone wandeln wir in Dreiecke um und zeichnen diese Dreiecke mit PostScript. Neben den zwei–dimensionalen Koordinaten hat jeder Punkt im Feedback–Buffer auch noch eine Z–Buffer–Koordinate. Mit Hilfe dieser Koordinaten können wir die Punkte, Linien und Dreiecke sortieren und die am weitesten hinten liegenden Primitives zuerst zeichnen. Dies ergibt einen einfachen Überdeckungsalgorithmus, der bei kleinen Primitives durchaus ausreicht. Die so erzeugte PostScript–Datei kann dann frei skaliert werden. Bei sehr vielen Primitives wird sie aber sehr schnell sehr groß. In diesen Fällen hilft es aber, die Auflösung der drei–dimensionalen Objekte zu verringern.

LiLit wurde für UNIX–ähnliche Systeme entwickelt. Die Entwicklungsplattform war ein erweitertes Debian/GNU 2.0 Linux System[2]. Linux ist nicht nur eine sehr stabile Plattform, sondern eignet sich auch ideal als Entwicklungsumgebung wegen der OpenSource[3] Philosophie und dem Free Software Konzept von GNU[4]. Aus diesem Grund wird auch LiLit selbst als freie Software unter der GNU General Public License vertrieben.

Die wesentlichen Programmteile wurden in ANSI C++ geschrieben. Um die Kompatibilität zwischen den UNIX–Varianten zu sichern, wurden manche Funktionen, die nicht in jeder libc Implementation gleich sind, in C geschrieben. Weiter wurde GNU autoconf[5] 2.12 zur Konfiguration des Quellcodes für die verschiedenen Systeme verwendet. Damit ist LiLit einfach auf viele andere UNIX–Varianten zu portieren.

Das Programm wurde mit der erweiterten GNU C Compiler Suite, egcs[6] Version 1.0.3 und 1.1 übersetzt. Weiter wurde die C++ Version des GNU Fast Lexical Analyzer Generators flex++ Version 2.5.4 und bison++ Version 1.21-7, welches von dem GNU Project Parser Generator bison[7] abgeleitet wurde, verwendet. Neben den Standard UNIX und X11R6 Bibliotheken wird auch noch LaPack[8] mit BLAS benötigt. Als Grafikbibliothek muß eine zum OpenGL Standard kompatible Bibliothek mit GLUT vorhanden sein. Ausführlich wurde dies aber nur für die freie OpenGL Implementierung Mesa[9] ab Version 3 getestet.

Der Quellcode von LiLit wurde mit dem Dokumentationssystem DOC++[10] kommentiert. Diese Dokumentation ist im Anhang A zu finden.

Wie schon erwähnt, handelt es sich bei LiLit nur um einen Prototyp, der allerdings den in dieser Arbeit benötigten Funktionsumfang schon vollständig enthält. Abbildung 4.1 zeigt eine verbesserte

[2] URI: `http://www.debian.org/`
[3] URI: `http://www.opensource.org/`
[4] URI: `http://www.gnu.org/`
[5] URI: `http://www.gnu.org/software/autoconf/autoconf.html`
[6] URI: `http://egcs.cygnus.com/`
[7] URI: `http://www.gnu.org/software/bison/bison.html`
[8] URI: `http://www.netlib.org/lapack/`
[9] URI: `http://www.mesa3d.org/`
[10] URI: `http://www.zib.de/Visual/software/doc++/`

Struktur für LiLit. Die ODL–Datei wurde hier durch ein über XML[11] (eXtensible Markup Language) definiertes Format ersetzt. Das Format selbst wird über eine DTD (Document Type Definition) festgelegt und kann somit einfach erweitert und modularisiert werden. Der XML–Parser wandelt die Eingabedatei in einen XML–Baum um. Aus diesem Baum könnte über eine externe Beschreibung der vorhandenen Objekte und Methoden eine interne Repräsentation der Eingabedatei erzeugt werden. Diese Daten können dann im folgenden Schritt zu LiLit–Primitives umgewandelt werden. Hierzu können sowohl interne als auch externe, dynamisch ladbare, Methoden verwendet werden. Dabei sollten die internen Methoden nur einen minimalen Funktionsumfang zur Verfügung stellen. Alle komplexeren Objekte könnten durch die externen Methoden bearbeitet werden. Welche Methoden hier zur Verfügung stehen, ließe sich durch eine Konfigurationsdatei angeben. Neben dynamisch ladbaren Methoden könnte man zusätzlich eine Skriptsprache entwickeln, über die man schon vorhandene Methoden auf komplexere Objekte anwenden kann. Die so erzeugten LiLit–Primitives ließen sich schließlich über

[11] URI: `http://www.w3.org/XML/`

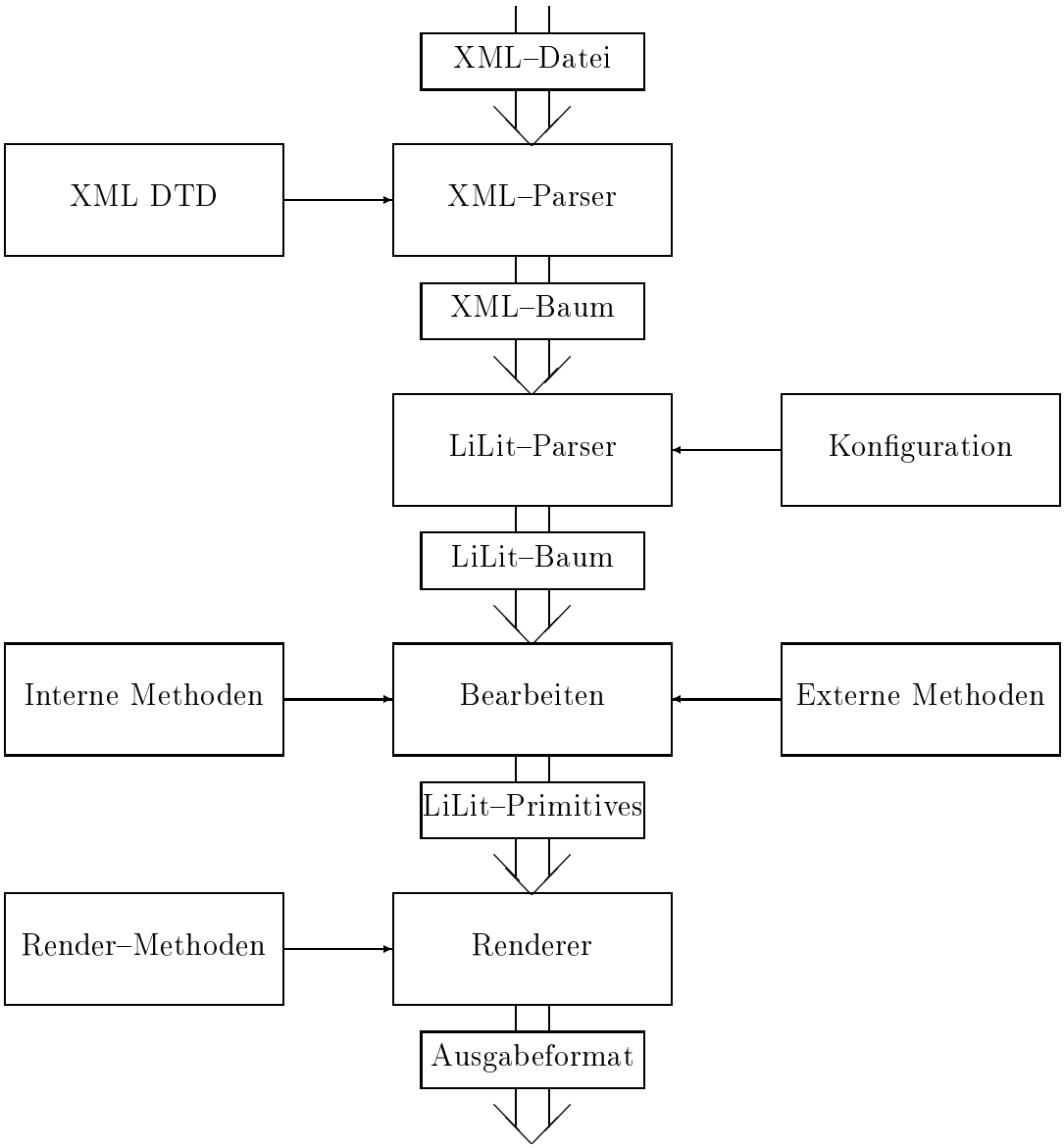


Abbildung 4.1: Verbesserte Struktur für LiLit

Render–Methoden darstellen. Dabei kann man natürlich verschiedene Ausgabeformate unterstützen.

Die momentane Version von LiLit verwendet noch ODL–Dateien und nicht XML für die Eingabe und alle Methoden zum Rendern und Bearbeiten sind intern implementiert ohne externe Konfigurationsdateien.

4.2 LiLit Benutzer–Dokumentation

Wir beschreiben nun LiLit in einem einer UNIX–Manualseite ähnlichem Format. Neben dem allgemeinen Programmaufruf stellen wir vor allem auch das ODL–Format vor.

Name

LiLit — Darstellen und Integrieren von drei–dimensionalen G–Spline–Flächen und Funktionen

Syntax

lilit [**–a** *sens*] [**–d** *display*] [**–e**] [**–f** *size*] [**–i**] [**–h**] [**–m** *sens*] [**–o**] [**–v**] [**–w** *width*] [**––**] [**––angle** *sens*] [**––display** *display*] [**––eps**] [**––gl**] [**––glps–feedsize** *size*] [**––help**] [**––iconic**] [**––info–width** *width*] [**––move** *sens*] [**––version**] *odl_file*

Beschreibung

LiLit dient im wesentlichen zum Anzeigen und Integrieren von G–Spline–Flächen und darauf definierten Funktionen.
Das Programm liest die Beschreibung für eine Szene aus einer Eingabedatei *odl_file* im ODL–Format. Diese Datei enthält eine Beschreibung von Grafikobjekten wie Punkte, Polygone, Flächen und Funktionen auf diesen Flächen. Eine Fläche wird dabei durch ein G–Spline–Kontrollnetz beschrieben, welches aus Polygonobjekten zusammengesetzt ist, die wiederum aus Punktobjekten bestehen. Die Funktionen werden über Kontrollpunkte auf den G–Spline–Kontrollnetzen beschrieben. Zusätzlich lassen sich aber auch spezielle Funktionen, wie die Krümmung der Fläche, definieren. Es gibt noch ein paar spezielle Grafikobjekte zur Darstellung der Funktion über Isolinien, etc. Die Datei enthält zusätzlich Befehle zum Anzeigen und Integrieren der Objekte.
Neben den Grafikobjekten enthält die Eingabedatei auch Informationen über Kameraposition und Lichtquellen. Die Kameraposition und Richtung kann interaktiv verändert werden. Es gibt zur Darstellung zwei Anzeigemodi. Im ersten kann die Kamera frei durch den Raum bewegt werden, im zweiten wird nur der Abstand der Kamera vom Ursprung angegeben und die Position der Objekte kann modifiziert werden.
Normalerweise zeigt LiLit die Objekte in einem X11–Fenster an. Eine Kurzbeschreibung der interaktiven Befehle und die Ergebnisse der Integrationen werden in einem zusätzlichen Informationsfenster angezeigt. Es besteht aber auch die Möglichkeit die Szene über den OpenGL–Feedback–Buffer im encapsulated PostScript–Format in einer Datei zu speichern. Der Name dieser Datei wird durch *odl_file* festgelegt, wobei ein Suffix `.odl` durch `.eps` ersetzt wird bzw. einfach `.eps` an den Dateinamen angehängt wird.

Optionen

–a *sens*, **––angle** *sens*
: Setzt die Sensitivität der Mausbewegungen zum Verändern der Kamerawinkel auf *sens*. Der default Wert ist 0.2.

–d *display*, **––display** *display*
: Setzt das X11–Display, das für die Anzeige der Fenster verwendet wird. Das default Display wird über die Umgebungsvariable **DISPLAY** festgelegt.

–e, **––eps**
: Die Objekte aus der ODL–Datei werden mit Hilfe des Feedback–Buffers von OpenGL in

eine encapsulated PostScript–Ausgabedatei geschrieben. Die Größe des Feedback–Buffers muß ausreichen, um alle OpenGL–Primitives für die Szene abzulegen.

–f *size*, **––glps–feedsize** *size*
Setzt die Größe des OpenGL–Feedback–Buffers zum Erzeugen der encapsulated PostScript–Dateien auf *size* Bytes. Sollte die Größe nicht ausreichen wird pro Ausgabeversuch die Buffergröße jeweils um diesen Wert erhöht, bis nicht mehr genügend Speicher vorhanden ist, oder ein Ausgabeversuch erfolgreich war. Der default Wert ist 1 000 000.

–h, **––help**
Druckt eine Kurzbeschreibung des LiLit Syntax' und beendet das Programm.

–i,**––iconic**
Beim Starten werden die LiLit Fenster im Icon–Modus geöffnet.

–m *sens*, **––move** *sens*
Setzt die Sensitivität der Mausbewegungen zum Verändern der Position der Kamera auf *sens*. Der default Wert ist 0.005.

–o, **––gl**
Die Objekte aus der ODL–Datei werden mit Hilfe von OpenGL bzw. MesaGL angezeigt. Dies ist die default Einstellung.

–v, **––version**
Zeigt Informationen über die LiLit Version an und beendet das Programm.

–w width, **––info–width** width
Setzt die Breite des Informationsfensters auf *width*. Der default Wert ist 350.

––
Alle Argumente nach dieser Option werden nicht mehr als Option behandelt, auch wenn sie mit – beginnen. Damit können auch Dateinamen, deren erstes Zeichen – ist, verwendet werden.

Interaktive Bedienung

Im interaktiven Modus von LiLit läßt sich die Kameraposition durch folgende Befehle verändern. Dabei wird im "Eye–Movement–Mode" die Kameraposition und im "Object–Movement–Mode" die Objektposition verändert. Zusätzlich gibt es Tasten zum Festlegen des Anzeigemodus, zum Erzeugen von encapsulated PostScript–Dateien, etc.

Allgemein

Tasten	Mausbewegung	Beschreibung
p	—	Ausgabe der momentanen Szene als encapsulated PostScript–Datei in Farbe.
P	—	Ausgabe der momentanen Szene als encapsulated PostScript–Datei in Graustufen.
q	—	Programm beenden.
e	—	"Eye–Movement–Mode" aktivieren.
o	—	"Object–Movement–Mode" aktivieren.
i	—	Anzeigeparameter zurück auf die Werte der Eingabedatei setzen.

Eye–Movement–Mode

Tasten	Mausbewegung	Beschreibung
j, k	Linke Taste + Y Bewegung	Y–Richtung der Kamera verändern.
h, l	Linke Taste + X Bewegung	X–Richtung der Kamera verändern.
s, d	Mittlere Taste + Y Bewegung	Kamera nach vorne oder hinten bewegen.
a, f	Mittlere Taste +	Kamera nach links oder rechts um die Blickrichtung

	X Bewegung	rotieren.
—	Rechte Taste + X/Y Bewegung	Objekte um die X– und Y–Achse rotieren.

Object–Movement–Mode

Tasten	Mausbewegung	Beschreibung
j, k	Linke Taste + Y Bewegung	Objekte um die X–Achse rotieren.
h, l	Linke Taste + X Bewegung	Objekte um die Y–Achse rotieren.
s, d	Mittlere Taste + Y Bewegung	Entfernung der Kamera vom Ursprung vergrößern oder verkleinern.
a, f	Mittlere Taste + X Bewegung	Objekte um die Z–Achse rotieren.
—	Rechte Taste + X/Y Bewegung	Objekte um die X– und Y–Achse rotieren.

ODL–Format

Eine ODL–Datei[12] besteht aus einer Reihe von Objektbeschreibungen und Befehlen zur Bestimmung der Anzeigeparameter. Dabei spielt die Reihenfolge keine Rolle. Allerdings müssen Objektbezeichner, die von Befehlen oder anderen Objektbeschreibungen benötigt werden, vorher eingeführt worden sein.
Eine Objektbeschreibung sieht allgemein wie folgt aus:

Objekttyp Bezeichner `{`
    Objektbeschreibung
`}`

Diese Objekte können dann mit folgenden Befehlen angezeigt und integriert werden:

`render` *Objekt*
`integrate_s` *Funktion maxiter prec*
`integrate_v` *Funktion maxiter prec*
`area` *Fläche maxiter prec*
`volume` *Fläche maxiter prec*

Mit `render` wird das Objekt gerendert. `integrate_s` ist das Oberflächenintegral für Skalarfelder, `integrate_v` das Oberflächenintegral für Vektorfelder, `area` berechnet den Flächeninhalt und `volume` das Volumen geschlossener Flächen. Die Integrale müssen für die Objekte sinnvoll sein. Die Parameter *maxiter* und *prec* geben die maximale Anzahl der Iterationen und die Genauigkeit pro Flächenstück an.
Die Kameraposition und der Anzeigemodus (`object` oder `eye`) werden durch folgenden Befehl festgelegt. Er legt gleichzeitig die Parameter für beide Anzeigemodi fest, wählt aber einen als default aus.

`viewmode` `eye`|`object`
Kamera_Azimuth Kamera_Elevation Kamera_X Kamera_Y Kamera_Z
Objekt_Azimuth Objekt_Elevation Objekt_Abstand

Die Projektionsmethode wird durch einen der folgenden Befehle festgelegt:

`orthographic` *Limit_links Limit_rechts Limit_unten Limit_oben Limit_nah Limit_fern*
`perspective` *Winkel Aspect Limit_nah Limit_fern*

Per default wird der Beobachtungspunkt der Kamera als unendlich angesehen. Mit folgendem Befehl wird ein lokaler Beobachtungspunkt eingeführt:

[12] ODL steht für "**O**bject **D**efinition **L**anguage"

`local_viewer`

Objekte können mit den Befehlen

`translate` <*x*,*y*,*z*>
`rotate` <*x*,*y*,*z*>
`scale` <*x*,*y*,*z*>

verschoben, rotiert und skaliert werden.
Das X11–Fenster wird definiert durch

`window` *Fenstertitel x y Breite Höhe*

Die Hintergrundfarbe und der Wert für die ambient Lichtquelle wird durch

`background` <*rot*,*grün*,*blau*,*alpha*>
`ambient_light` <*rot*,*grün*,*blau*,*alpha*>

gesetzt. Die Größe eines einzelnen Punktes wird gesetzt durch

`point_size` *Punktgröße*

Mit

`epsilon` *eps*

wird die Genauigkeit zum Bestimmen von Null–Werten, etc. festgelegt.
Objekte werden durch Eigenschaften genauer beschrieben. Dabei gibt es eine Reihe von Eigenschaften, die jedes Objekt unabhängig vom Typ besitzt. Andere können nur bei bestimmten Objekttypen angegeben werden. Die Objektstruktur ist dabei hierarchisch, d.h. eine Funktion enthält einen Verweis auf eine Fläche, eine Fläche enthält eine Liste von Polygonen und diese wiederum enthalten Verweise auf Punkte. Dabei ist auch zu beachten, daß für Kontrollnetze für den gleichen Punkt, das gleiche Punktobjekt benutzt werden muß. Ansonsten wird angenommen, daß es sich um zwei verschiedene Punkte handelt, auch wenn die Position gleich ist.
Unabhängig vom Objekttyp wird ein einzelnes Objekt mit den Standardeigenschaften wie folgt definiert:

Objekttyp Bezeichner {
    `location` <*x*,*y*,*z*>
    `translate` <*x*,*y*,*z*>
    `scale` <*x*,*y*,*z*>
    `rotate` <*x*,*y*,*z*>
    `ambient` <*rot*,*grün*,*blau*,*alpha*>
    `diffuse` <*rot*,*grün*,*blau*,*alpha*>
    `specular` <*rot*,*grün*,*blau*,*alpha*>
    `emission` <*rot*,*grün*,*blau*,*alpha*>
    `shininess` *s*
    `resolution` *res*
    spezielle Objekteigenschaften ...
}

`location`, `translate`, `scale` und `rotate` geben dabei die Position, die Translation, die Skalierung und die Rotation bzgl. der *x*–, *y*– und *z*–Achse an. Die Position hat natürlich bei Objekten mit Kontrollnetz, etc. keine Bedeutung, sondern wird durch die Punkte der Teilobjekte festgelegt. `ambient`, `diffuse`, `specular`, `emission` und `shininess` geben die Materialeigenschaften des Objekts an. Es wird hierzu das Lichtmodell von OpenGL verwendet. Der Alpha–Kanal wird zum Erzeugen von teilweise transparenten Materialien verwendet. `resolution` legt die Auflösung fest, mit der das Objekt gezeichnet wird. Die Auflösung gibt dabei die univariate Anzahl

der Unterteilungen der zur Berechnung benötigten Gitter an. Die Reihenfolge der Eigenschaften ist nicht wichtig. Für manche existieren default Werte, so daß sie weggelassen werden können. Weitere Eigenschaftstypen werden durch den Objekttyp festgelegt. Dabei stehen `point`, `polygon`, `gspline`, `function`, `isoline` und `light` zur Verfügung.
`point` definiert einen Punkt. Dieser hat keine weiteren Eigenschaften. Allerdings läßt sich ein Punktobjekt vereinfacht nur über die Position durch

`point` *Bezeichner* <*x*,*y*,*z*>

definieren.
Ein Polygon wird durch

`polygon` *Bezeichner* {
    Standardeigenschaften ...
    Punktbezeichner1 Punktbezeichner2 ...
    `reverse_orientation`
}

definiert. Die *PunktbezeichnerX* geben die Eckpunkte des Polygons an. Die Reihenfolge der Punkte und damit die Orientierung wird umgedreht, wenn `reverse_orientation` angegeben ist.
Eine biquadratische G–Spline–Fläche `gspline` wird über ein entsprechendes Kontrollnetz definiert:

`gspline` *Bezeichner* {
    Standardeigenschaften ...
    Polygonbezeichner1 Polygonbezeichner2 ...
    `trim_ball` *x y z r*
    `trim_ellipsoid` *x y z rx ry rz*
    `trim_func` *m11 m12 m13 m21 m22 m23 m31 m32 m33 v1 v2 v3*
    `trim_curve` {
        Punktbezeichner
        <*x1*,*y1*> <*x2*,*y2*> ...
    }
    `trim_polygon` {
        Punktbezeichner
        <*x1*,*y1*> <*x2*,*y2*> ...
    }
    `trimlevel` *level*
    `doosabin` *level*
    `doosabin_draw` [`grid`| `all`|`last`
    `doosabin_color` <*rot*,*grün*,*blau*,*alpha*>
    `reverse_orientation`
}

Die *PolygonbezeichnerX* geben die Polygone des Kontrollnetzes an. Die Befehle `trim_func`, `trim_ball` und `trim_ellipsoid` definieren implizit ein Trimmobjekt im Raum. Mit `trim_curve` und `trim_polygon` werden lineare bzw. quadratische Trimmkurven im Parameterbereich definiert. Der *Punktbezeichner* gibt dabei an, für welche Bézierfläche, deren mittlerer Kontrollpunkt zu diesem Bezeichner gehört, die Trimmkurve angewendet wird. `trimlevel` gibt die maximale Rekursionstiefe für die Trimmalgorithmen an. `doosabin` legt die Anzahl der Doo–Sabin–Subdivisionen auf dem Kontrollnetz fest. Mit `doosabin_draw` kann man alle oder nur die letzte Stufe der Subdivision darstellen lassen. Wird der optionale Parameter `grid` angegeben, werden nur die Kontrollnetze und nicht die Fläche selbst angezeigt. Mit `doosabin_color` kann man die Farbe der Kontrollnetze festlegen. Mit `reverse_orientation` wird die Orientierung des Kontrollnetzes und damit auch die der Fläche umgedreht.

Eine Funktion auf einer G–Spline–Fläche wird wie folgt definiert:

```
function Bezeichner {
    Standardeigenschaften ...
    G–Spline–Bezeichner
    map {
        Punktbezeichner1: val11[,val12 ... ]
        Punktbezeichner2: val21[,val22 ... ]
         ...
    }
    curvature H|K|max|min|square
    fscale scale
    coldelta cdelta
    colscale cscale
    display color|colorf|grid|spikes|vectors ...
}
```

Der *G–Spline–Bezeichner* verweist auf die Fläche, auf der die Funktion definiert ist. Über `map` wird jedem Kontrollpunkt der Fläche ein Funktionskontrollpunkt zugewiesen. Natürlich müssen alle Flächenkontrollpunkte in der Liste vorkommen. Alternativ kann man über `curvature` eine Krümmungsfunktion definieren. `H` steht für die Gaußkrümmung, `K` für die mittlere Krümmung, `min` für die kleinste Hauptkrümmung, `max` für die größte Hauptkrümmung und `square` für die quadrierte Länge des Hauptkrümmungsvektors. `fscale` gibt einen Skalierungsfaktor für die gesamte Funktion an. `coldelta` und `colscale` geben die Verschiebungs- und Skalierungsfaktoren für die Farbdarstellung an. Mit `display` kann man schließlich festlegen welche Darstellungsarten verwendet werden sollen. `color` stellt die Funktion farbig auf der Fläche dar, `colorf` stellt sie als farbige Fläche über der Fläche dar, `grid` als Gitter über der Fläche, `spikes` als Stacheln auf der Fläche und `vectors` stellt drei–dimensionale Vektorfelder durch Vektoren dar.
Isolinien werden wie folgt definiert:

```
isoline Bezeichner {
    Standardeigenschaften ...
    G–Spline–Bezeichner
    function Funktionsbezeichner
    reflection <eyex,eyey,eyez> <planex,planey,planez> <lx,ly,lz> <dx,dy,dz>
    base baseval
    isostep step
    levels { level1 , level2 , ... }
    maxlevel mlevel
    maxlines mlines
    raise raiseval
    color_lines
    coldelta cdelta
    colscale cscale
}
```

G–Spline–Bezeichner gibt die G–Spline–Fläche an, auf der die Isolinien gezeichnet werden sollen. `function` gibt die Funktion an, für die die Isolinien gezeichnet werden sollen. Alternativ kann man Reflektionslinien mit `reflection` verwenden. `base` und `isostep` geben einen Wert und den Abstand zwischen den Werten für kontinuierliche Isolinien an. Alternativ kann man einzelne Werte mit `levels` festlegen. `maxlevel` legt die maximale Rekursionstiefe des Algorithmus' fest. `maxlines` gibt die maximale Anzahl der Isolinien an, die bei bilinearer Interpolation pro Gitterquadrat gezeichnet werden. Mit `raise` kann man die Isolinien um den angegebenen Faktor entlang der Normalen über oder unter die Fläche schieben. Wird `color_lines` angegeben, werden die Isolinien farbig gezeichnet. Die Farben werden mittels `coldelta` und `colscale` in

Abhängigkeit von dem Funktionswert festgelegt.
Eine Lichtquelle wird definiert durch

```
light {
    Standardeigenschaften ...
    direction <dx,dy,dz>
    spot_direction <dx,dy,dz>
    spot_exponent exponent
    spot_cutoff cutoff
    constant_attenuation ac
    linear_attenuation al
    quadratic_attenuation aq
}
```

Eine Lichtquelle besitzt keine Bezeichner. Es ist allerdings zu beachten, daß OpenGL nur 7 Lichtquellen zuläßt. Die Parameter entsprechen den OpenGL–Parametern zur Spezifikation von Lichtquellen. Die Spotlight–Parameter werden über `spot_direction`, `spot_exponent` und `spot_cutoff` bestimmt. Die Dämpfung des Lichts wird durch `constant_attenuation`, `linear_attenuation` und `quadratic_attenuation` festgelegt. Ein gerichtetes Licht wird über `direction` eingeführt.
Kommentare in ODL–Dateien beginnen mit einem # und werden immer durch das Zeilenende beendet.

ODL BNF

Das ODL–Format kann auch in BNF (Backus–Naur–Form) dargestellt werden. Im folgenden werden jedoch die unterschiedlichen Eigenschaften für verschiedene Objekttypen nicht berücksichtigt. Zur Vereinfachung werden sie alle durch `prop_value` repräsentiert. Das Startsymbol ist `<odl>`. Ein `<ident>` ist ein Bezeichner und `<real>` eine reelle Zahl.

```
<odl>             ::=   | <decl> <odl>
<decl>            ::=   <render>
                        | <integrate>
                        | <view>
                        | <light>
                        | <point>
                        | <polygon>
                        | <gspline>
                        | <function>
                        | <isoline>
                        | window <ident> <real> <real> <real> <real>
                        | orthographic <real> <real> <real> <real> <real>
                          <real>
                        | perspective <real> <real> <real> <real>
                        | local_viewer
                        | ambient_light <vector4>
                        | background <vector4>
                        | translate <vector3>
                        | rotate <vector3>
                        | scale <vector3>
                        | point_size <real>
                        | epsilon <real>
<render>          ::=   render <ident>
<integrate>       ::=   integrate_s <ident> <real> <real>
                        | integrate_v <ident> <real> <real>
                        | area <ident> <real> <real>
                        | volume <ident> <real> <real>
<view>            ::=   viewmode <mmode> <real> <real> <real> <real>
```

```
                              <real> <real> <real> <real>
<mmode>             ::=   object | eye
<light>             ::=   light { <prop> }
<point>             ::=   point <ident> <vector3>
                          | point <ident> { <prop> }
<polygon>           ::=   polygon <ident> { <prop> }
<gspline>           ::=   gspline <ident> { <prop> }
<function>          ::=   function <ident> { <prop> }
<isoline>           ::=   isoline <ident> { <prop> }
<prop>              ::=   | <prop_value> <prop>
<prop_value>        ::=   ambient <vector4>
                          | base <real>
                          | color_lines
                          | coldelta <real>
                          | colscale <real>
                          | constant_attenuation <real>
                          | curvature H
                          | curvature K
                          | curvature max
                          | curvature min
                          | curvature square
                          | diffuse <vector4>
                          | direction <vector3>
                          | display <display_mode>
                          | doosabin <real>
                          | doosabin_color <vector4>
                          | doosabin_draw all
                          | doosabin_draw last
                          | doosabin_draw grid all
                          | doosabin_draw grid last
                          | emission <vector4>
                          | function <ident>
                          | fscale <real>
                          | levels { <reallist> }
                          | linear_attenuation <real>
                          | location <vector3>
                          | maxlevel <real>
                          | maxlines <real>
                          | quadratic_attenuation <real>
                          | map { <map_desc> }
                          | raise <real>
                          | reflection <vector3> <vector3> <vector3> <vector3>
                          | resolution <real>
                          | reverse_orientation
                          | rotate <vector3>
                          | scale <vector3>
                          | shininess <real>
                          | specular <vector4>
                          | spot_cutoff <real>
                          | spot_direction <vector3>
                          | spot_exponent <real>
                          | step <real>
                          | translate <vector3>
                          | trim_ball <real> <real> <real> <real>
```

```
                          | trim_curve { <ident> <trimcurve_desc> }
                          | trim_ellipsoid <real> <real> <real> <real>
                              <real> <real>
                          | trim_func <real> <real> <real> <real> <real>
                              <real> <real> <real> <real> <real> <real>
                              <real> <real>
                          | trim_polygon { <ident> <trimcurve_desc> }
                          | trimlevel <real>
                          | <ident>
<display_mode>     ::=    color <x_display_mode>
                          | colorf <x_display_mode>
                          | grid <x_display_mode>
                          | spikes <x_display_mode>
                          | vectors <x_display_mode>
<x_display_mode>   ::=    | color <x_display_mode>
                          | colorf <x_display_mode>
                          | grid <x_display_mode>
                          | spikes <x_display_mode>
                          | vectors <x_display_mode>
<trimcurve_desc>   ::=    | <vector2> <trimcurve_desc>
<map_desc>         ::=    | <ident> :  <reallist> <map_desc>
<reallist>         ::=    <real> | <real> , <reallist>
<vector2>          ::=    "<" <real> , <real> ">"
<vector3>          ::=    "<" <real> , <real> , <real> ">"
<vector4>          ::=    "<" <real> , <real> , <real> , <real> ">"
```

Beispiel

Folgende ODL–Datei definiert ein Möbiusband über 8 Polygone mit Hilfe der Doo-Sabin–Subdivision. Zusätzlich wird eine Funktion auf dem Möbiusband definiert. Die Funktion wird durch `color` angezeigt und sowohl der Flächeninhalt als auch das Oberflächenintegral der Funktion wird berechnet.

```
# moebius.odl: Möbiusband.

point p_1_0 <0,2,0>
point p_2_0 <2,0,0>
point p_3_0 <5,2,2>
point p_4_0 <2,4,0>

point p_1_1 <0,2,2>
point p_2_1 <2,0,2>
point p_3_1 <4,2,2>
point p_4_1 <2,4,2>

point p_1_2 <0,2,4>
point p_2_2 <2,0,4>
point p_3_2 <3,2,2>
point p_4_2 <2,4,4>

polygon P1_0 {
  p_1_0 p_2_0 p_2_1 p_1_1
  ambient <0.0,0.0,0.0,1.0>
}
polygon P2_0 {
  p_2_0 p_3_0 p_3_1 p_2_1
```

```
  ambient <0.0,0.0,0.0,1.0>
}
polygon P3_0 {
  p_3_0 p_3_1 p_4_1 p_4_2
  ambient <0.0,0.0,0.0,1.0>
}
polygon P4_0 {
  p_4_2 p_4_1 p_1_1 p_1_2
  ambient <0.0,0.0,0.0,1.0>
}

polygon P1_1 {
  p_1_1 p_2_1 p_2_2 p_1_2
  ambient <0.0,0.0,0.0,1.0>
}
polygon P2_1 {
  p_2_2 p_2_1 p_3_1 p_3_2
  ambient <0.0,0.0,0.0,1.0>
}
polygon P3_1 {
  p_3_2 p_3_1 p_4_1 p_4_0
  ambient <0.0,0.0,0.0,1.0>
}
polygon P4_1 {
  p_4_1 p_4_0 p_1_0 p_1_1
  ambient <0.0,0.0,0.0,1.0>
}

gspline gs {
  P1_0 P2_0 P3_0 P4_0
  P1_1 P2_1 P3_1 P4_1
  shininess 0.5
  diffuse   <0.65,0.0,0.0,1.0>
  ambient   <0.5,0.0,0.0,1.0>
  specular  <1.0,0.6,0.6,1.0>
  location  <0,0,0>
  resolution 8
  reverse_orientation
  doosabin 2
  # doosabin_draw grid last
  # doosabin_color <0.0,0.0,0.0,1.0>
}

function func {
  gs
  map {
    p_1_0:4
    p_2_0:4
    p_3_0:9
    p_4_0:4
    p_1_1:4
    p_2_1:4
    p_3_1:4
    p_4_1:4
```

```
    p_1_2:4
    p_2_2:4
    p_3_2:1
    p_4_2:4
  }
  shininess 1.0
  diffuse <1.0,1.0,1.0,1.0>
  ambient <1.0,1.0,1.0,1.0>
  specular <1.0,1.0,1.0,1.0>
  emission <0.0,0.0,0.0,0.0>
  resolution 10
  colscale 0.7
  coldelta -0.6
  display color
  # display grid spikes colorf
  fscale 1
}

light {
  ambient <1.0,1.0,1.0,1.0>
  diffuse <1.0,1.0,1.0,1.0>
  specular <1.0,1.0,1.0,1.0>
  location <2,-1,4>
  constant_attenuation 1.0
  linear_attenuation 0.25
  quadratic_attenuation 0.015
}

window Moebiusband 0 0 500 500

background <1.0, 1.0, 1.0, 1.0>
ambient_light <0.4, 0.4, 0.4, 1.0>

viewmode eye 65.0 10 8 1 5.5 0 0 1
perspective 35.0 1.0 0.01 100.0

# render gs
render func

area gs 50 0.000015625
integrate_s func 50 0.000015625
```

4.3 Erzeugen von ODL–Dateien

Wir stellen nun noch zwei Methoden zur Erzeugung der ODL–Dateien vor. Eine Möglichkeit, kompliziertere Objekte einfach zu beschreiben ist, sie aus gleichgroßen Würfeln zusammenzusetzen. Dies wird über das SBW–Dateiformat ermöglicht. Hierdurch erhalten wir Flächenkontrollnetze mit Irregularitäten. Eine andere Möglichkeit besteht darin, die Flächen aus einer Parametrisierung zu erzeugen. Dabei entstehen aber nur reguläre Kontrollnetze.

Die Programme zur Umwandlung spezieller Formate in ODL wurden alle in Perl implementiert. Es lassen sich auf ähnliche Weise über Skriptsprachen auch andere Wege zur Erzeugung von ODL–Dateien implementieren. Man kann so auch andere Formate in ODL umwandeln.

4.3.1 SBW–Format

Das SBW–Format[13] ermöglicht es, G–Spline–Flächen durch zusammensetzen von gleichgroßen Würfeln mit der Seitenlänge 1 zu erzeugen. Die Ecken der Würfel legen dabei die Kontrollpunkte fest. Die Polygone für das Kontrollnetz werden aus den Seiten der Würfel gebildet. Dabei sind aber nur die Seiten zu berücksichtigen, die nur Teil eines einzigen Würfels sind. Alle anderen liegen innerhalb oder außerhalb des Körpers. Um die Flächen interessanter zu gestalten, kann man die Würfel auch noch längs ihrer x–, y– oder z–Achse verdrehen. Hierzu müssen nur die Kontrollpolygone jeweils um eine Seite versetzt gewählt werden. Neben der Fläche selbst können wir auch gleich eine Funktion auf dieser Fläche über Funktionskontrollpunkte bestimmen. Dies wird über eine von den Koordinaten des Flächenkontrollpunktes abhängige Parametrisierung erreicht. Durch zweimaliges anwenden der Doo–Sabin–Subdivision kann man aus so erzeugten Kontrollnetzen semi–reguläre Kontrollnetze für biquadratische G–Splines erzeugen.

Die Anordnung der Würfel kann über eine ASCII Datei festgelegt werden. Wir geben dabei nacheinander die x–y–Ebenen in z–Richtung an. Das so erzeugte SBW–Format hat folgende Form:

X`x`Y`x`Z[`;`*func*]

$$
\begin{array}{cccc}
a_{1,1,1} & a_{2,1,1} & \cdots & a_{X,1,1} \\
\vdots & & \cdots & \vdots \\
a_{1,Y,1} & a_{2,Y,1} & \cdots & a_{X,Y,1} \\
a_{1,1,2} & a_{2,1,2} & \cdots & a_{X,1,2} \\
\vdots & & \cdots & \vdots \\
a_{1,Y,2} & a_{2,Y,2} & \cdots & a_{X,Y,2} \\
\vdots & & & \vdots \\
a_{1,Y,Z} & a_{2,Y,Z} & \cdots & a_{X,Y,Z}
\end{array}
$$

Die Zeilenumbrüche werden hier berücksichtigt. X, Y und Z geben die maximale Anzahl der Würfel in x–, y–, und z–Richtung an. Dabei wird vom Ursprung in die jeweilige positive Richtung gezählt. Optional kann man eine Funktion *func* im Perl Syntax angeben, welche die Funktionskontrollpunkte festlegt. Diese kann auf die Koordinaten `$x`, `$y`, `$z` des Flächenkontrollpunktes zugreifen. Die Würfel werden durch den Rest der Datei bestimmt. Dazu wird der Raum in ein drei–dimensionales Gitter zerlegt. Die Belegung der einzelnen Zellen wird durch das Array a bestimmt. Ein `.` markiert eine leere Zelle, ein `*` einen Würfel in dieser Zelle, `x` ,`X`, `y`, `Y`, `z` und `Z` markieren Zellen, in denen der Würfel um die x–, y– oder z–Achse verdreht wurde.

Die in Abbildung 4.2 dargestellte Fläche wurde mit Hilfe der folgenden SBW–Datei erzeugt:

```
2x2x2;$x*$x+$y*$y+$z*$z
**
y.
*.
..
```

Die Farben und die Lichtquelle wurden durch editieren der ODL–Datei angepaßt. Die Abbildung zeigt auch nur die Fläche und nicht die darauf definierte Funktion.

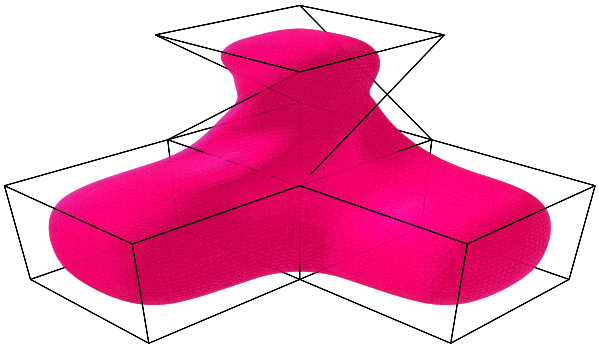


Abbildung 4.2: Beispiel zum SBW–Format

Das Perlskript `sbw2odl.pl` konvertiert eine Datei im SBW–Format in das ODL–Format. Die SBW–Dateien werden dazu als Argumente übergeben.

[13] SBW steht für “**S**imple **B**lock **W**orld”

4.3.2 FNC–Format

Durch das FNC–Format[14] können reguläre Kontrollnetze über Parametrisierungen erzeugt werden. Wir müssen dazu nur die Parametrisierungen für die x–, y– und z–Koordinaten und das zugehörige Gitter im Parameterbereich angeben. Über das Gitter im Parameterbereich erhalten wir dann das Kontrollnetz aus den Parametrisierungen. Zusätzlich können wir auch noch eine Parametrisierung für Funktionskontrollpunkte hinzufügen.

Das FNC–Format hat dann folgende Form:

$X\,;Y\,;Z[\,;\mathit{func}]$
$u0:du:u1$
$v0:dv:v1$

Auch hier werden die Zeilenumbrüche berücksichtigt. Das Gitter im Parameterbereich liegt über dem zwei–dimensionalen Intervall $[u0,u1]\times[v0,v1]$ mit du als Abstand in u–Richtung und dv als Abstand in v–Richtung. X, Y, Z sind die Parametrisierungen für die Flächenkontrollpunkte im Perl–Syntax, wobei die Variablen `$u` und `$v` als Parameter verwendet werden. Optional kann man die Funktionskontrollpunkte über *func* in Perl–Syntax hinzufügen. Hierfür können die Parameter `$u`, `$v` und die Koordinaten `$x`, `$y` und `$z` verwendet werden.

Die Kleinsche Flasche können wir z.B. durch eine um sich selbst rotierende Acht, deren Mittelpunkt gleichzeitig um eine Achse rotiert, darstellen. Die Parametrisierung hierfür lautet

$$\begin{bmatrix} \left(2+\cos\left(\frac{u}{2}\right)\sin(v)-\sin\left(\frac{u}{2}\right)\sin(2v)\right)\cos(u) \\ \left(2+\cos\left(\frac{u}{2}\right)\sin(v)-\sin\left(\frac{u}{2}\right)\sin(2v)\right)\sin(u) \\ \sin\left(\frac{u}{2}\right)\sin(v)+\cos\left(\frac{u}{2}\right)\sin(2v) \end{bmatrix} \tag{4.1}$$

(vgl. [Gra94]). Dies im FNC–Format ergibt:

```
(2+cos($u/2)*sin($v)-sin($u/2)*sin(2*$v))*cos($u); ...
      (2+cos($u/2)*sin($v)-sin($u/2)*sin(2*$v))*sin($u); ...
            sin($u/2)*sin($v)+cos($u/2)*sin(2*$v)
-0.78:0.31:4.7
0:0.31:6.3
```

Die ... sollen dabei nur andeuten, daß in der Datei selbst kein Zeilenumbruch an dieser Stelle stehen darf. Aus der FNC–Datei erhalten wir eine ODL–Datei, mit der wir die in Abbildung 4.3 dargestellte Fläche erzeugen können. Auch wurde der Parameterbereich so gewählt, daß die Fläche an einer Seite offen ist. Die Materialeigenschaften und die Kameraposition wurden in der ODL–Datei editiert. Um eine geschlossene Fläche zu erhalten, haben wir die ersten und letzten Punkte bzgl. v in der ODL–Datei miteinander identifiziert.

Die FNC–Dateien werden durch das Perlskript `fnc2odl.pl` ins ODL–Format konvertiert. Die FNC–Dateinamen werden dazu als Argumente übergeben.

[14] FNC ist eine Abkürzung für "**Fun**c**c**tion"

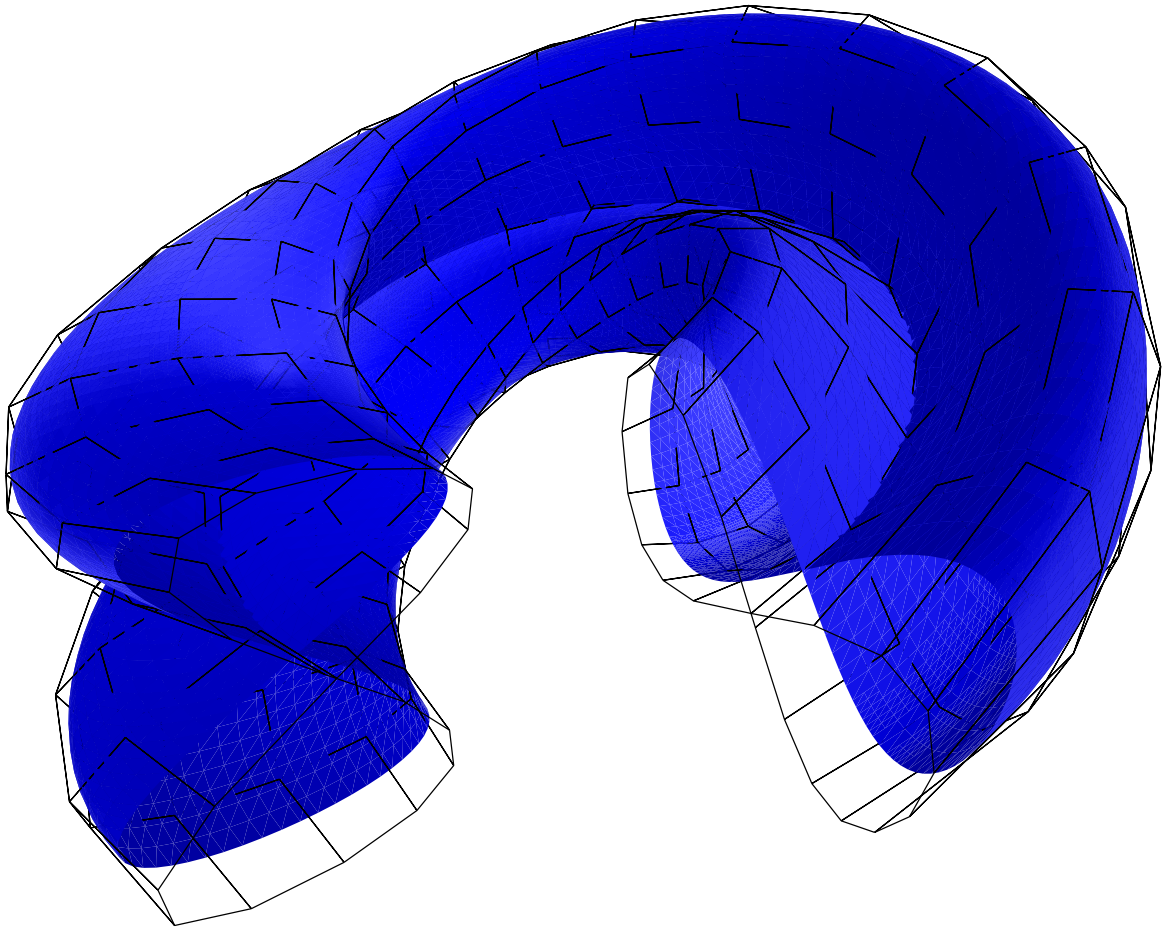


Abbildung 4.3: Kleinsche Flasche

Anhang A

Programm–Dokumentation

Die vollständige Programm–Dokumentation von LiLit wurde mit mit Hilfe des DOC++ Dokumentationssystems aus dem Quellcode erzeugt. Sie enthält die Dokumentation sämtlicher Klassen, Funktionen, etc. von LiLit und dient vor allem als Referenzmaterial zum Verständnis des Quellcodes. Für weitere Details sei auf den Quellcode und die darin enthaltenen Kommentare verwiesen.

Aus Platzgründen befindet sich der Text zur Programm–Dokumentation im HTML–Format auf der beigelegten CD–ROM im Verzeichnis `HTML/src`. Sie kann am besten mit einem WWW bzw. HTML Browser beginnend mit der Datei `HTML/src/index.html` angeschaut werden. Die besten Ergebnisse erhält man mit einem JAVA–fähigen Browser. Zusätzlich bietet das HTML Format und JAVA den Vorteil, die Abhängigkeiten und Verbindungen zwischen den Klassen über Links darzustellen.

LiLit Version 0.2 CD–ROM Index

Die CD–ROM enthält alle für diese Arbeit entwickelten Programme, den zugehörigen Quellcode und die Quellen für diese Arbeit.

Notation

$f(A)$	Abkürzung für das Bild eine Menge A: $\{f(x) : x \in A\}$
$f^{(k)}$	k–te Ableitung von f
$f_+^{(k)}, f_-^{(k)}$	k–te links- bzw. rechtsseitige Ableitung von f
f_x	Partielle Ableitung von f nach x
∇f	Gradient von f
Df	Ableitung bzw. Funktionalmatrix von f
$\operatorname{supp} f$	Support von f
$(x)_+$	Abkürzung für $max\{x, 0\}$
$a : b$	Zahlen von a bis b mit der Schrittweite 1
$a :$	Zahlen von a bis unendlich mit der Schrittweite 1
$a : s : b$	Zahlen von a bis b mit der Schrittweite s
$a : s :$	Zahlen von a bis unendlich mit der Schrittweite s
$[\ldots]$	Vektor oder Matrix
E	Einheitsmatrix
a^T	Transponierter Vektor oder Matrix von a
$\langle a, b\rangle$	Skalarprodukt der Vektoren a, b
A^+	Pseudoinverse der Matrix A
$\|x\|$,$\|x\|_2$	Euklidische Norm von x
$O(X)$	Landausymbol für die Größenordnung einer Funktion
$\delta_{j,k}$	Kroneckersymbol
$\mathbb{N}$	Menge der natürlichen Zahlen ohne 0
$\mathbb{N}_0$	Menge der natürlichen Zahlen mit 0
$\mathbb{Z}$	Menge der ganzen Zahlen
$\mathbb{R}$	Menge der reellen Zahlen
$\mathbb{R}^+$	Menge der positiven, reellen Zahlen
0	Abkürzung für einen Nullvektor oder eine Nullmatrix
$B_{j,d,u}$	B–Splines
B_j^d	Bernstein–Polynome
L_j	Lagrange–Polynome
$S_{u,d}(U)$	Splineraum
$\omega_{j,d}(x)$	Koeffizienten der Rekursionsformel für B–Splines
`ident`	Bezeichner von Variablen, Algorithmen, etc. werden in einer Typewriter Schrift dargestellt

Literaturverzeichnis

[ABB 95] Anderson, E. ; Bai, Z. ; Bischof, C. ; Demmel, J. ; Dongarra, J. ; Croz, J. D. ; Greenbaum, A. ; Hammarling, S. ; McKenney, A. ; Ostrouchov, S. ; Sorensen, D.: *LAPACK Users' Guide*. 2nd edition. Philadelphia: Society for Industrial and Applied Mathematics, 1995

[Amm95] Ammeraal, Leendert: *C++ for Programmers*. 2nd edition. Chichester; New York; Brisbane; Toronto; Singapore : John Wiley and Sons Ltd., 1995

[BBP96] Barth, Rainer ; Beier, Ekkehard ; Pahnke, Bettina: *Grafikprogrammierung mit OpenGL*. Reading, Massachusetts; Paris, Singapore; Bonn : Addison Wesley, 1996 (Praktische Informatik)

[Boo78] Boor, Carl de: *Applied Mathematical Sciences*. Bd. 27 : A Practical Guide to Splines. New York, Berlin; Heidelberg; London; Paris; Tokyo; Hong Kong; Barcelona : Springer Verlag, 1978

[BS89] Bronstein, I.N. ; Semendjajew, K.A.: *Taschenbuch der Mathematik*. 24. Auflage. Leipzig : BSB Teubner, 1989

[Car93] do Carmo, Manfredo P.: *Differentialgeometrie von Kurven und Flächen*. 3., durchgesehene Auflage. Braunschweig; Wiesbaden : Vieweg, 1993 (Vieweg Studium: Aufbaukurs Mathematik)

[CC78] Catmull, E. ; Clark, J.: Recursively generated B–Spline surfaces on arbitrary topological meshes. **In:** *Computer–Aided Design* (1978), September, S. 10:350–355

[Doo78] Doo, Donald: A subdivision algorithm for smoothing down irregularly shaped polyhedrons. **In:** *Proced. Int'l Conf. Interactive Techniques in Computer Aided Design* IEEE Computer Society, 1978, S. 157–165

[Dot92] Dotzauer, Ernst: *Mathematische Modellierung von 3D–Freiformobjekten*. München; Wien : Carl Hanser Verlag, 1992

[Far93] Farin, Gerald: *Kurven und Flächen im Computer Aided Geometric Design – Eine praktische Einführung*. 2. Auflage. Braunschweig; Wiesbaden : Vieweg, 1993

[FDFH90] Foley, James D. ; Dam, Andries van ; Feiner, Steven K. ; Hughes, John F.: *Computer Graphics – Principles and Practice*. 2nd edition. Reading, Massachusetts; Paris, Singapore; Bonn : Addison Wesley, 1990 (Addison Wesley Systems Programming Series)

[GKV89] Gerthsen, Christian ; Kneser, Hans O. ; Vogel, Helmut: *Physik, Ein Lehrbuch zum Gebrauch neben Vorlesungen*. 16. Auflage. Berlin, Heidelberg, New York, London, Paris, Tokyo, Hong Kong : Springer, 1989

[Gra94] Gray, Alfred: *Differentialgeometrie, Klassische Theorie in moderner Darstellung*. Heidelberg, Berlin, Oxford : Spektrum Akademischer Verlag, 1994

[Gri92] GRIEGER, Ingolf: *Graphische Datenverarbeitung: Mit einer Einführung in PHIGS und PHIGS–PLUS*. 2., überarbeitete und erweiterte Auflage. Berlin; Heidelberg; New York; London; Paris; Tokyo; Hong Kong; Barcelona; Budapest : Springer Verlag, 1992

[Heu90] HEUSER, Harro: *Lehrbuch der Analysis, Teil 2*. 5., durchgesehene Auflage. Stuttgart : B.G. Teubner, 1990 (Mathematische Leitfäden)

[HH92] HÄMMERLIN, Günther ; HOFFMAN, Karl–Heinz: *Grundwissen Mathematik*. Bd. 7 : Numerische Mathematik. 3., unveränderte Auflage. Berlin; Heidelberg; New York; London; Paris; Tokyo; Hong Kong; Barcelona; Budapest : Springer Verlag, 1992

[HL92] HOSCHEK, Josef ; LASSER, Dieter: *Grundlagen der geometrischen Datenverarbeitung*. 2., neubearbeitete und erweiterte Auflage. Stuttgart : B. G. Teubner, 1992

[Höl94] HÖLLIG, Klaus: *Script zur CAGD Vorlesung*. unveröffentlicht, 1994. – Universität Stuttgart, Mathematisches Institut A, 2. Lehrstuhl

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

[KU98] KROMMER, Arnold R. ; UEBERHUBER, Crhistoph W.: *Computational Integration*. Philadelphia : Society for Industrial and Applied Mathematics, 1998

[Rei95] REIF, Ulrich: Biquadratic G–Spline Surfaces. **In:** *Computer Aided Geometric Design* (1995), 12, S. 193–205

[Sad98] SADR, Babak: *Unified Objects: Object–Oriented Programming Using C++*. Los Alamitos, California; Washington; Brussels; Tokyo : IEEE Computer Society, 1998

[Sey96] SEYERLE, Holger: *Flächen und Funktionen auf Flächen in der geometrischen Datenverarbeitung*, Universität Stuttgart; Mathematisches Institut A, Diplomarbeit, 1996

[Sto93] STOER, Josef: *Numerische Mathematik 1*. 6., korrigierte Neuauflage. New York, Berlin; Heidelberg; London; Paris; Tokyo; Hong Kong; Barcelona : Springer Verlag, 1993. – Eine Einführung – unter Berücksichtigung von Vorlesungen von F.L. Bauer

[Str87] STROUSTRUP, Bjarne: *Die C++ Programmiersprache*. Reading, Massachusetts; Paris, Singapore; Bonn : Addison Wesley, 1987 (Internationale Computer–Bibliothek)

[WCS96] WALL, Larry ; CHRISTIANSEN, Tom ; SCHWARTZ, Randal L.: *Programming Perl*. 2nd edition. Sebastopol, Bonn, Cambridge, Paris, Tokyo : O'Reilly & Associates, Inc., 1996

[WND96] WOO, Mason ; NEIDER, Jackie ; DAVIS, Tom: *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.1*. 2nd edition. Reading, Massachusetts; Paris, Singapore; Bonn : Addison Wesley, 1996

Erklärung

Ich erkläre, daß ich die vorliegende Arbeit selbständig und nur mit den angegebenen Hilfmitteln angefertigt habe, und daß alle Stellen, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen sind, durch Angabe der Quellen als Entlehnung deutlich kenntlich gemacht worden sind.

Frank Langbein
Stuttgart, Mai 1999