Diplomarbeit

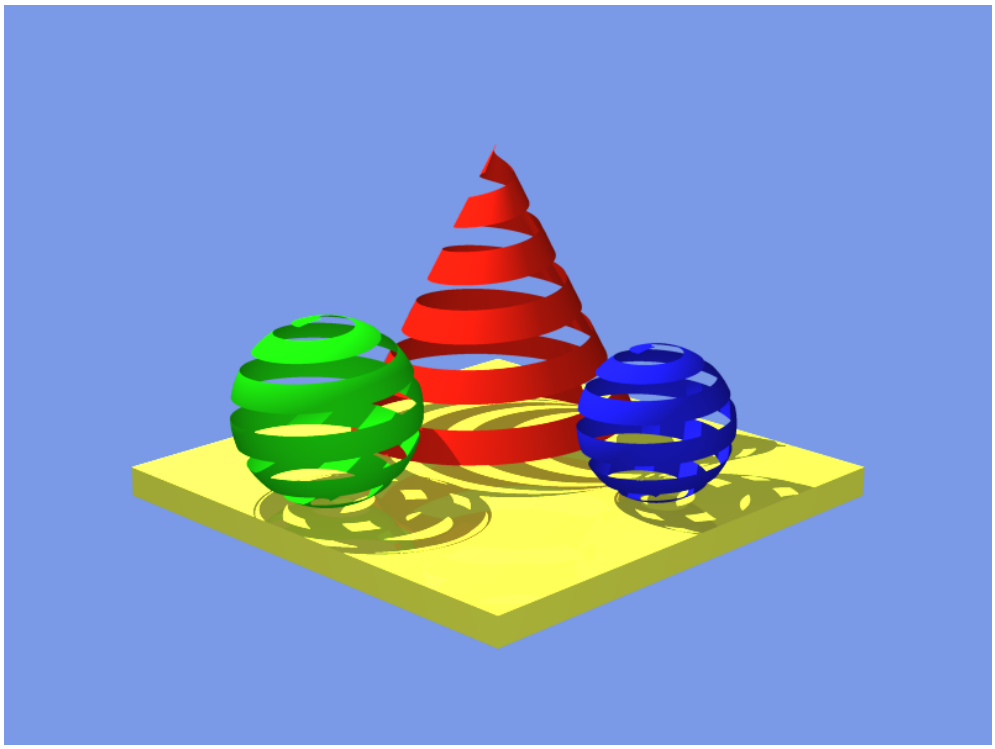
Ray Tracing allgemeiner
Flächen in Parameter-Darstellung

ausgeführt am
Institut für Computergrafik
Abt. Algorithmen und Programmiermethoden
der Technischen Universität Wien

unter Anleitung von
O.Univ.Prof. Dr. Wilhelm Barth
und
Univ.Ass. Dipl-Ing. Michael Schindler

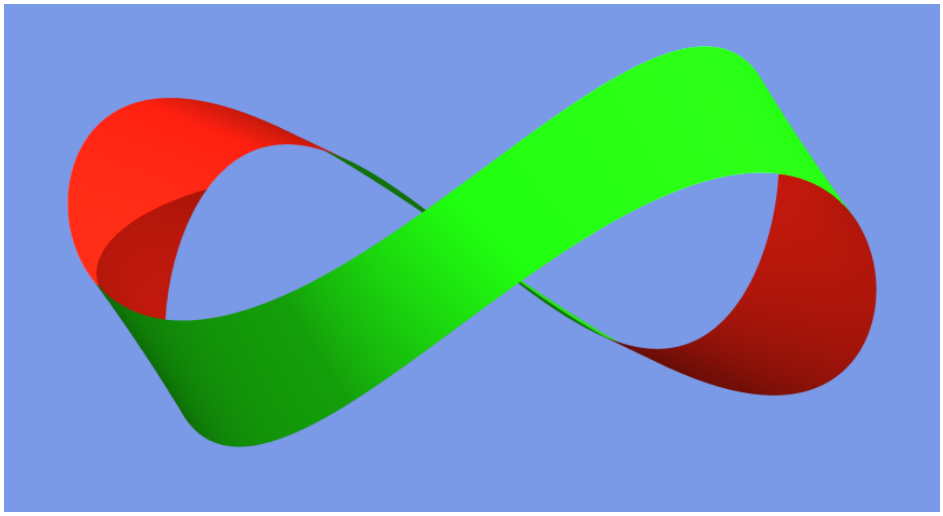
durch
Roland LIEGER
Matr.Nr. 8825560
Goethegasse 39
2340 Mödling

März - September 1992



Scene #2

Inhaltsverzeichnis:



Möbius-Band

0. Kurzfassung

Diese Diplomarbeit beschäftigt sich damit, Flächen in Parameter-Darstellung durch Ray Tracing darzustellen. Unter einer Fläche in Parameter-Darstellung versteht man eine Fläche, die durch Formeln X(u, v), Y(u, v) und Z(u, v) über einem rechteckigen (u, v) Parameter-Bereich definiert ist. Für die Berechnung eines Bildes durch Ray Tracing ist es erforderlich, sehr viele Strahlen effizient mit den Objekten der Szene zu schneiden und die Oberflächennormalen an den Schnittpunkten zu bestimmen.

Um später schnell alle Schnittpunkte zwischen einem Strahl und einer Fläche in Parameter-Darstellung auffinden zu können, wird die Fläche in der Vorbereitungsphase sukzessive unterteilt und für jedes Flächenstück ein umschließendes Parallelepiped bestimmt. Zum späteren Aufsuchen der relevanten Flächenteile bzw. Parallelepipede werden die Epipede in einem binären Baum angeordnet. Dazu wird zunächst ein Parallelepiped erstellt, das die gesamte Fläche einschließt. Wenn dieses Epiped die Fläche nicht eng genug umschließt, wird die Fläche in zwei Teile zerlegt, für die dann je ein neues einschließendes Epiped erstellt wird. Diese beiden Epipede sind dann Kinder des Ursprungsepipeds. Auf diese Weise wird die Fläche so lange in Teile zerlegt, bis jedes Stück fast eben ist und eng von einem Epiped umschlossen wird. Dabei entsteht der Baum der Parallelepipede.

Wenn nun für einen konkreten Strahl die Schnittpunkte mit der Fläche gesucht werden, so wird ein Suchvorgang im Baum der Parallelepipede gestartet. Schneidet der Strahl ein Parallelepiped nicht, so schneidet er auch sicher nicht den darin enthaltenen Flächenteil. Wenn ein Schnitt existiert, so werden die beiden Nachfolger im Epipedbaum untersucht. Auf diese Weise werden rasch jene Blattepipede gefunden, die der Strahl schneidet. Da diese kleinen Flächenstücke fast eben sind, kann nun relativ einfach der Schnittpunkt durch Newton-Iteration gefunden werden.

Um den Ray Tracer unabhängig von einer dargestellten Fläche zu halten, werden die Formeln der Parameterfläche als Daten eingelesen. Sie werden dann mit einer kontextfreien Grammatik in Bäume übersetzt, bei denen die Blattknoten Variable und Konstante enthalten, während die inneren Knoten die Operatoren wiedergeben. Als nächstes werden die Bäume optimiert. Dabei werden Operationen, die nur von Konstanten abhängen sofort ausgeführt (z.B. Sqrt(2) ~ 1.41421...). 'Tote' Operationen, wie Multiplikationen mit 0.0 oder 1.0 werden entfernt. Auch Negationen müssen oft nicht explizit ausgeführt werden, sondern lassen sich in einer anderen Operation verstecken (z.B. -X + Y → Y - X). Wenn bei einer Grundrechnungsart zweimal der gleiche Operand angegeben wird, kann dies auch kompakter angeschrieben werden (z.B. X*X → X^2 ~ X-X → 0). Die so optimierten Bäume werden nun in Postfix-Code umgewandelt. Dabei wird darauf geachtet, daß Teilausdrücke, die mehrfach (auch in verschiedenen Formeln) auftreten, nur einmal in Postfix-Code umgesetzt werden. Dafür wird zusätzlicher Code erstellt, der das Ergebnis der Auswertung in ein Register sichert, sodaß später nur noch eine Referenz auf das Ergebnis in den Postfix-Code aufgenommen werden muß. So entsteht Postfix-Code, der fast so effizient ausgewertet werden kann wie compilierter C-Code.

Durch den Postfix-Ausdruck lassen sich die Formeln leicht an einer Stelle auswerten. Durch automatisches Differenzieren ist es aber auch möglich, zusätzlich zum Funktionswert auch die Werte der partiellen Ableitungen zu berechnen. Dies ist beim Ray Tracing notwendig, um die Oberflächennormalen zu bestimmen. Weiters ist es möglich, den Postfix-Code mit Intervall-Arithmetik auszuwerten. Dabei wird nicht der Funktionswert an einem Punkt des Parameterbereichs bestimmt, sondern es werden Schranken für den Funktionswert über einem Parameterbereich berechnet. Durch Kombination der automatischen Differenzierung mit der Intervall-Arithmetik ist es möglich, neben den Schranken für den Funktionswert auch Grenzen für die Werte der partiellen Ableitungen zu erhalten. Dies ist sehr nützlich, wenn ein Flächenstück durch einen engen Hüllkörper umschlossen werden soll.

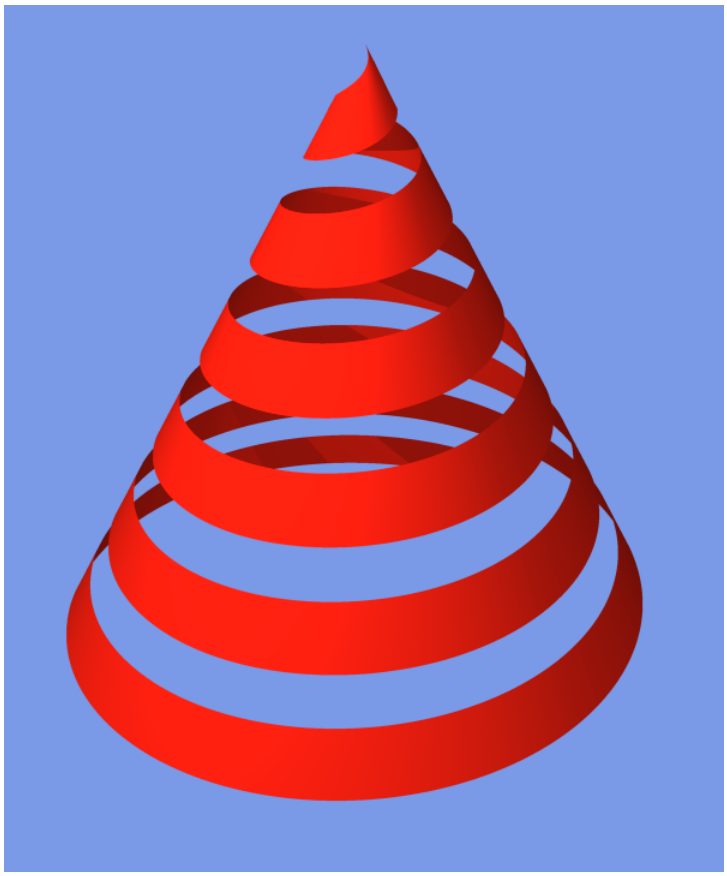
1. Behandelte Flächen

Ziel dieser Diplomarbeit ist, 'glatte' Flächen, die in Parameter-Darstellung gegeben sind, durch Ray Tracing darzustellen.

Unter der Parameter-Darstellung einer Fläche versteht man die Definition der Flächenpunkte durch explizite Gleichungen für die X, Y und Z Richtung in Abhängigkeit von zwei freien Parametern u und v. Weiters wird für u und v je ein Definitionsbereich angegeben.

$$XYZ(u,v) = \begin{cases} X(u,v) \\ Y(u,v) \\ Z(u,v) \end{cases} \quad u \in U := [u_0 ... u_1],\ v \in V := [v_0 ... v_1]$$

Als Beispiel für eine solche Definition sei hier die Parameter-Darstellung eines Spiralkegels angegeben:



Cone

```
X(u,v) = Sin(12*u) * (u+v);
Y(u,v) = Cos(12*v) * (u+v);
   Z(u,v) = -2.5 * (u+v);
u ∈ [0..π] ~ v ∈ [-0.1..0.1]
```

Wenn man keine Beschränkungen für die Art der Gleichungen für X, Y und Z aufstellt, kann es passieren, daß die Fläche in unzusammenhängende Teilflächen, und im Extremfall zu feinverteiltem Staub zerfällt. Derartige Punktewolken sind aber kaum mehr als 'glatte' Flächen zu bezeichnen und werden deshalb hier nicht betrachtet.

Im Rahmen dieser Diplomarbeit sollen nur 'brave' Flächen behandelt werden. Bei diesen müssen die Gleichungen für X, Y und Z im Definitionsbereich stetig und fast überall differenzierbar sein. Die Forderung nach Stetigkeit erzwingt, daß die Funktionen keine Sprünge machen, also auch nicht in Teilflächen zerfallen können. Um den Normalvektor auf eine Fläche, der intern vom Ray Tracing benötigt wird, an einem Punkt berechnen zu können, ist die Differenzierbarkeit der einzelnen Gleichungen nach u und v Voraussetzung. Diese Bedingung ist notwendig, aber nicht hinreichend für die Berechenbarkeit des Normalvektors. Sind die Ableitungen nach u und v an einem Punkt linear abhängig oder ist mindestens einer der Ableitungsvektoren gleich dem Nullvektor, so ist der Normalvektor an diesem Punkt nicht berechenbar. An solchen Punkten können auch Knicke und Ecken (aber keine Sprünge) in einer Fläche auftreten, deren Funktionen für jede der drei Raumrichtungen stetig differenzierbar sind. Weiters wird gefordert, daß nicht nur die Funktionen, sondern auch ihre partiellen Ableitungen nach u und v stetig und beschränkt sind. Somit sind unter anderem oszillierende Funktionen wie Sin(1/u) in der Umgebung von Null unzulässig.

Auch bei der Auswahl der Definitionsbereiche ist darauf zu achten, daß die Flächen nicht ausarten. Besteht die Definitionsmenge der (u,v)-Parameter aus mehreren, unzusammenhängenden Bereichen der (u,v)-Ebene, so wird im allgemeinen auch die daraus errechnete parametrisierte Fläche aus unzusammenhängenden Teilen bestehen. Ebenso gilt, daß isolierte Punkte im (u, v)-Definitionsbereich, beziehungsweise ein aus Staub bestehender Definitionsbereich ihren Durchschlag auf die Fläche finden werden. Nur wenn der (u, v)-Definitionsbereich aus genau einem einfachen, zusammenhängenden Bereich besteht, ist weitgehend sichergestellt, daß dies auch für die Fläche in Parameter-Darstellung gelten wird.

Um eine einfache Angabe des (u,v)-Parameterbereichs zu ermöglichen wurde die Menge der zulässigen Definitionsbereiche noch weiter beschränkt. Im Rahmen dieser Diplomarbeit sind nur rechteckige, bezüglich der u und v Richtung achsparallele Parameterintervalle erlaubt. Andere zusammenhängende, (auch nicht-einfache) Formen des Definitionsbereichs, wie allgemeine Parallelogramme, Kreise, (Kreisringe,) Ellipsen oder Ellipsensegmente sind durch einfache Übergänge von u und v auf transformierte Parameter u' und v' leicht möglich.

2. Einführung in Ray Tracing

Ray Tracing ist eine Methode zur Erzeugung realistischer Raster-Bilder mit einem Computer.

In der Natur senden Lichtquellen Strahlen aus. Diese breiten sich geradlinig aus, bis sie auf ein Objekt treffen. Dort werden sie teils absorbiert, teils nach den optischen Gesetzen reflektiert, gebrochen oder gestreut. Welche dieser Möglichkeiten dabei einen bedeutenden Anteil hat, hängt vom Material des Objektes ab. Alle Lichtstrahlen, die direkt oder indirekt in eine Kamera gelangen, tragen dazu bei, auf dem Film ein Bild entstehen zu lassen.

Eine Möglichkeit, mit einem Computer realistische Bilder zu erstellen, wäre, den natürlichen Prozeß für jeden Strahl zu simulieren. Die optischen Gesetze sind bekannt und (in guter Näherung) berechenbar. Allerdings hat dieses Verfahren den gravierenden Nachteil, daß nur ein geringer Teil der ausgesandten Lichtstrahlen in die Kamera trifft und dort einen Beitrag zum Bild liefert. Der Großteil des gewaltigen Rechenaufwands für die Strahlverfolgung wäre also umsonst.

Rückwärts gelangt man effizienter ans Ziel. Da die optischen Gesetze invertierbar sind, kann ein Lichtstrahl auch in umgekehrter Richtung verfolgt werden. Dabei reicht es nun aus, nur jene Strahlen rückwärts zu verfolgen, die einen Beitrag zum Bild liefern. Für jeden Rasterpunkt der Bildfläche (Pixel) wird ein Blickstahl vom Augpunkt auf die Szene gelenkt. So entsteht eine perspektivische Darstellung der Szene.

Die Blickstrahlen (Primärstrahlen) werden in der Szene verfolgt. Dazu werden sie mit den Objekten der Szene geschnitten. Existiert kein Schnittpunkt, so ist für dieses Pixel der Hintergrund sichtbar. Ansonsten bestimmt der vorderste, der Bildfläche näheste Schnittpunkt die Farbe und Helligkeit des Bildpunktes. Um festzustellen, ob der Objektpunkt von den Lichtquellen beleuchtet wird, wird zu jeder Lichtquelle ein Strahl (Schattenfühler) gelegt. Schneidet ein solcher Schattenfühler zwischen dem Objektpunkt und der Lichtquelle einen Körper (dies kann auch jener sein, auf dem der Objektpunkt selbst liegt), so liegt der Objektpunkt im Schatten dieses Körpers, andernfalls wird er von der Lichtquelle direkt beleuchtet. Neben den Punktlichtquellen gibt es stets noch Umgebungslicht, das überall leuchtet. Es stellt sicher, daß auch Objekte, die von keiner Punktlichtquelle direkt beleuchtet werden, nicht in totaler Dunkelheit liegen. Für spiegelnde oder das Licht brechende Objekte ist weiters wichtig, welche Farbe jener Körper hat, der durch die Umlenkung der Lichtes an dieser Stelle sichtbar wird. Dazu wird entsprechend den optischen Gesetzen ein Sekundärstrahl in der Szene verfolgt. Wie für den Primärstrahl wird auch für den Sekundärstrahl die Helligkeit des Objektpunktes bestimmt, den er trifft.

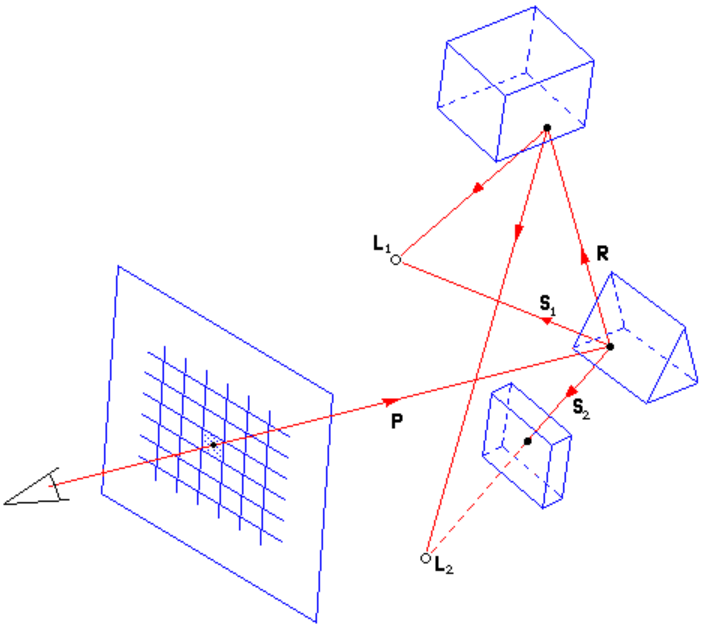


Abb. Ray Tracing: Vom Auge aus werden durch alle Bildpunkte (Pixel) Primärstrahlen (P) in die Szene geschickt und ihr optisches Verhalten simuliert. An den Schnittpunkten mit Objekten werden Schattenfühler (S) zu den Lichtquellen (L) und Reflexionsstrahlen (R) gelegt.

Die Helligkeit, die ein Objektpunkt abstrahlt, ergibt sich nun aus dem Anteil des Lichtes von den Lichtquellen, den er diffus abstrahlt, sowie dem gespiegelten und durchgelassenen Licht.

Für die Modellierung einer Szene im Computer muß für jeden Körper neben der äußeren Gestalt und der Lage im Raum festgelegt werden, welche Farbe und Musterung er hat. Weiters ist die Oberflächenstruktur (porös, glatt, poliert) wichtig, um die Anteile des diffus und spiegelnd reflektierten Lichts korrekt zu bestimmen. Um auch Brechung richtig behandeln zu können, benötigt man für jeden transparenten Körper die Angabe der optischen Dichte und des Absorptionskoeffizienten in seinem Inneren. Weiters müssen die Lichtquellen in ihrer Position und Lichtstärke festgelegt werden. Die Kamera wird durch ihre Lage im Raum, ihre Ausrichtung auf die Szene und die Projektionsart (parallel- oder Zentralprojektion) angegeben.

Obwohl Ray Tracing bereits ein sehr komplexes und rechenintensives Modell der Wirklichkeit ist, gibt es einige grundlegende Probleme. Das wesentlichste Problem besteht darin, daß diffuse Reflexion nicht modelliert wird. Jedes Objekt wird nicht nur durch Lichtquellen beleuchtet, sondern auch durch das von anderen Objekten diffus oder spiegelnd reflektierte Licht. Um den Einfluß dieser Lichtquellen zu ermitteln, müßte in jede Richtung ein Sekundärstrahl gelegt werden. Dies ist extrem rechenintensiv, wird aber dennoch bei der Bildgenerierung nach dem Radiosity-Verfahren ansatzweise versucht. Neben diesem schweren Fehler gibt es auch das Problem, daß die optischen Gesetze nur angenähert werden. So wird in der Regel die Lichtbrechung unabhängig von der Wellenlänge des Lichtes berechnet, was dazu führt, daß die Spektralzerlegung an einem Prisma nicht erkannt wird. Auch die Reflexion an einer spiegelnden Oberfläche ist von der Wellenlänge des Lichts abhängig. Auch für diese Probleme gibt es Speziallösungen, die bei deutlicher Erhöhung des Rechenaufwands bessere Bilder liefern.

3. Epipedbäume

Um ein Bild mit Ray Tracing zu berechnen, ist es nötig, viele Strahlen effizient mit den Objekten einer Szene zu schneiden. Dabei sind die Objekte der Szene über geeignete Verknüpfungen aus Basisobjekten, etwa Würfeln, Kugeln oder Zylindern zusammengesetzt. Um nun einen Strahl mit den Objekten einer Szene zu schneiden, wird der Strahl zunächst mit jedem einzelnen Objekt geschnitten. Aus den so errechneten Schnittpunkten wird dann jener bestimmt, der dem Auge am nächsten ist.

Für einfache Basisobjekte, etwa Würfel, Kugeln oder Kegel, existieren geschlossene Lösungen für die Berechnung aller Schnittpunkte zwischen dem Objekt und einem Strahl.

Für andere Einzelobjekte, etwa Bezier- oder B-Spline-Flächen, oder die hier behandelten allgemeinen Flächen in Parameter-Darstellung gibt es keine geschlossenen Lösungen. Es ist daher erforderlich, für alle Durchstoßpunkte zunächst eine Näherungslösung zu bestimmen und diese dann durch Iteration bis auf die benötigte Genauigkeit zu verfeinern.

Eine allgemeine Fläche kann sehr stark im Raum gewunden sein. Es ist daher nicht einfach möglich, gute erste Näherungen für die Schnittpunkte zwischen der Fläche und einem Strahl anzugeben. Wird allerdings nicht die gesamte Fläche auf einmal betrachtet, sondern nur hinreichend kleine Teilstücke, so zeigt sich, daß jedes Teilstück relativ glatt ist, und durch ein Ebenenstück gut angenähert werden kann.

Um den Schnitt zwischen Strahl und Fläche zu errechnen, reicht es nicht aus, jedes Flächenstück durch ein Ebenenstück zu approximieren und den Schnittpunkt dieser Approximation mit dem Strahl zu berechnen. Sind der Strahl und das Ebenenstück nämlich fast parallel, so wird auf diese Weise oft selbst dann kein Schnittpunkt erkannt, wenn ein Schnittpunkt zwischen dem Strahl und dem Flächenstück existiert. Daher werden die Flächenstücke durch kleine Körper, in deren Inneren das Flächenstück vollständig liegt, umschlossen. Die Körper sollten dabei so beschaffen sein, daß sie einerseits schnell mit einem Strahl geschnitten werden können, andererseits nicht deutlich größer sind als das Flächenstück, das sie umschließen. Die Wahl des Körpers stellt sich damit unter anderem zwischen Parallelepipeden, achsparallelen Quadern und Kugeln. Parallelepipede sind durch affine Transformation verallgemeinerte Quader. Sie verfügen wie Quader über zwölf Kanten, von denen jeweils vier zueinander parallel sind, doch während die Winkel zwischen den aufspannenden Kanten beim Quader stets 90 Grad betragen, können sie beim Parallelepiped beliebige Werte annehmen.

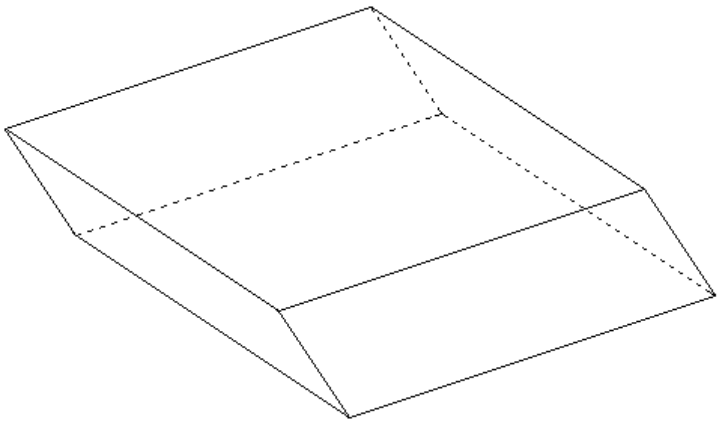


Abb. Parallelepiped

Der Aufwand für den Schnitt zwischen einem Strahl und einem Parallelepiped ist höher als der mit einem achsparallelen Quader oder mit einer Kugel, doch dadurch, daß das Parallelepiped nicht an fixe Achsrichtungen gebunden ist, läßt es sich sehr gut an relativ flache Flächenstücke anpassen. Dabei entsprechen zwei Flächen ('Boden' und 'Deckel') des Parallelepipeds der Näherungsebene des Flächenstücks, und die 'Höhe' des Epipeds richtet sich nach der Wölbung.

Der wesentliche Vorteil des Parallelepipeds besteht darin. unabhängig von einer affinen Transformation (z.B. Drehung, Schiebung, Scherung) von Fläche und Parallelepiped, die Fläche immer gleich gut zu umschließen.

Der Schnitt eines Strahls mit einem Quader kann zwar schneller berechnet werden als mit einem Parallelepiped, und es gibt auch Flächenstücke, die sich gut in einen achsparallelen Quader einschließen lassen, doch gibt es auch Flächenstücke für die ein achsparalleler Quader nur eine sehr schlechte Näherung ist.

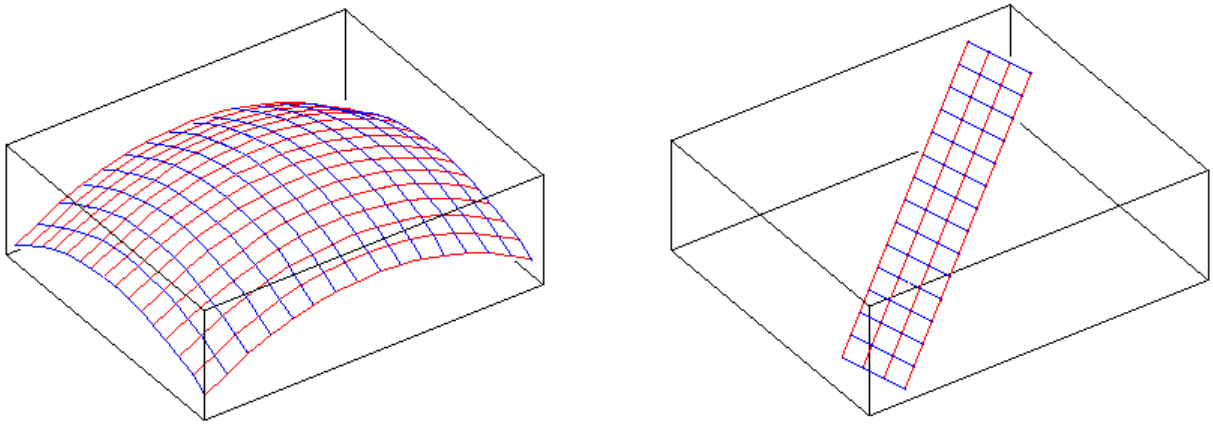


Abb. Je nach Form und Lage des Flächenstücks läßt es sich mehr oder minder eng mit einem achsparallelen Quader umschließen.

Läßt man auch nicht-achsparallele Quader zu, so ist der Schnitt mit einem Strahl nicht mehr schneller als bei einem Parallelepiped. Für den Quader bleibt aber die Beschränkung auf rechte Winkel wirksam. wohingegen beim Parallelepiped die Freiheit in den Winkeln dazu genutzt werden kann, eine engere Umschließung zu finden.

Der Schnitt zwischen einer Kugel und einem Strahl ist zwar sehr schnell möglich. doch ist eine Kugel keine enge Umschließung für ein (fast ebenes) Flächenstück. Eine Erweiterung auf achsparallele Ellipsoide ist zwar noch relativ effizient möglich, doch stößt man damit auf die gleichen Einschränkungen wie bei achsparallelen Quadern. Eine Erweiterung auf allgemeine Ellipsoide ist zu langsam.

Aus diesen Gründen wurden für die vorliegende Implementation Parallelepipede verwendet, um die Fläche zu umschließen.

Die Gesamtfläche wird also in viele kleine, jeweils fast ebene Flächenstücke zerlegt. die von je einem Parallelepiped eingeschlossen werden.

Um einen Schnitt zwischen einem Strahl und der Fläche zu bestimmen, müssen alle Parallelepipede auf einen Schnitt mit dem Strahl hin untersucht werden. Da zu einer allgemeinen Fläche im Regelfall einige 100 bis einige 1000 Parallelepipede gehören. ist es viel zu rechenintensiv, wirklich alle Parallelepipede einzeln zu untersuchen.

Um die Anzahl der untersuchten Parallelepipede zu senken, kann ein Epipedbaum eingesetzt werden. Dabei wird zunächst ein Parallelepiped berechnet, daß die gesamte Fläche einschließt. Dieses Parallelepiped verwendet man als Wurzel für einen Epipedbaum. Es wird im Allgemeinen nicht flach sein. Daher wird die Fläche, die es umschließt, in zwei Teile zerlegt, und für jeden dieser Teile ein umschließendes Parallelepiped erzeugt. Diese beiden Parallelepipede sind dann Nachfolger jenes Parallelepipeds, aus dem sie durch Teilung der Fläche hervorgegangen sind. Die Nachfolger überlappen einander im Allgemeinen und liegen nicht zwingend vollständig innerhalb des Vorgängers. Für jedes der neu erzeugten Parallelepipede wird nun untersucht, ob es bereits hinreichend flach ist. Ist es noch zu dick, so wird die zugehörige Fläche wieder in zwei Teile zerlegt, für die dann wieder Parallelepipede erzeugt werden. Durch rekursive Wiederholung dieses Vorganges wird die Fläche immer feiner zerlegt, bis die umschließenden Parallelepipede hinreichend flach sind.

Dabei entsteht ein binärer Baum, bei dem die Parallelepipede in den Knoten jeder Ebene die gesamte allgemeine Fläche umschließen, und dessen Blattepipede relativ flach sind.

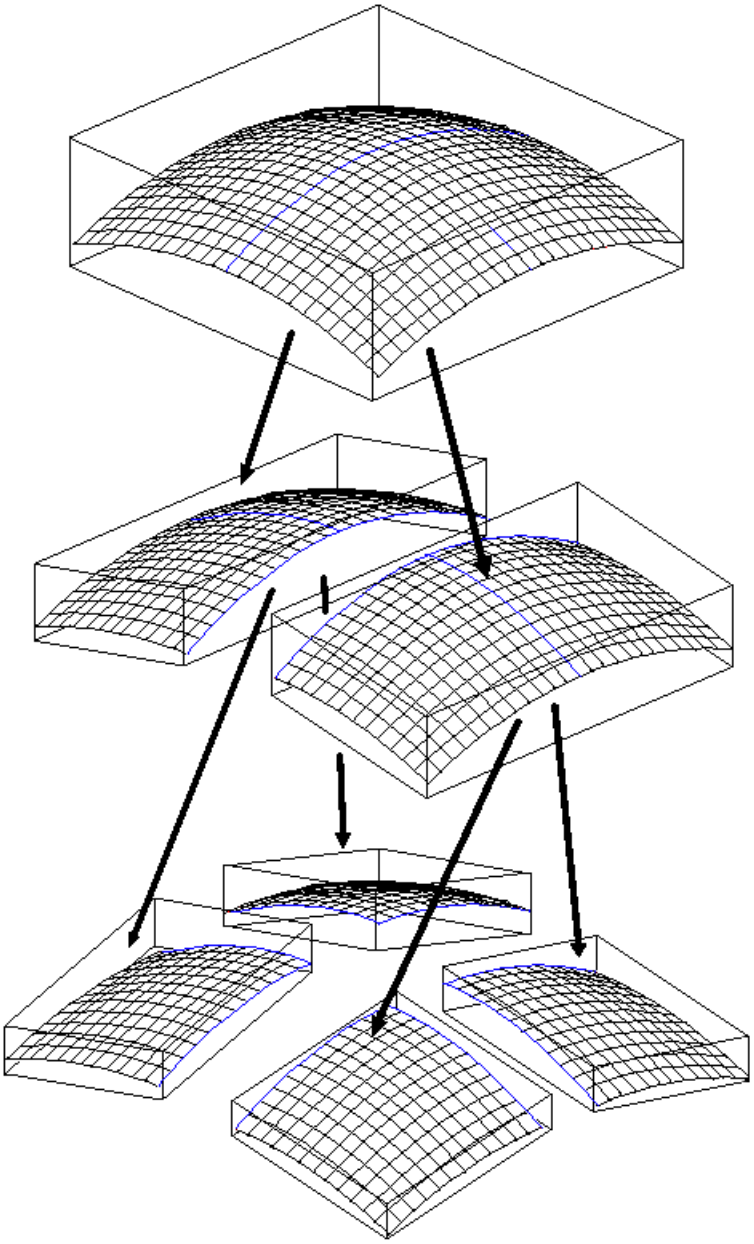


Abb. Zerlegung einer Fläche in fast ebene Teilflächen mit umschließendem Epipedbaum.

Um einen Strahl mit der Fläche zu schneiden, wird zuerst untersucht, ob der Stahl die Wurzel des Epipedbaums durchdringt. Ist dies nicht der Fall, so existiert sicher kein Schnitt des Strahls mit der im Parallelepiped enthaltenen Fläche und jede weitere Berechnung ist unnötig. Liegt hingegen ein Schnittpunkt vor, so werden der linke und rechte Nachfolger des Parallelepipeds rekursiv mit demselben Verfahren untersucht, bis entweder kein Schnitt mehr vorliegt oder die Blattepipede erreicht sind.

Bei den Blattepipeden ist die Fläche bereits so weit zerlegt, daß sie relativ flach ist. Es ist daher möglich, eine erste Näherung des Schnittpunktes zwischen Strahl und Fläche zu erhalten, indem der Strahl mit der Mittelebene des Parallelepipeds geschnitten wird.

Von dieser ersten Näherungslösung ausgehend, kann der Schnittpunkt mittels Newton-Iteration verfeinert werden, bis die nötigen Genauigkeit erreicht ist, oder feststeht, daß der Strahl zwar das Parallelepiped schneidet, nicht aber das darin enthaltene Flächenstück.

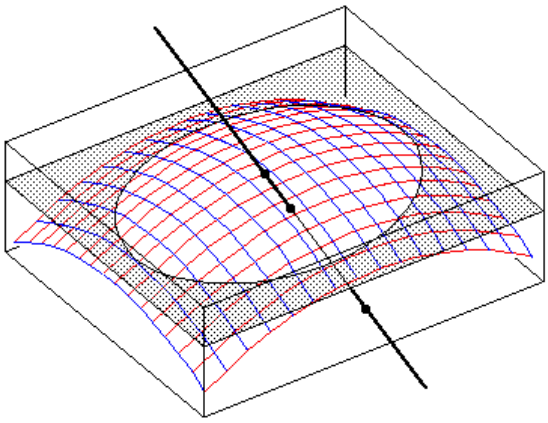


Abb. Erste Näherung des Schnitts Strahl/Fläche durch den Schnitt mit der Zentralebene des Epipeds.

Bisher wurde nicht erwähnt, wie die Epipede des Baumes bestimmt werden. Um ein Parallelepiped festzulegen ist, es erforderlich, einerseits die Richtungen der Begrenzungsebenen vorzugeben, andererseits muß ihre konkrete Lage bestimmt werden.

Eine Möglichkeit, die Ebenenrichtungen festzulegen, wäre, die Richtung der Tangentialebene in einem Punkt der Fläche (etwa in der Mitte des Parameterbereichs) zu verwenden. Wenn das Flächenstück allerdings noch stark gekrümmt ist, so nähert diese Lösung nur die lokale Richtung der Fläche um diesen Punkt, nicht aber die Orientierung des gesamten Flächenstücks an.

Eine bessere Möglichkeit, die Ebenenrichtungen festzulegen, wurde von Michael Schindler für seine Implementierung des Epipedbaums gewählt. Für das Flächenstück über dem Parameterintervall $[u_0, u_1]x[v_0, v_1]$ wird je ein 'mittlerer' Vektor in U- und V Richtung bestimmt.

$$VecU = ((XYZ(u_1,v_0) - XYZ(u_0,v_0)) + (XYZ(u_1,v_1) - XYZ(u_0,v_1))) / 2$$
$$VecV = ((XYZ(u_0,v_1) - XYZ(u_0,v_0)) + (XYZ(u_1,v_1) - XYZ(u_1,v_0))) / 2$$

Diese beiden Vektoren definieren eine Ebenenrichtung, die eine gute Näherung der Richtung des Flächenstücks ist. Somit ist die Richtung von Boden und Deckel des Parallelepipeds festgelegt. Wenn das Flächenstück fast eben ist, dann ist die Höhe des Parallelepipeds sehr klein und die beiden anderen Ebenenrichtungen sind nicht so wichtig. Sie können also so gewählt werden, daß der Rechenaufwand beim Schnitt mit einem Strahl minimiert wird. In der gegebenen Implementierung werden dazu die gleichen Richtungen gewählt, die schon eine Ebene höher im Epipedbaum verwendet wurden, da dann Teilergebnisse von dort weiterverwendet werden können.

Für eine gegebene Ebenenrichtung ist es nun noch erforderlich, die Lage der Begrenzungsebenen festzulegen. Wie diese Aufgabe gut gelöst werden kann, hängt stark von der Art der betrachteten Fläche ab. Für allgemeine Flächen in Parameter-Darstellung wird dieses Problem in Kapitel 5 behandelt.

Diese Beschreibung des Epipedbaums sollte keine exakt nachvollziehbare Lösung vorstellen, sondern nur grob die grundlegenden Ideen zeigen. Eine genauere Beschreibung des Epipedbaums, die auf seine Anwendung auf Bezier- und B-Spline-Flächen und die Probleme der Newton-Iteration eingeht, ist in [BAR90] zu finden. Die dort beschriebenen Algorithmen wurden in den Diplomarbeiten von Kiendl [KIE91] und Groß [GRO9I] implementiert.

4. Formelauswertung

Eine allgemeinen Fläche in Parameter-Darstellung wird durch drei Formeln für X(u,v), Y(u,v) und Z(u,v) (kurz: XYZ(u, v)) und je ein Intervall für die Parameter u und v angegeben. Um einzelne Flächenpunkte bestimmen zu können, werden konkrete Werte für die Parameter u und v eingesetzt und die Formeln damit ausgewertet. Weiters müssen für das Ray Tracing auch die Werte der partiellen Ableitungen nach u und v bestimmt werden.

Dabei stellt sich die Frage, in welcher Form der Anwender dem Ray Tracer seine Formeln am besten zur Verfügung stellt.

4.1 Verfahren zur Formelauswertung

Eine Möglichkeit ist, daß der Anwender ein Programmstück in einer Hochsprache (etwa C, PASCAL etc.) schreibt, die die Berechnung von X, Y und Z mit den partiellen Ableitungen nach u und v vornimmt. Dieser Ansatz setzt voraus, daß der Anwender die Sprache beherrscht und über einen Compiler verfügt. Ein weiterer Nachteil ist, daß die Formeln somit Bestandteile des Programms werden und daher für jedes Bild eine neue Version des Ray Tracers gelinkt werden muß. Sobald mehrere verschiedene Flächen in einem Bild verwendet werden sollen, entstehen außerdem Namenskonflikte, die vom Ray Tracer gelöst werden müßten.

Eine günstigere Möglichkeit, die Formeln auszuwerten besteht darin, dem Ray Tracer die Formeln als Zeichenketten in der Szenendatei zur Verfügung zu stellen und zur Laufzeit zu interpretieren. Diese Möglichkeit hat den Nachteil, daß im Ray Tracer ein komplexer Code für das Auswerten der Formeln existieren muß. Da die Formeln hier interpretiert werden, ist diese Lösung auch etwas langsamer als eine compilierte Auswertung. Da die Formeln nun aber als manipulierbare Daten und nicht mehr als Programmbestandteil vorliegen, ist es kein Problem, ohne Änderungen am Ray Tracer verschiedene Flächen, auch in einem Bild, zu verwenden.

In der vorliegenden Implementation fiel die Entscheidung, die Formeln vom Ray Tracer interpretieren zu lassen. Wie dieses Problem genau gelöst wurde, wird in Abschnitt 6 beschrieben.

Neben den Punktkoordinaten für einen (u,v)-Wert sind für die Newton-Iteration, wie sie zur exakten Bestimmung der Durchstoßpunkte eines Strahls durch die Fläche verwendet wird, auch noch die Tangenten in u- und v- Richtung notwendig.

Numerische Differentiation ist eine sehr einfache Methode, die partiellen Ableitungen zu ermitteln. Dabei wird die Tangente durch eine Sehne, also der Differential- durch einen Differenzenquotienen angenähert. In Zeichen: $(dXYZ/du)(u,v) \approx (XYZ(u+\Delta u,v) - XYZ(u,v))/\Delta u$. Bei diesem Ansatz besteht das Problem, daß für sehr kleine Δu die beiden XYZ(., v) fast gleich sind, einander bei der Subtraktion numerisch weitgehend auslöschen und so kleine Rechenfehler in den X(.,v) starken Einfluß auf das Ergebnis haben. Wählt man Δu allerdings groß, so entsteht ein deutlicher Unterschied zwischen der Sehne und der Tangente der Fläche. Diese Methode ist somit zwar einfach zu programmieren, da außer der Formelauswertung an zwei Stellen kaum Code anfällt, liefert aber ungenaue Ergebnisse.

Wenn man die exakte Tangente berechnen will, muß die Ableitung direkt ausgewertet werden. Somit stellt sich die Frage, woher die Formeln für die Ableitungen gewonnen werden können.

Eine Möglichkeit wäre, daß vom Benutzer neben den Formeln für die Punktkoordinaten auch die Formeln für die Tangenten in der Szenendatei angegeben werden. Die Tangenten können dann mit dem gleichen Formelinterpreter bestimmt werden, der auch die Punktkoordinaten errechnet. Die Berechnung erfolgt schnell und ohne zwingende Ungenauigkeit. Der Nachteil dieses Systems ist, daß der Anwender fehlerfrei differenzieren können muß. Diese Methode ist also einfach zu programmieren, schnell in der Ausführung, aber unkomfortabel für den Benutzer.

Um den Benutzer von der Arbeit der händischen Bestimmung der Tangentenformeln zu befreien, ist es möglich, im Programm aus den Punktformeln durch symbolisches Differenzieren die Formeln für die Ableitungen zu gewinnen. Dabei ist es wichtig, einen gründlichen Optimierer einzusetzen, der gleichartige Ausdrücke erkennt und zusammenfaßt und Teilausdrücke, die mit Null multipliziert werden, wie sie etwa bei der Ableitung des Produkts einer Konstanten mit einem Term entstehen, entfernt. Wenn die Tangentenformel gut optimiert wird, dann ist die

Auswertung ebenso schnell Wie bei händischer Angabe der Formel. Der Aufwand für die Bestimmung der Ableitungen wurde vom Benutzer weg in den Ray Tracer hinein verlagert. Dadurch ist dieses Verfahren zwar schnell und einfach zu benutzen, aber mühsam zu programmieren.

Eine andere Methode, um die Ableitungen zu bestimmen, ist, die Formeln an einer Stelle (u_0,v_0) automatisch zu differenzieren.

4.2 Automatische Differentiation

Bei der automatischen Differentiation wird mit der Originalformel nicht nur der Funktionswert an einer Stelle (u_o,v_o) berechnet, sondern gleichzeitig auch die Werte der partiellen Ableitungen nach u und v an dieser Stelle bestimmt.

Für primitive Ausdrücke läßt sich das Tripel (x, x_u, x_v) aus Funktionswert und partiellen Ableitungen leicht bestimmen:

$u \rightarrow (u_0,1,0) \quad \sim \quad v \rightarrow (v_0,0,1) \quad \sim \quad c \rightarrow (c,0,0)$ c konst.

Aus den Tripeln für einfache Ausdrücke lassen sich die Werte für komplexere, zusammengesetzte Ausdrücke über die Rechenregeln der Ableitungsarithmetik ermitteln:

$(a,a_u,a_v) + (b,b_u,b_v) = (a+b,\ a_u+b_u,\ a_v+b_v)$
$(a,a_u,a_v) - (b,b_u,b_v) = (a-b,\ a_u-b_u,\ a_v-b_v)$
$(a,a_u,a_v) * (b,b_u,b_v) = (a*b,\ a_u*b+a*b_u,\ a_v*b+a*b_v)$
$(a,a_u,a_v) / (b,b_u,b_v) = (a/b,\ a_u/b-a*b_u/b^2,\ a_v/b-a*b_v/b^2)$

Auch Funktionen lassen sich in diesem System berechnen.

$Funct(a,a_u,a_v) = (Funct(a),\ Funct'(a)*a_u,\ Funct'(a)*a_v)$

Etwa: $Sin(a,a_u,a_v) = (Sin(a),\ Cos(a)*a_u,\ Cos(a)*a_v)$

Bsp: Auswertung von Sin(12*U) * (U+V) für $u_o=3$, $v_o=2$

```
Sin(12*U) * (U+V) =
     = Sin(   12    *    U)            * (   U    +    V)    =
     = Sin((12,0,0) * (3,1,0))          * ((3,1,0) + (2,0,1))=
     = Sin((12*3, 0*3+1*12, 0*3+0*12)) *    (3+2, 1+0, 0+1) =
     = Sin(( 36,     12,       0))     *    ( 5,   1,   1)  =
     = (Sin(36), Cos(36)*12, Cos(36)*0)*    ( 5,   1,   1)  =
     = (-0.99178, -1.53556, 0)          *    ( 5,   1,   1)  =
     = (-0.99178*5, -1.53556*5+1*-0.99178, 0*5+1*-0.91178)  =
     = (-4.95889, -8.66960, -0.91178)
```

Dieses Verfahren zur Berechnung der Ableitungen zeichnet sich dadurch aus, daß es die Formeln der Ableitungen nie explizit verwendet, diese also auch nicht vom Benutzer händisch bestimmt werden müssen. Es entsteht kein Approximationsfehler. Ein weiterer Vorteil besteht darin, daß nur je eine Formel für X, Y und Z interpretiert werden muß, um den Funktionswert und die Ableitungen zu bestimmen, also der Aufwand für die Verwaltung relativ gering ist. Daraus folgt die Einschränkung, daß die Ableitungen nicht getrennt vom Funktionswert bestimmt werden können, doch werden von der Newton-Iteration ohnehin immer Funktionswert und Ableitungen gemeinsam benötigt, sodaß diese Eigenschaft kein Nachteil ist. Ein anderer Nachteil ist, daß oft 'tote' Operationen, etwa Multiplikationen mit Null auftreten. Es ist möglich, einige überflüssige Operationen durch Optimierung einzusparen. Verwendet man etwa die Multiplikation mit einer Konstante als zusätzliche Basisoperation, so kann man die Tatsache, daß die Ableitungen der Konstante Null sind, direkt in die Produktregel einfließen lassen.
Also: $c * (a,a_u,a_v) = (c*a,\ c*a_u,\ c*a_v)$ an Stelle von
$(c,c_u,c_v)*(a,a_u,a_v) = (c*a,\ c*a_u + c_u*a,\ c*a_v + c_v*a)$ mit $c_u=c_v=0$

Im Vergleich der Verfahren zeigt sich, daß nur die programminterne symbolische Differentiation und die automatische Differentiation anwenderfreundlich exakte Ergebnisse liefern. Das wesentliche Problem bei der symbolischen Differentiation besteht darin, die Ableitungsformeln gut zu optimieren. Selbst wenn dies gelingt, bleibt immer noch der Nachteil, für Funktionswert und Ableitungen insgesamt drei Formeln interpretieren zu

müssen. Bei der hier verwendeten automatischen Differentiation muß nur eine Formel interpretiert werden, dafür bleiben immer tote Operationen übrig, die bei der symbolischen Ableitung von einem guten Optimierer erkannt werden könnten.

Da moderne Mikroprozessoren Gleitkomma Operationen (fast) ebenso schnell ausführen können wie Verwaltungsoperationen, fiel die Entscheidung in der vorliegenden Implementation zugunsten der automatischen Differentiation, die zwar mehr Gleitkomma Operationen, aber weniger Verwaltungsaufwand benötigt.

Eine Beschreibung all dieser Möglichkeiten zur Bestimmung der Ableitung ist in [MIT91] zu finden.

4.3 Intervall-Arithmetik

Durch Intervall-Arithmetik lassen sich Schranken für den Funktionswert einer Formel für ein gegebenes Parameterintervall berechnen. Dabei wird die Formel nicht für konkrete Werte der Parameter, sondern für Intervalle ausgewertet.

Operationen zwischen Intervallen werden so ausgeführt, daß die Operation auf alle Elementpaare aus den Intervallen ausgeführt werden, und als das Ergebnisintervall das engste Intervall verwendet wird, in dem alle Ergebnisse der Operationen auf den Elementen liegen.

$$X = [X_{MIN} .. X_{MAX}] \sim Y = [Y_{MIN} .. Y_{MAX}]$$
$$X \text{ op } Y = \{x \text{ op } y \mid x \in X, y \text{ v } Y\}$$

Analog wird eine unäre Funktion auf ein Intervall angewendet, indem die Funktion auf alle Werte des Quellintervalls angewendet wird, und als Ergebnisintervall das engste Intervall gilt, das alle Funktionswerte umfaßt.

$$X = [X_{MIN} .. X_{MAX}]$$
$$Funct(X) = \{Funct(x) \mid x \in X\}$$

Natürlich ist es nicht möglich, für alle Elemente von X und Y die Funktion auszuwerten und so Stück für Stück die Elemente des Ergebnisintervalls zu ermitteln. Für alle üblichen Funktionen und Operatoren ist dies auch nicht erforderlich, wenn man nur die Grenzen des Intervalls ermitteln will. So lassen sich für die vier Grundrechnungsarten folgende, einfache Rechenregeln aufstellen:

$$[X_{MIN} .. X_{MAX}] + [Y_{MIN} .. Y_{MAX}] = [X_{MIN}+Y_{MIN} .. X_{MAX}+Y_{MAX}]$$
$$[X_{MIN} .. X_{MAX}] - [Y_{MIN} .. Y_{MAX}] = [X_{MIN}-Y_{MAX} .. X_{MAX}-Y_{MIN}]$$
$$[X_{MIN} .. X_{MAX}] * [Y_{MIN} .. Y_{MAX}] =$$
$$= [Min(X_{MIN}*Y_{MIN}, X_{MIN}*Y_{MAX}, X_{MAX}*Y_{MIN}, X_{MAX}*Y_{MAX}) ..$$
$$Max(X_{MIN}*Y_{MIN}, X_{MIN}*Y_{MAX}, X_{MAX}*Y_{MIN}, X_{MAX}*Y_{MAX})]$$
$$[X_{MIN} .. X_{MAX}] / [Y_{MIN} .. Y_{MAX}] =$$
$$= [-\infty .. +\infty] \text{ falls } Y_{MIN} \le 0 \le Y_{MAX}$$
$$= [X_{MIN} .. X_{MAX}] * [1.0/Y_{MAX} .. 1.0/Y_{MIN}] \text{ sonst}$$

Auch für andere Funktionen lassen sich einfache Regeln verwenden:

$$Sin([X_{MIN} .. X_{MAX}]) = [Y_{MIN} .. Y_{MAX}] \text{ mit}$$
$$Y_{MIN} = -1 \text{ falls } X_{MIN} \le 2\pi*N - \pi/2 \le X_{MAX} \text{ für ein } N \in Z$$
$$= Min(Sin(X_{MIN}), Sin(X_{MAX})) \text{ sonst}$$
$$Y_{MAX} = +1 \text{ falls } X_{MIN} \le 2\pi*N + \pi/2 \le X_{MAX} \text{ für ein } N \in Z$$
$$= Max(Sin(X_{MIN}), Sin(X_{MAX})) \text{ sonst}$$

$$Abs([X_{MIN} .. X_{MAX}]) =$$
$$= [-X_{MAX} .. -X_{MIN}] \quad \text{falls } X_{MAX} \le 0$$
$$= [0 .. Max(-X_{MIN}, X_{MAX})] \quad \text{falls } X_{MIN} \le 0 \le X_{MAX}$$
$$= [X_{MIN} .. X_{MAX}] \quad \text{falls } 0 \le X_{MIN}$$

Diese Rechenregeln führen zu den gleichen Ergebnissen wie jene über den Mengen, haben aber den Vorteil, schnell ausführbar zu sein.

Komplexere Ausdrücke lassen sich auswerten, indem man sie auf Grundoperationen zurückführt und diese dann einzeln ausführt.

Dazu ein Beispiel: `Sin(U)*(U+V)` für `U=[0..1]` und `V=[0..0.4]`

```
Sin(U) * (U + V) = Sin([0 .. 1]) * ([0 .. 1] + [0 .. 0.4]) =
                 = [0 .. 0.8415] * ([0 .. 1.4]) =
                 = [0 .. 1.1781]
```

Bei diesem Beispiel liefert die Intervall-Arithmetik bei schrittweiser Auswertung der Formeln jenes Ergebnis, das man auch erhalten hätte, wenn man für alle $u \in U$ und $v \in V$ in den ganzen Ausdruck eingesetzt hätte und das Maximum und Minimum der Funktionswerte bestimmt hätte. Leider ist das nicht bei allen Funktionen der Fall.

Dazu einige Beispiele: `U=[-1..2], V=[-1..2]`

```
.) U - V = [-1 .. 2] - [-1 .. 2] = [-3 .. 3]
   U - U = [-1 .. 2] - [-1 .. 2] = [-3 .. 3] >> [0 .. 0] !!
.) U * V = [-1 .. 2] * [-1 .. 2] = [-2 .. 4]
   U * U = [-1 .. 2] * [-1 .. 2] = [-2 .. 4] >> [0 .. 4] !!
   Sqr(U)= Sqr([-1 .. 2]) = [0 .. 4]
```

Sobald die einzelnen Teile eines Ausdrucks nicht mehr voneinander unabhängig sind, besteht die Gefahr, daß das Ergebnisintervall zu groß wird. Dies rührt daher, daß bei der Verknüpfung von zwei Intervallen angenommen wird, daß alle möglichen Wertepaare aus den Intervallen tatsächlich auftreten, und daher im Ergebnis zu berücksichtigen sind. Wenn die beiden Intervalle allerdings voneinander abhängig sind, so werden nicht alle theoretisch möglichen Wertepaare realisiert und das Ergebnisintervall wird zu groß. Da stets eine 'Worst Case' Abschrankung errechnet wird, kann das Intervall nie zu klein werden.

Das Problem, daß Intervalle zu groß ausfallen, ist in der Intervall-Arithmetik fest verankert und kann nicht grundsätzlich gelöst werden. Man kann nur versuchen, die Auswirkungen so gut wie möglich in den Griff zu bekommen. Dazu ist es etwa möglich, die Formeln symbolisch zu optimieren und Auslöschungen der Form X-X symbolisch durch die Konstante Null zu ersetzen. Auch Ausdrücke der Form X*X lassen sich durch die Funktion Sqr(X) ersetzen. Dadurch wird explizit ausgedrückt, daß X mit sich selbst multipliziert wird, und nicht mit irgendeinem anderen Intervall. Dieses explizite Wissen kann dann für eine exaktere Auswertung genutzt werden. Natürlich gibt es zu viele unterschiedliche Möglichkeiten, ein Intervall über verschiedene Zwischenstufen mit sich selbst zu verknüpfen, um alle erkennen und explizit darstellen zu können. Daher wird diese Optimierung nur für die wichtigen Verknüpfungen möglich sein, und die anderen Möglichkeiten werden weiterhin das Problem zu großer Intervalle in sich tragen.

Bisher wurde nur die Bestimmung von Funktionswerten mit Hilfe der Intervall-Arithmetik behandelt. In Kapitel 4.2 wurde erläutert, wie gleichzeitig zur Bestimmung der Funktionswerte auch die Werte der partiellen Ableitungen mitbestimmt werden können. Es ist möglich, diese beiden Verfahren zu kombinieren, indem man alle Operationen über reellen Zahlen aus der automatischen Differentiation durch ihre Äquivalente in der Intervall- Arithmetik ersetzt. Dadurch erhält man Intervalle für den Funktionswert und für die partiellen Ableitungen. Die primitiven Ausdrücke werden dabei wie folgt ausgewertet:

$$u \rightarrow ([u_0..u_1],[1..1],[0..0]) \sim v \rightarrow ([v_0..v_1],[0..0],[1..1])$$
$$c \rightarrow ([c..c],[0..0],[0..0]) \quad c \text{ konst.}$$

Die Rechenregeln für zusammengesetzte Ausdrücke sind die gleichen, wie bei der Auswertung für reelle Zahlen.

$$(a,a_u,a_v) + (b,b_u,b_v) = (a+b,\ a_u+b_u,\ a_v+b_v) \quad \text{(etc. vgl. Kapitel 4.2)}$$

Allerdings sind die Komponenten a, a_u, a_v sowie b, b_u, b_v jetzt keine reellen Zahlen mehr, sondern Intervalle.Die Operationen sind daher auch keine Operationen auf reellen Zahlen sondern Operationen auf Intervallen, die mit den oben beschriebenen Regeln auf Operationen über reellen Zahlen zurückgeführt werden können.

Weitere allgemeine Betrachtungen zum Thema Intervall-Arithmetik finden sich in [ALE74]. Mögliche Anwendungen in der Computergrafik sind in [MIT90] und [MIT91] beschrieben.

5. Enge Epipede um Flächenstücke

Um einen Epipedbaum aufzubauen, ist es erforderlich, Flächenstücke möglichst eng in Parallelepipede einzuschließen. Jedes Epiped wird durch die Richtung seiner Begrenzungsebenen und deren Lage definiert. Eine Möglichkeit, die Richtung der Epipedseiten geschickt zu wählen, wurde bereits in Kapitel 3 erwähnt. Um ein Parallelepiped vollständig zu bestimmen, muß somit nur noch die Position der Seitenflächen festgelegt werden. Das Problem besteht also darin, zwei Ebenen zu finden, die parallel zu einer vorgegebenen Ebenenrichtung sind, und zwischen denen alle Punkte eines Flächenstückes liegen. Andererseits sollten die Ebenen die Fläche möglichst eng umschließen, also nicht weiter auseinander liegen als unbedingt nötig, da ansonsten der Rechenaufwand für die Suche im Epipedbaum unnötig hoch wird.

Eine theoretische Möglichkeit wäre, alle Punkte des Flächenstücks zu berechnen und jene Ebenen zu wählen, die die Punktwolke einschließen. Dieser Lösungsweg liefert zwar das optimale Ergebnis, da aber unendlich viele Punkte berechnet werden müßten, ist er praktisch nicht einsetzbar. Eine Variation dieses Algorithmus, bei der die Fläche nur an endlich vielen Stellen abgetastet wird, läuft immer Gefahr, eine einzelne Spitze zwischen den abgetasteten Werten zu übersehen.

Eine andere Möglichkeit besteht darin, einen (evt. komplizierteren) Körper zu finden, der das betrachtete Flächenstück umschließt und diesen Körper dann mit einem Parallelepiped zu umschließen. Die Genauigkeit, mit der das Parallelepiped die Fläche umschließt, hängt dabei wesentlich von der Wahl des Körpers ab. Verschiedene Möglichkeiten einen solchen Körper für Flächenstücke in Parameterdarstellung auszuwählen werden im folgenden erläutert.

5.1 Umschließende Körper für Flächenstücke

Mittels Intervall-Arithmetik (vgl. Kapitel 4.3) ist es möglich, durch direktes Auswerten der XYZ(u,v) Formeln Schranken für die Funktionswerte X, Y und Z in einem (U, V) Bereich zu berechnen. Diese Schranken liefern direkt einen achsparallelen Quader, der die Fläche umschließt.

Leider wird bei dieser Methode die Hauptrichtung der Fläche überhaupt nicht berücksichtigt, sondern stets ein achsparalleler Quader als Umhüllung verwendet. Wenn das Flächenstück also nicht fast normal auf eine der drei Koordinatenachsen ist, so führt dieser Ansatz zu sehr schlechten Resultaten.

Eine andere Möglichkeit ist, neben den Funktionswerten auch die Steigung der Fläche in Betracht zu ziehen.

Da alle Parameterfunktionen XYZ(u,v) auf dem betrachteten Intervall $[u_{MIN}...u_{MAX}]x[v_{MIN}...v_{MIN}]$ stetig und differenzierbar sind, gilt der Mittelwertsatz der Differentialrechnung

$$f(t) = f(t_0) + (t-t_0) * f'(\tau) \quad \text{für ein } \tau \text{ zwischen } t_0 \text{ und } t \tag{1}$$

Wenn man für die Ableitung eine obere und untere Schranken kennt, so gilt

$$f(t) \in f(t_0) + (t-t_0) * [f'_{MIN}...f'_{MAX}] \tag{2}$$

für alle t aus dem betrachteten Intervall

Zunächst betrachten wir ein ebenes Kurvenstück

$$X(u) \quad Y(u) \quad \text{mit } u \in [u_{MIN}...u_{MAX}] \tag{3}$$

Nach obigen Überlegungen gilt daher

$$X(u) \in X(u_0) + (u-u_0) * [X'_{MIN}...X'_{MAX}] \tag{4}$$
$$Y(u) \in Y(u_0) + (u-u_0) * [Y'_{MIN}...Y'_{MAX}]$$

Daraus lassen sich leicht Schranken für X und Y über das gesamte U-Intervall ermitteln:

$$X([u_{MIN}...u_{MAX}]) \in X(u_0) + ([u_{MIN}...u_{MAX}] - u_0) * [X'_{MIN}...X'_{MAX}] \tag{5}$$
$$Y([u_{MIN}...u_{MAX}]) \in Y(u_0) + ([u_{MIN}...u_{MAX}] - u_0) * [Y'_{MIN}...Y'_{MAX}]$$

Die so erhaltenen Schranken für X und Y, können nun wieder dazu verwendet werden eine achsparallele Umschließung der Kurve zu berechnen. Leider sind dadurch die Probleme, die sich durch die starre Wahl der umschließenden Richtungen ergeben, nicht beseitigt.

Bisher wurde jede Dimension getrennt betrachtet. Es ist besser, wenn alle Richtungen gemeinsam betrachtet und Abhängigkeiten zwischen den einzelnen Dimensionen ausgenutzt werden. Dadurch ist es möglich. der Grundrichtung der Kurve besser zu folgen.

Dazu wird ein Punkt u_0 aus dem Parameterintervall gewählt (etwa in der Mitte) und die Einschließung für positives und negatives $(u-u_0)$ getrennt durchgeführt.

Der Endpunkt $X(u_{MAX}),Y(u_{MAX})$ des Kurvenstücks liegt in jenem Intervallpaar, das man durch Einsetzen von $u=u_{MAX}$ in beiden Gleichungen (4) und Auswertung mit Intervall-Arithmetik erhält. Diese Intervalle definieren ein Rechteck (siehe Abb.). Jeder Kurvenpunkt mit $u_0 < u < u_{MAX}$ liegt in dem vom Punkt $X(u_0),Y(u_0)$ ausgehenden Winkelbereich, der durch das Rechteck begrenzt wird. Analog läßt sich ein entsprechender Winkelbereich für den Parameterbereich $u_{MIN} < u < u_0$ erstellen. Daraus folgt aber, daß jede Fläche, die die beiden Winkelbereiche enthält sicher die Kurve umschließt. Jede konvexe Fläche, die die 2*4 Eckpunkte der Rechtecke enthält. enthält auch die Winkelbereiche und damit das Kurvenstück. Dies gilt insbesondere für Parallelogramme.

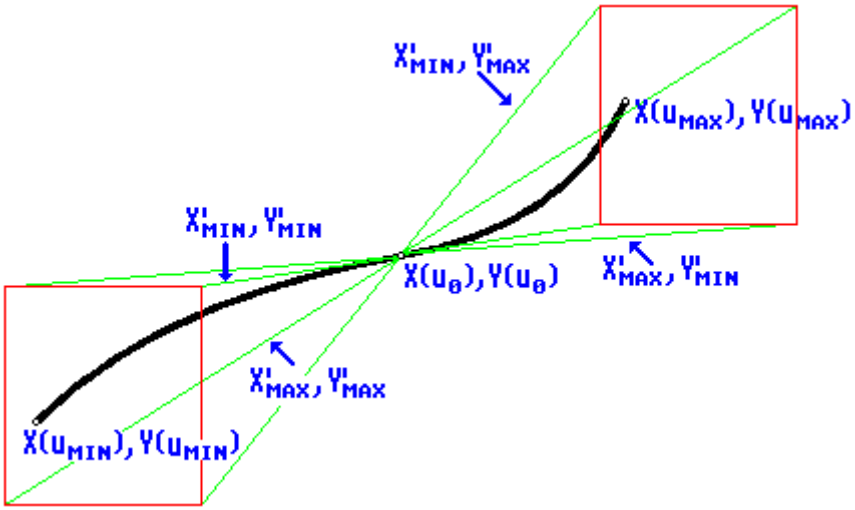


Abb.: Zwei Winkelbereiche zwischen einem Punkt und je einem Rechteck umschließen ein Kurvenstück.

Wenn wir nun zu Kurven im Raum übergehen, so kommt eine dritte Einschließung, für Z(u), hinzu. Aus den beiden Rechtecken werden zwei Quader, aus den Winkelbereichen werden Pyramidenbereiche, und jeder konvexe Körper, der die Eckpunkte der Quader enthält ist eine konvexe Umschließung der Kurve; insbesondere auch jedes entsprechende Parallelepiped.

Schließlich betrachten wir Flächenstücke XYZ(u,v). Die Einschließungen lauten:

$$\begin{aligned} X(u,v) &\in X(u_0,v_0) + (u-u_0) * [X^U_{MIN}...X^U_{MAX}] + (v-v_0) * [X^V_{MIN}...X^V_{MAX}] \\ Y(u,v) &\in Y(u_0,v_0) + (u-u_0) * [Y^U_{MIN}...Y^U_{MAX}] + (v-v_0) * [Y^V_{MIN}...Y^V_{MAX}] \\ Z(u,v) &\in Z(u_0,v_0) + (u-u_0) * [Z^U_{MIN}...Z^U_{MAX}] + (v-v_0) * [Z^V_{MIN}...Z^V_{MAX}] \end{aligned} \quad (6)$$

(X^U,X^V... sind die partiellen Ableitungen)

Durch Einsetzen von u_{MAX} für u und v_{MAX} für v und Auswerten von (6) mit Intervall-Arithmetik erhält man einen Begrenzungsquader in dessen Pyramidenbereich alle Flächenpunkte für $u_0 < u < u_{MAX}$ und $v_0 < v < v_{MAX}$ liegen. Ein zweiter Quader mit Pyramidenbereich entsteht analog für $u_0 < u < u_{MAX}$ und $v_{MIN} < v < v_0$. Schließlich müssen auch noch die beiden anderen Teilparameterbereiche behandelt werden, sodaß schließlich 4 Begrenzungsquader entstehen. Somit gilt:

> Jeder konvexe Körper, insbesondere jedes Parallelepiped, der alle 4*8 Eckpunkte der 4 Begrenzungsquader enthält, schließt das gegebene Flächenstück ein.

Um ein Parallelepiped vollständig zu bestimmen, sind also folgende Schritte erforderlich. Zunächst werden die drei Ebenenrichtungen der Epipedseiten festgelegt (vgl. Kapitel 3). Weiters werden Körper erstellt, die das Flächenstück umschließen. Nun werden jeweils gegenüberliegende Epipedseiten so weit verschoben, daß sie die Körper (möglichst scharf) einschließen.

5.2 Effizienz verschiedener Verfahren

Um nun entscheiden zu können, ob es besser ist, einen achsparallelen Quader oder ein an die Flächenrichtung angepaßtes Objekt als Hüllkörper zu verwenden, werden hier Beispiele mit Funktionen vom Typ Y(X) betrachtet, doch lassen sich die Resultate leicht auf den dreidimensionalen Fall verallgemeinern.

Als Beispielfunktion dient hier $Y(X) = X*(X^2-1)$. Zunächst soll eine konvexe Hülle für den Fall gefunden werden, daß X aus dem Intervall [-1.25...1.25] stammt.

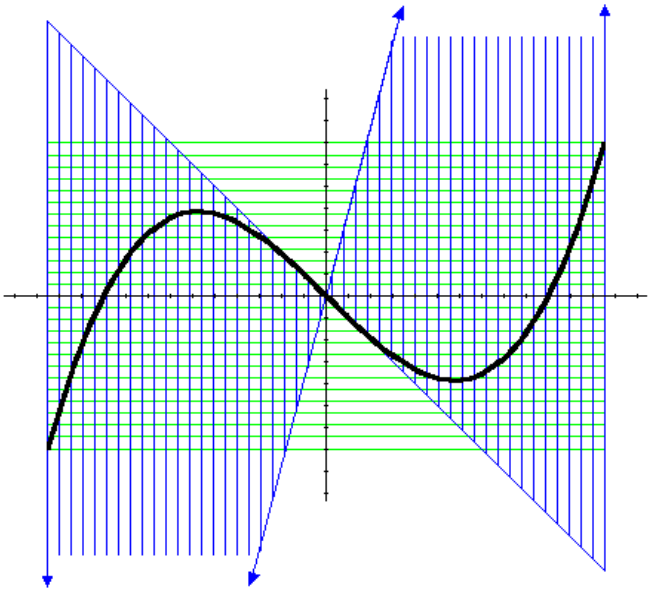


Abb. $Y(X) = X*(X^2-1)$ für $X \in [-1.25...1.25]$

Es zeigt sich, daß die achsparallele Hülle wesentlich enger ist als jene, die die Ableitungen berücksichtigt. Nun die gleiche Funktion auf dem Intervall [-1/2...1/2].

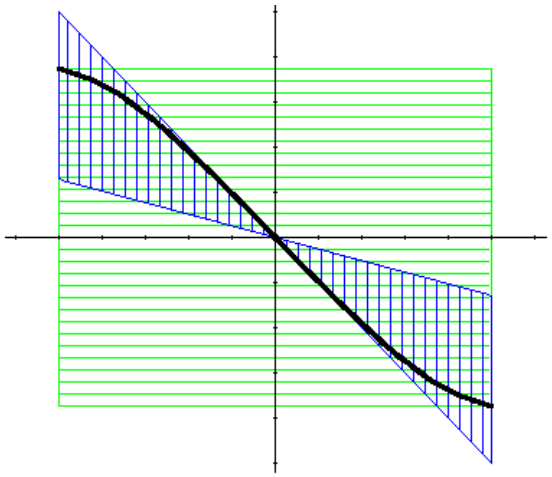


Abb. $Y(X) = X*(X^2-1)$ für $X \in [-1/2...1/2]$

Hier ist der achsparallele Quader wesentlich zu groß, während jener Hüllkörper, der mit den Schranken für die Ableitungen errechnet wurde, gut paßt.

Allgemein läßt sich sagen, daß Kurven, die starke lokale Krümmungen aufweisen, durch eine achsparallele Hülle genauer umschlossen werden, wohingegen lokal schwach gekrümmte Kurven von der Berücksichtigung der Ableitung profitieren.

Zu Beginn dieses Kapitels wurde festgehalten, daß die optimalen Schranken dadurch bestimmt sind, daß alle Flächenpunkte verwendet werden. Es ist nicht möglich, alle Punkte zu verwenden, doch wird die konvexe Hülle, die über die Ableitungen errechnet wird, genauer, wenn statt einer Hülle über das gesamte Flächenstück mehrere kleine Hüllen errechnet werden, die je einen Teil der Fläche umschließen. Die Vereinigung dieser kleinen konvexen Hüllen umfaßt dann die Gesamtfläche.

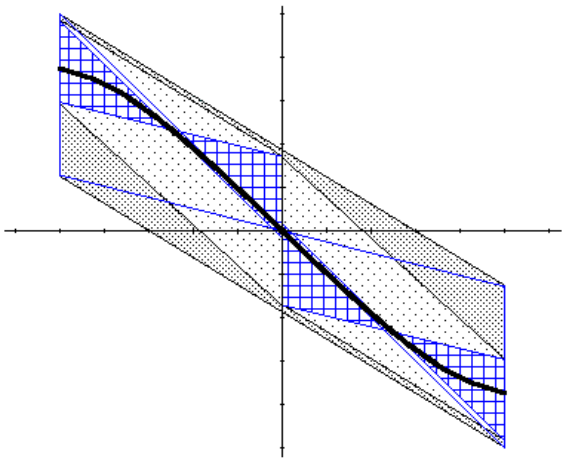


Abb. Eine konvexe Hülle über das gesamte Intervall (dunkelgrauerBereich) bzw.
Zwei kleine Hüllen über je das halbe Intervall. (hellgrauer Bereich)

Im Rahmen dieser Diplomarbeit wurde folgende Lösung gewählt: Es wird stets sowohl ein achsparalleler Quader, als auch ein der Flächenrichtung angepasster Hüllkörper berechnet. Für jeden dieser Körper wird dann ein Parallelepiped konstruiert, daß ihn eng umschließt. Das Ergebnisepiped ist der Durchschnitt der beiden Epipede. Es ist dann mindestens so gut wie jede der beiden Lösungen für sich. oft sogar besser.

Wenn das Flächenstück in einer Dimension sowohl nach U als auch nach V monoton ist, so liegen die Extremwerte für diese Dimension in den Eckpunkten. Sie können also exakt berechnet werden und definieren in dieser Dimension den minimalen Hüllquader. Für alle übrigen Dimensionen werden zunächst mit Intervall-Arithmetik auf den Formeln der Fläche für jede Dimension einzeln Schranken für den Funktionswert in dieser Richtung bestimmt. Weiters werden nach Formel (5) aus Kapitel 5.1 über die Schranken der partiellen Ableitungen Schranken für die Funktionswerte bestimmt. Der Durchschnitt der so erhalten Schranken definiert die übrigen Dimensionen des Hüllquaders.

Der über die Ableitungen definierte Hüllquader schließt sicher den gesamten Pyramidenkörper ein. Es wirkt daher auf den ersten Blick unnütz ihn zu berechnen, da es scheint, daß jede Verbesserung der Schranken auch durch den Pyramidenkörper erbracht werden könnte. Dadurch, daß dieser Quader aber noch mit dem anderen geschnitten wird und in manchen Dimensionen durch Monotonie der Fläche optimiert wird, lassen sich Fälle konstruieren, in denen dieser Quader Vorteile bringt.

Zur Berechnung eines Hüllkörpers, der der Flächenrichtung angepaßt ist, wird das in Kapitel 5.1 beschriebene Verfahren der Pyramidenkörper eingesetzt. Dazu werden zunächst für den Parameter-Bereich Schranken für die partiellen Ableitungen bestimmt. Der Parameterbereich wird dann je einmal in U und V Richtung in der Mitte unterteilt, und für jeden dieser Teile werden die Koordinaten des Mittelpunktes berechnet. Ausgehend von jedem dieser Punkte wird nun ein Pyramidenkörper erstellt, der nur ein Viertel des Flächenstücks umfaßt. Der gesamte

Flächenteil liegt dann in der Vereinigung der vier Körper. Dadurch, daß man den Parameterbereich in vier Teile zerlegt, statt einen Pyramidenkörper über dem gesamte Flächenstück zu errichten, erhält man engere Umschließungen. Andererseits fällt nicht viel mehr Rechenarbeit an. Da für alle vier kleinen Pyramidenkörper die Schranken für die Ableitungen über das ganze Intervall verwendet werden, muß nur einmal mit (relativ) langsamer Intervall-Arithmetik gerechnet werden. Weiters sind die kleinen Pyramidenbereiche doch offensichtlich bis auf Verschiebung ident. Um nun Ebenen zu finden, die das Flächenstück einschließen, werden zunächst die vier Mittelpunkte mit zwei Hilfsebenen eng umgeben. Weiters sucht man den maximalen Abstand, den ein Eckpunkt des Pyramidenkörpers von einer Ebene hat, die durch den Mittelpunkt des Körpers geht, und die Richtung der Epipedseite aufweist. Aus Symetriegründen müssen hier nur 2*8 der 4*8 Quaderecken untersucht werden. Die umschließenden Ebenen erhält man nun, indem man die beiden Hilfsebenen um die Distanz nach außen verschiebt. Der Zusatzaufwand für die Verwendung der vier Pyramidenkörper besteht also nur in der Berechnung von vier statt einem Punkt und ihrer Einschließung zwischen zwei Ebenen. Dafür erhält man fast den Vorteil, den man durch zwei weitere Zerlegungen des Parameterbereichs erhalten hätte. Natürlich könnte man den Parameterbereich auch in mehr als vier Teile zerlegen und dadurch die Genauigkeit der Einschließung zu steigern, doch steigt dadurch auch der Rechenaufwand.

6. Interpretation von Formeln

Bisher wurde stets davon ausgegangen, daß es möglich ist, Formeln, die durch Zeichenketten gegeben sind, auszuwerten. Nun soll untersucht werden, wie dies effizient zu realisieren ist.

Es wird grundsätzlich in vier Schritten vorgegangen. Im ersten Schritt wird die Zeichenkette der Formel in einen Parse-Baum, dessen innere Knoten Operationen und dessen Blattknoten Konstante und Variable sind, übergeführt. Im zweiten Schritt wird dieser Baum optimiert. Anschließend wird der Baum noch in einen Postfix-Ausdruck umgewandelt. Der vierte Schritt besteht dann darin, den Postfix-Ausdruck für konkrete Werte der Variablen auszuwerten. Wegen des einfachen Aufbaus von Postfix-Ausdrücken ist dies sehr schnell möglich.

Offensichtlich sind die ersten drei, relativ langsamen Schritte unabhängig von den konkreten Werten der Variablen und müssen daher nur einmal zu Beginn des Programms ausgeführt werden. Nur die schnelle Auswertung des Postfix-Ausdrucks muß für jeden Satz von Variablen neu ausgeführt werden. Da bei Ray Tracing jede Formel sehr oft (einige 10.000 - 1.000.000 mal) ausgewertet werden muß, ergibt sich durch die Vorverarbeitung eine große Einsparung an Rechenzeit.

Eine ausführliche Darstellung zu diesem Themenkreis befindet sich in [AHO86].

6.1 Erstellung von Parse-Bäumen

Im ersten Schritt der Formelinterpretation wird der als Zeichenkette gegebene Benutzerausdruck in einen Parse-Baum umgewandelt.

In einem Parse-Baum entspricht jeder Konstanten oder Variablen ein Blattknoten. Operatoren werden zu inneren Knoten des Baums. Die Nachfolger der Operatorknoten ergeben sich aus den Operanden. Durch diese direkte Zuordnung von Operanden zu ihren Operatoren ist es nicht nötig, Klammem explizit darstellen zu können. Auch die Präzendenzregeln der Operatoren (Punkt-vor-Strich) sind in der Hierarchie des Baumes direkt berücksichtigt.

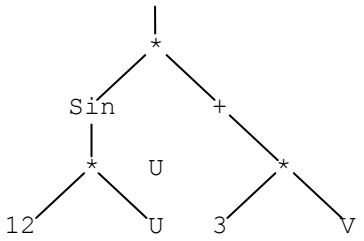


Abb. Parse-Baum für Sin(12*U) * (U + 3*V)

Um aus einer Zeichenkette einen Parse-Baum zu generieren, wird eine Grammatik benötigt, in der der Aufbau von gültigen mathematischen Ausdrücken definiert ist. Eine kleine Grammatik wäre etwa:

```
  Expr = Term [(+ | -) Term]
  Term = Faktor [(* | /) Faktor]
Faktor = Const | Var | (Expr) | Funct(Expr)

 Const = reelle Zahl
 Var   = U | V
 Funct = Sin | Cos | Sqr | Sqrt ...
```

Abb. Eine Grammatik für mathematische Ausdrücke

Bei dieser Grammatik sind Terminalsymbole fett geschrieben. Der senkrechte Strich '|' steht für die exklusive Auswahl. Ausdrücke in eckigen Klammem können beliebig oft, auch kein Mal, verwendet werden. Dies ist nur ein kleiner Ausschnitt aus der für diese Diplomarbeit verwendeten Grammatik. Diese ist vollständig in Kapitel 7.7 wiedergegeben.

Diese Grammatik läßt sich sehr einfach in ein Programm umwandeln. Dabei wird für jede Regel eine Prozedur geschrieben, die Prozeduren der anderen Regeln rekursiv aufruft und ihren Teil des Ableitungsbaumes erzeugt.

Als Beispiel soll hier die Prozedur für die Regel von EXPR stehen:

```
PARSENODE *expr(char **equ)
 {
  PARSENODE node;
  node.u.left = term(equ);
  while ((**equ == '+') || (**equ == '-'))
   {
    if (**equ == '+')
      node.op = ADD;
    else
      node.op = SUB;
    (*equ)++;
    node.u.right = term(equ);
    node.u.left = new_node(&node);
   }
  return(node.u.left);
 } /* Expr */
```

Abb. Eine C-Funktion zur Abarbeitung der Regel EXPR = TERM [(+|-) TERM]

Bei diesem Codesegment ist vor allem die Funktion für das Anlegen des neuen Knotens interessant. Programmtechnisch ist es am einfachsten, wenn der neue Knoten im Systemspeicher angelegt wird. Allerdings existiert dann keine Verbindung zwischen gleiche Knoten. Dies ist offensichtlich eine Vergeudung von Speicherplatz und führt, wie sich später zeigen wird, zu einer Verschwendung von Rechenzeit. Es ist besser, wenn alle Knoten in einer zentralen Datenstruktur, etwa einer (Hash-)Tabelle verwaltet werden. Wenn nun ein neuer Knoten angefordert wird, kann mit der zentralen Datenstruktur erkannt werden, ob der Knoten bereits existiert. Bei einem Treffer wird ein Verweis auf den bereits existierenden Knoten zurückgeliefert, ansonsten wird ein neuer Knoten erstellt und in die zentrale Datenstruktur eingetragen.

Um den Unterschied zu verdeutlichen, sollen hier als Beispiel die Formeln des Spiralkegels dienen:

```
X(u,v) = Sin(12*u) * (u+v)
Y(u,v) = Cos(12*u) * (u+v)
Z(u,v) = (u+v) * -2.5
```

Werden die Parse-Bäume voneinander unabhängig im Speicher angelegt, so ergibt sich folgendes Bild:

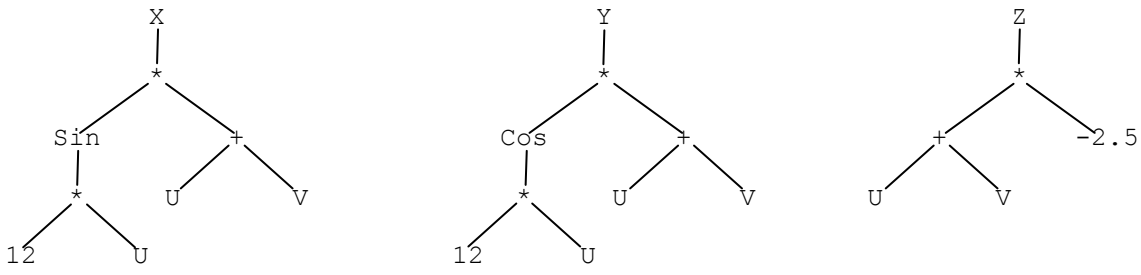


Abb. Voneinander unabhängig angelegte Parse-Bäume

Wird hingegen eine zentrale Datenstruktur verwendet, so ergibt sich folgende Struktur:

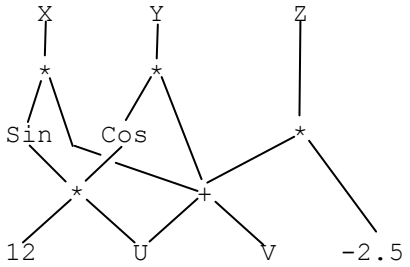


Abb. Parse-Bäume unter Verwendung einer zentralen Datenstruktur

Auf den ersten Blick mögen die Parse-Bäume, die mit der zentralen Datenstruktur angelegt wurden sehr kompliziert wirken. Betrachtet man jedoch die Verbindungen von oben nach unten, so zeigt sich, daß sie nicht komplizierter sind als die direkt im Systemspeicher angelegten Bäume.

6.2 Optimierung von Parse-Bäumen

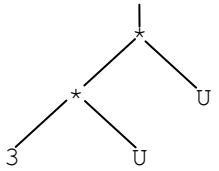
Bisher wurde die als Zeichenkette gegebene Formel in einen Parse-Baum übersetzt. Nun wird dieser Parse-Baum optimiert. Dabei werden an jedem Operator-Knoten zunächst alle Operanden rekursiv optimiert, bevor versucht wird, den Operator selbst zu vereinfachen.

Wenn alle Operanden eines (deterministischen) Operators konstant sind, so ist auch das Ergebnis der Operation konstant. Sofern der Operator frei von Seiteneffekten ist, kann somit im Parse-Baum der Operator und seine konstanten Operanden durch einen Knoten, der das Ergebnis der Operation enthält, ersetzt werden. Alle Operatoren für die Grundrechnungsarten und die üblichen mathematischen Basisfunktionen erfüllen diese beiden Bedingungen und können daher auf diese Weise optimiert werden. Beispiele für diese Art von Optimierung sind etwa Sqrt(2) → 1.41421356.. oder 3+8 → 11.

Wenn bei den Grundrechnungsarten ein Operand eine bestimmte Konstante ist, läßt sich die Operation eliminieren. Dies gilt etwa für 1*X → X, 0*X → 0, X+0 → X, 0-X → -X und verwandte Fälle. Dabei ist X stets ein beliebiger (seiteneffektfreier) Ausdruck.

Wird eine Grundrechnungsart auf zwei gleiche Argumente angewendet, so ergeben sich besondere Ergebnisse, die gut optimiert werden können. So ist X+X → 2*X, X-X → 0, X*X → Sqr(X), X/X → 1, wobei X wieder ein beliebiger (seiteneffektfreier) Teilausdruck ist. Dadurch, daß die Knoten in einer zentralen Datenstruktur verwaltet werden und gleiche Knoten die gleiche Adresse im Speicher haben, ist es an dieser Stelle einfach, beliebige Teilbäume auf Gleichheit zu testen. Bei der Erläuterung der Intervallarithmetik wurde darauf hingewiesen, welche Probleme Operationen mit voneinander nicht unabhängigen Operanden haben (vgl. Kapitel 4.3). Diese Problematik wird hier teilweise entschärft.

An dieser Stelle tritt ein Problem des Parse-Baumes zu Tage. Werden mehrere gleichartige Operationen, etwa mehrere Multiplikationen, hintereinander angegeben, so sind nicht alle Operanden aller Multiplikationen in einem Knoten versammelt. Dies wird am Beispiel des Ausdrucks 3*U*U klar:



Offensichtlich ist hier eine Optimierung von 3*U*U auf 3*Sqr(U) nur möglich, wenn mehrere Knoten auf einmal betrachtet werden. Da Fälle X-X und X/X in 'vernünftigen' Formeln nur selten auftreten und das Ergebnis von X + X bei Auswertung mit der Intervallarithmetik nicht schlechter ist als jenes von 2 *X, wurde in dieser

Diplomarbeit nur für die Multiplikation der Programmaufwand getrieben, auch in Nachbarknoten nach gleichartigen Teilbäumen zu suchen.

Viele Prozessoren können Divisionen nur wesentlich langsamer ausführen als Multiplikationen. Daher werden Divisionen durch Konstante ersetzt durch Multiplikationen mit dem (konstanten) Kehrwert. Also etwa X*0.2 statt X/5.0

Die Auswertung von Negationen läßt sich oft leicht vermeiden. So sind etwa folgende Optimierungen möglich X+(-Y) → X-Y, -X+Y → Y-X, -X*-Y → X*Y, -(-X) → X, Sqr(-X) → Sqr(X). An anderen Stellen läßt sich die Negation zwar nicht wegoptimieren, aber sie läßt sich an den Anfang des Teilausdrucks schieben. Dort besteht dann die Möglichkeit, daß sie von der darüber liegenden Operation wegoptimiert werden kann. Beispiele dafür sind X*(-Y) → -(X*Y), (-X)+(-Y) → -(X+Y), Sin(-X) → -Sin(X) etc.

Bisher wurde stets davon ausgegangen, daß alle Operatoren deterministisch und frei von Seiteneffekten sind. Sollte dies für einen Operator nicht der Fall sein (etwa Zufallsfunktionen), so sind viele Optimierungen nicht zulässig. So ist etwa Rnd(1)+Rnd(1) nicht das gleiche wie 2*Rnd(1). Selbst 0*Rnd(1) darf nicht zu 0 optimiert werden, obwohl das Ergebnis des Ausdrucks stets 0 ist, da der Aufruf von Rnd() den internen Zustand des Zufallsgenerators verändert. Es muß daher darauf geachtet werden, daß solche Aufrufe erhalten bleiben.

Natürlich gibt es noch viele weitere Möglichkeiten, Ausdrücke zu optimieren. Es gilt jedoch: je ausgefeilter die Methoden der Optimierung werden, desto mehr globaler Überblick über die Formel ist erforderlich, desto größer ist der Programmieraufwand und desto seltener findet sich eine Formel, die davon profitiert.

6.3 Erstellen von Postfix-Code

Bisher wurde die als Zeichenkette gegebene Formel des Benutzers in einen Parse-Baum umgewandelt und dieser optimiert. Nun stellt sich die Frage, wie ein solcher Baum am besten ausgewertet werden kann.

Eine Möglichkeit wäre, den Baum rekursiv abzuarbeiten. Die Blattknoten, die Konstante oder Variable enthalten, sind einfach auszuwerten. Innere Knoten, die einen Operator enthalten, werden ausgewertet, indem zunächst die Teilbäume der Operanden rekursiv berechnet werden und mit den Ergebnissen dann die Operation ausgeführt wird. Dieses Vorgehen ist zwar grundsätzlich möglich, aber ineffizient. Zum einen muß für jeden Knoten des Baumes ein rekursiver Funktionsaufruf mit all seinem Verwaltungsaufwand durchgeführt werden. Wesentlich schwerwiegender ist allerdings, daß gemeinsame Teilausdrücke im Baum mehrfach ausgewertet werden. An folgendem Beispiel ist zu sehen, wie oft jeder Knoten ausgewertet wird:

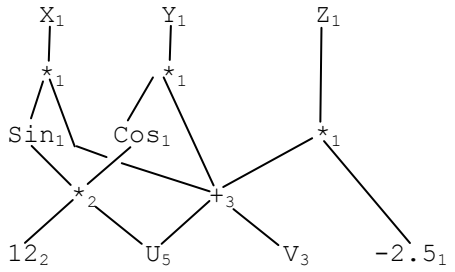
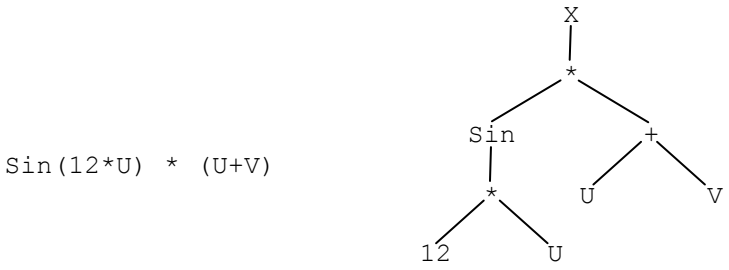


Abb. Parse-Baum mit mehrfach verwendeten Knoten

Eine Möglichkeit, diese Mehrfachauswertungen zu unterbinden, wäre, für jede Auswertungen des Baums einen Index zu vergeben. In jedem Knoten wird dann gespeichert, für welchen Index er zuletzt ausgewertet wurde und was das Ergebnis der Auswertung war. Wenn nun ein Knoten erneut aufgerufen wird, kann erkannt werden, daß der laufende Auswertungsindex gleich dem im Knoten gespeicherten ist, und das alte Ergebnis verwendet werden. Auch wenn zu dieser Methode noch Optimierungen möglich sind, ist sie doch mit einem großen Verwaltungsaufwand verbunden.

Eine Möglichkeit die rekursiven Funktionsaufrufe zu umgehen, besteht darin, die Rekursion vorher aufzurollen und die Formeln in Postfixschreibweise anzugeben. Dazu wird der Baum so abgearbeitet wie für die Auswertung,

allerdings werden die Aktionen nicht ausgeführt, sondern hintereinander angeschrieben. Dadurch wird der baumförmige Ausdruck wieder in einen linearen Ausdruck umgewandelt.




```
Const 12 ~ Var U ~ * ~ Sin ~ Var U ~ Var V ~ + ~ * ~ Done
```

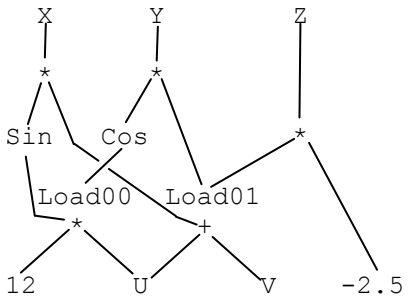
Abb. Eine Formel in mathem. Schreibweise, als Parse-Baum und als Postfix-Ausdruck

Diese Vorgangsweise beseitigt die Schwachstelle der rekursiven Aufrufe, läßt aber das Problem der Mehrfachauswertungen offen.

Bei der Auswertung der Formeln für Flächenpunkte wird der Funktionswert stets unmittelbar hintereinander für alle drei Formeln bestimmt. Es ist daher keine wesentliche Einschränkung gegeben, wenn die Abarbeitungsreihenfolge auf X vor Y vor Z festgelegt wird.

Beim Aufbau der Bäume wurden die Knoten zentral verwaltet. Dabei ist es kein Problem mitzuzählen, welcher Knoten wie oft verwendet wird.

Nun wird die Methode, mit der Postfix-Ausdrücke gewonnen werden, leicht modifiziert. Blattknoten werden, wie bisher, einfach angeschrieben. Bei Operatorknoten des Parse-Baumes werden zunächst die Operandenbäume abgearbeitet. Nach jedem Operanden wird allerdings überprüft, ob der Operand gleich häufig verwendet wird wie der Operator. Wenn dies der Fall ist, so wird der Operand nur für diesen Operator verwendet. Wenn hingegen der Operand öfter verwendet wird, so folgt, daß der Operand auch in einer anderen Operation mitwirkt. Sofern es sich nicht um einen trivialen Operanden (etwa eine Konstante oder Variable) handelt, sondern um einen Operanden, der selbst wieder über Operationen zusammengesetzt ist, lohnt es, den Wert zu sichern, und bei der erneuten Auswertung zu verwenden. Die Zwischenspeicherung erfolgt dabei über Register. Dazu wird hinter den Postfix-Code der Operandenauswertung eine Store-Anweisung für ein bestimmtes Register gestellt. Im Baum wird die Operation, deren Ergebnis gespeichert wurde, durch eine Load-Anweisung ersetzt. Dadurch wird der Baum verändert, und der Verweis auf den Teilbaum, der vorher an dieser Stelle, steht geht verloren; allerdings ist der Inhalt bereits in den Postfix-Code eingegangen und wird nie mehr als Baum benötigt.



X: Const 12 ~ Var U ~ * ~ Store 00 ~ Sin ~
VarU ~ Var V ~ + ~ Store 01 ~ * ~ Done
Y: Load 00 ~ Cos ~ Load 01 ~ * ~ Done
Z: Load 01 ~ Const 2.5 ~ * ~ Done

Abb. Postfix-Code für den Spiralkegel mit Load/Store Operatoren

Diese Form der Optimierung geht davon aus, daß die Operatoren deterministisch und frei von Seiteneffekten sind. Dies ist zwar für viele wichtige Operatoren der Fall, nicht aber für Zufallsgeneratoren. Wenn in einem Ausdruck mehrmals der Zufallsgenerator aufgerufen wird, so ist es natürlich wünschenswert, daß jedesmal eine neue Zufallszahl generiert wird, und nicht der Wert des ersten Aufrufs gespeichert und mehrfach verwendet wird. Es ist daher erforderlich, vor der Umwandlung eines Teilbaums in einen Load-Knoten sicherzustellen, daß in dem Teilbaum kein Operator verwendet wurde, der unbedingt ausgeführt werden muß.

Bisher wurde für jeden Knoten des Baumes ein Stück Postfix-Code erstellt. Da die Auswertung des Codes um so länger dauert, je mehr Postfix-Anweisungen abzuarbeiten sind, ist es klug, möglichst viel Funktionalität in eine Anweisung zu packen.

Konstante Werte werden oft mit einem anderen Ausdruck über eine der Grundrechnungsarten verknüpft. Es lohnt sich daher, neue Operatoren einzuführen, die zusätzlich zur Rechenvorschrift die Konstante direkt enthalten. Dadurch ergeben sich zwei Vorteile. Einerseits wird jedesmal, wenn diese Operatoren verwendet werden, der Postfix-Code um eine Anweisung kürzer, da aus den zwei Anweisungen für Konstante und Operation eine zusammengefaßte entsteht. Andererseits kann das Wissen, daß eine Operation mit einer Konstanten (deren Ableitungen Null sind) erfolgt, verwendet werden, um die Berechnung zu optimieren, indem unnötige Multiplikationen und Additionen mit der Ableitung der Konstanten entfallen. Dies entschärft die in Kapitel 4.2 genannten Probleme mit den 'toten' Operationen bei der automatischen Differenzierung.

Wenn der Sinus eines Wertes berechnet wird, dann wird oft, etwa bei Drehflächen, auch der Cosinus des selben Argumentes benötigt. Bei der Berechnung einer dieser Funktionen und ihrer Ableitung fällt die andere Funktion mit Ableitung fast von selbst an. Es ist daher klug, die Auswertung von Sinus und Cosinus eines Argumentes zusammen zu fassen. Wenn bei der Umwandlung des Parse-Baumes in den Postfix-Ausdruck ein Sinus-Operator auftritt, wird zunächst der Operand abgearbeitet. Dann wird in der zentralen Knotentabelle nach dem Cosinus- Knoten für das selbe Argument gesucht. Wenn dieser Knoten nicht gefunden wird, ist hier keine spezielle Optimierung möglich und es wird wie oben beschrieben untersucht, ob der Operand anderswo benötigt wird, bevor die Sinus-Anweisung in den Postfix-Code geschrieben wird. Wenn allerdings sowohl Sinus als auch Cosinus des Winkels benötigt werden, so wird zunächst untersucht, ob der Operand außer an diesen beiden Stellen noch woanders benötigt wird. Danach wird eine Sin_with_Cos Anweisung, in den Postfix-Code geschrieben. Diese berechnet die beiden Winkelfunktionen, speichert den Cosinus in einem Register und liefert den Sinus als Ergebnis zur weiteren Berechnung. Im Parse-Baum wird nun der Cosinus-Knoten durch eine Load-Anweisung ersetzt. Natürlich lassen sich in obiger Betrachtung die Funktionen Sinus und Cosinus gegeneinander austauschen. Weiters wäre es möglich neben dem Funktionspaar Sinus/Cosinus auch andere Paare, wie etwa Tangens/Cotangens zu betrachten, doch treten diese so selten auf, daß in dieser Diplomarbeit nur das Paar Sinus/Cosinus betrachtet wurde.

```
X: Var U ~ Const(12) Mul ~ Sin_with_Cos 00 ~
   VarU ~ Var V ~ + ~ Store 01 ~ * ~Done
Y: Load 00 ~ Load 01 ~ * ~ Done
Z: Load 01~ Const(-2.5) Mul ~ Done
```

Abb. Optimierter Postfix-Code des Spiralkegels

6.4 Auswertung von Postfix-Code

Bisher wurde die Formel von der Darstellung als Zeichenkette in einen Postfix-Ausdruck umgewandelt. Alle dazu nötigen Schritte waren unabhängig von den konkreten Werten der Variablen und müssen daher nur ein Mal, zu Beginn des Programms, ausgeführt werden. Nun geht es darum, die Postfix-Formel auch für konkrete Variablenwerte auszuwerten.

Postfix-Code ist sehr einfach auszuwerten. Es gibt keine Hierarchie der Operatoren oder Klammern mehr. Die Anweisungen können direkt von vorne nach hinten abgearbeitet werden. Natürlich gibt es trotzdem Zwischenergebnisse, die zur weiteren Verarbeitung gespeichert werden müssen. Wenn dafür ein Stack verwendet wird, ergeben sich einfache Abarbeitungsregeln für Postfix-Ausdrücke. Die Werte von Konstanten und Variablen im Code werden oben auf dem Stack abgelegt. Operatoren nehmen so viele Elemente, wie sie benötigen, vom Stack, führen ihre Verknüpfung aus und legen das Ergebnis wieder auf dem Stack ab. Load legt den Inhalts eines

Registers oben auf den Stack. Store kopiert das oberste Stackelement in ein Register ohne den Stack zu verändern. Done schließt die Auswertung ab. Es liegt dann nur ein einziger Wert, das Ergebnis, auf dem Stack.

Dabei ist es prinzipiell egal, ob nur der Funktionswert allein oder auch die Ableitungen berechnet werden, bzw. ob die Parameter feste Werte haben, oder ob mit Intervallarithmetik gerechnet wird. Der Unterschied besteht nur im Datentyp des Stacks und den Routinen für die einzelnen Operationen.

Zur Verdeutlichung noch ein Beispiel für die Auswertung der Postfix-Ausdrüce für den Spiralkegel an der Stelle U=l, V=2.

Postfix-Code	Stack			Reg. 0	Reg. 1
Var U	1.0				
Const (12.0) Mul	12.0				
Sin_With_Cos 00	-0.53658			0.84385	
Var U	-0.53658	1.0		0.84385	
Var V	-0.53658	1.0	2.0	0.84385	
+	-0.53658	3.0		0.84385	
Store 01	-0.53658	3.0		0.84385	3.0
*	-1.60972			0.84385	3.0
Done	X → -1.60972				
Load 00	0.84385			0.84385	3.0
Load 01	0.84385	3.0		0.84385	3.0
*	2.53156			0.84385	3.0
Done	Y → 2.53156				
Load 01	3.0			0.84385	3.0
Const (-2.5) Mul	-7.5			0.84385	3.0
Done	Z → -7.5				

6.5 Boolesche Algebra, Vergleiche und Bedingungen

Die Formeln für Flächen in Parameterdarstellung müssen stetig und differenzierbar sein. Daher sind Boolesche Funktionen für solche Flächen ungeeignet. Da der Parser, der im Rahmen dieser Diplomarbeit entwickelt wurde, auch für zukünftige, andere Anwendungen verwendbar sein soll, wurden die Booleschen Funktionen, Vergleiche und Bedingungen dennoch implementiert.

Alle bisherigen Funktionen wurden für reelle Zahlen berechnet. Um nicht einen weiteren Datentyp einführen zu müssen, werden auch die Booleschen Werte als reelle Zahlen dargestellt. Dabei wird Null als False und jeder Wert verschieden von Null als True interpretiert. Das Ergebnis einer Booleschen Operation ist immer Null oder Eins.

Für Bedingungen wurde der Operator Cond(X, Y, Z) eingeführt, dessen Ergebnis jenes von Y oder Z ist, je nachdem ob X True oder False ist. Cond(X, Y, Z) entspricht somit einem If X Then Y Else Z. Alle Operationen, auch Cond, müssen ein Ergebnis liefern. Daher müssen stets Y und Z angegeben sein.

Um alle üblichen Booleschen Operatoren (And, Or, Exor, Not) sowie die Vergleichsoperatoren (<, >, =, !=...) mit der in C üblichen Priorität zu implementieren, wurde die Grammatik um einige Regeln erweitert. (vgl. Kapitel 7.7)

Aus der erweiterten Grammatik ergeben sich keine prinzipiell neuen Probleme bei der Umwandlung der Formel von einer Zeichenkette in einen Parse-Baum.

Auch bei der Optimierung des Parse-Baumes werden nur einige spezielle Regeln hinzugefügt, die helfen, die neuen Operatoren zu vereinfachen. Einige Beispiele dafür sind etwa X And 1 → X, X Nor 1 → 0, X Exor 1 → Not X, (Not X) Or (NotY) → X NAnd Y, Cond(1,X,Y) → X, Cond(X,Y,Y) → Y.

Bei der Auswertung von Booleschen Ausdrücken gibt es Fälle, in denen aus einem Argument bereits das Ergebnis der Verknüpfung bestimmt werden kann, ohne das zweite Argument auswerten zu müssen. Beispielsweise ist False And X gleich False, egal welchen Wert X annimmt. Somit stellt sich die Frage, ob es überhaupt sinnvoll ist, X auszuwerten. X nicht auszuwerten, ist schneller und erlaubt auch Bedingungen, deren zweiter Teil zu einem Fehler führen würde wie etwa (U != 0) And (3/U = V) für U=0. Wenn X nicht ausgewertet wird, dann muß eine Methode gefunden werden, mit der der nicht benötigte Programmteil übersprungen wird. Im Rahmen dieser Diplomarbeit wurde aus Gründen der Geschwindigkeit entschieden Teilausdrücke, die nichts zum Ergebnis betragen, nicht auszuwerten und dafür Sprunganweisungen in den Postfix-Code aufzunehmen, obwohl diese einen Stilbruch darstellen.

Ein einfaches Beispiel für die Verwendung Boolescher Logik stellt folgender Ausdruck dar, der je nach den Werten von U und V +0.5 oder -0.5 liefert:

```
X = 0.5 - (U Or V);
```

Der dazu gehörige Postfix-Ausdruck lautet:

```
X: Var U ~ Or (Offset +2) ~ Var V ~ Bool ~ Const 0.5 Sub ~ Done
```

Wenn dieser Ausdruck nun ausgewertet wird, so gibt es am Or-Operator zwei Möglichkeiten: Wenn U True (ungleich Null) ist, dann hängt das Ergebnis von (U Or V) nicht von V ab. Der Or-Operator liefert also Eins zurück und überspringt die nächsten zwei Anweisungen, sodaß bei Const 0.5 Sub weitergerechnet wird. Ist U hingegen False (gleich Null), so wird die Null vom Stack entfernt und mit der folgenden Operation (Var V) fortgefahren.

Da oben definiert wurde, daß das Ergebnis einer Booleschen Operation stets Null oder Eins ist, reicht es nicht, einfach den Wert von V als Ergebnis von U Or V zu verwenden. Der Operator Bool entspricht der Booleschen Identität und dient dazu, aus einer beliebigen, von Null verschiedenen Zahl eine Eins zu machen.

Da das Konzept des Sprungs bereits eingeführt ist, liegt es nahe, auch Bedingungen auf Sprünge zurückzuführen. Dazu zunächst ein einfaches Beispiel:

```
X = Cond(U Or V, U + 3, 5 * V)
```

Entspricht dem Postfix-Code:

```
X: Var U ~ Or (Offset +2) ~ Var V ~ Bool ~ Cond (Offset +3) ~
   Var U ~ Const 3 Add ~ Branch (Offset +2) ~
   Var V ~ Const 5 Mul ~ Done
```

Wie man sieht, sind zwei Sprünge erforderlich, um die Cond-Anweisung zu realisieren. Der erste Sprung direkt in der Cond-Anweisung dient dazu, im False-Fall über den True-Code hinwegzuspringen. Am Ende des True-Codes muß dann noch ein unbedingter Sprung stehen, der über den False-Code führt.

Offensichtlich läßt sich dieser Postfix-Code noch deutlich optimieren. So testet Cond die Bedingung auf gleich/ungleich Null. Es ist somit nicht nötig, vor dem Cond noch mit einer BooI-Operation den Wert der Bedingung auf Null/Eins zu bringen. Weiters springt der Branch-Befehl direkt zum Ende des Codes und kann daher durch ein Done ersetzt werden.

Somit lautet der optimierte Postfix-Code:

```
X: Var U ~ Or (Offset +1) ~ Var V ~ Cond (Offset +3) ~
   Var U ~ Const 3 Add ~ Done ~
   Var V ~ Const 5 Mul ~ Done
```

Es wäre noch möglich, direkt aus dem Or in den True-Zweig der Bedingung zu springen, doch muß in diesem Fall darauf geachtet werden, das Ergebnis des Or vom Stack zu löschen. Im Rahmen dieser Diplomarbeit wurde auf diese Optimierung verzichtet.

Bei der Auswertung von bedingten Ausdrücke ergibt sich auch ein Problem für das Konzept der Load/Store Operationen. Wenn jene Teile des Postfix-Codes, die übersprungen werden, Store-Befehle enthalten, werden für spätere Load-Befehle keine Werte zur Verfügung stehen. Dieses Problem wurde hier dadurch gelöst, daß innerhalb einer Bedingung oder der zweiten Hälfte eines Booleschen Ausdrucks bei der Umwandlung des Parse-Baums in den Postfix-Code keine Store-Operationen zugelassen wurden. Dies kann zwar zu einer Doppelauswertung von Teilausdrücken führen, ist aber ein sicheres Verfahren.

7. Implementation / Module

Bei der Implementation des 'Faster than LIght Ray Tracer' kurz FLIRT am Institut für Computergrafik der TU-Wien wurde darauf geachtet, daß der Code möglichst einfach auf verschiedenen Maschinen verwendbar ist.

Als Programmiersprache wurde ANSI C gewählt. Dabei werden statt der üblichen zwei drei Filearten unterschieden. *.C Files enthalten Programm-Code. Die Endung *.F deutet auf Macro- und Funktions- deklarationen hin, wären die Definitionen von Konstanten und Typen in *.D Files enthalten sind.

Um unabhängig von der Implementierung der üblichen C-Typen auf verschiedenen Maschinen zu sein (etwa. 16 vs. 32 Bit für Integer-Variablen) sind alle Standardtypen in System.D neu deklariert worden. Die Namen der neuen Datentypen bestehen aus Großbuchstaben und lauten praktisch gleich wie die Standardnamen (z.B. INT ↔ int). Durch diese Definitionen ist es leicht möglich, auf einen Schlag im gesamten Programm die Datentypen an eine neue Maschine anzupassen.

Aus ähnlichen Gründen gibt es Ein-/ Ausgabeprozeduren, deren Funktionalität etwa jener der Standardprozeduren entspricht.

Für die Flächen in Parameterdarstellung sind zehn Dateien von besonderer Bedeutung:

- EPIPED.D definiert die Schnittstelle zum Epipedbaum
- XYZ_UV.D definiert jene Typen, über die mit den Epipeden kommuniziert wird.
- XYZPARSE.D definiert die Schnittstelle des Parsers nach außen. Dadurch ist es auch für andere Module möglich, Formeln auszuwerten.
- XYZ_PRIV.D definiert jene Typen, die nur innerhalb des Parsercodes sichtbar sein müssen, nicht aber außerhalb.
- XYZ_UV.F definiert die Prozeduren, die vom Epipedbaum verwendet werden.
- XYZPARSE.F definiert jene Prozeduren, die für die Formelauswertung nötig sind.
- XYZParse.C enthält den Code für die Umwandlung eines Stringausdrucks in einen Parse-Baum und weiter in einen Postfix-Ausdruck.
- XYZ_Opt.C beschäftigt sich mit der Optimierung von Parse-Bäumen.
- XYZ_Eval.C: Hier findet die Auswertung von Postfix-Code statt. Dabei können neben dem Funktionswert auch die Ableitungen berechnet werden. Auch die Verwendung von Intervallen ist möglich.
- XYZ_UV.C kümmert sich um die Schnittstelle zum Epipedbaum. Hier wird die Bilddefinition gelesen, die Verwaltung der Flächen durchgeführt und die Spaltung der Flächenstücke in kleinere Teile wahrgenommen. Weiters wird die Statistik für die Auswertungen der Flächen geführt.

Eine genauere Beschreibung aller dieser Module befindet sich im folgenden Kapitel.

Das Institut für Computergrafik der TU-Wien verfügt zur Zeit über einen VAX-Cluster unter VMS, an dem FLIRT entwickelt wird. Der Code dieser Diplomarbeit wurde auf einem ATARI ST entwickelt und anschließend auf den Cluster überspielt. Dort fand dann die Einbindung in FLIRT, die Fehlersuche und die Berechnung der Bilder statt.

7.1 Epiped.D

Das Modul Epiped.D definiert alle Datentypen, über die der Code für die Epiped-Bäume mit seinen Anwendern kommuniziert. Es bildet daher für die Flächen in Parameterdarstellung die Schnittstelle zu FLIRT.

Eine Fläche, die mit einem Epipedbaum bearbeitet wird, verfügt über globale Daten für die gesamte Fläche, sowie an jedem Knoten des Epipedbaums über lokale Information. Dabei wird zwischen inneren Knoten und Blattknoten unterschieden. Für allen Knoten wurde in der Initalisierungsphase ein umschließendes Parallelepiped bestimmt. Für innere Knoten ist zur Laufzeit nur noch das Parallelepiped wichtig, um die Suche nach Schnittpunkten in die richtige Richtung zu lenken. Da von Blattknoten die Newton-Iteration ausgeht, muß hier oft mehr Information gespeichert werden. Die Art dieser Daten hängt wesentlich von dem Typ der Fläche ab. Die Daten werden vom Epipedbaum nur verwaltet, aber nicht gelesen. Um eine einheitliche Verwaltung der Daten zu ermöglichen, existieren drei Union-Datentypen, in die jede Flächenart ihre spezifischen Datenformate einträgt.

```
typedef union surface
 {
  XYZ_UV xyz_uv;
  ...
 } EPI_SURFACE;

typedef union patch
 {
  XYZ_PATCH xyz_patch;
  ...
 } EPI_PATCH;

typedef union leaf
 {
  XYZ_LEAF xyz_leaf;
  ...
 }
EPI_LEAF;
```

Bisher wurde die Datenschnittstelle zwischen dem Epipedbaum und den Flächen behandelt. Nun soll auch die funktionale Schnittstelle behandelt werden. Die Deklarationen aller Funktionen, die eine Fläche dem Epipedbaum zur Verfügung stellen muß, sind in der Struktur EPIPED_FUNCT zusammengefasst:

```
typedef struct epiped_funct
 {
  UCHAR        *name;
  BOOLEAN      externtransform;
  VOID         (*init surface) (VOID);
  EPI_SURFACE* (*read_surface) (UINT block_nr);
  VOID         (*write_surface)(EPI_SURFACE *surf);
  EPI_SURFACE* (*copy_surface) (EPI_SURFACE *source, UINT block_nr,
                                MAT3 *obj2world, MAT3 *world2obj);
  VOID         (*setup_surface)(EPI_SURFACE *surf, BOX2 *uv,
                                EPI_PATCH *patch);
  VOID         (*uv_split_surface)(EPI_SURFACE *surf, SPLIT_DIR *dir,
                                FLOAT* split_pos, EPI_PATCH *parent,
                          EPI_PATCH *small child, EPI PATCH *big child);
  VOID         (*u_split_surface) (EPI_SURFACE *surf, FLOAT *u_split,
                                   EPI_PATCH *parent,
                           EPI_PATCH *small_child, EPI_PATCH *big child);
  VOID         (*v_split_surface) (EPI_SURFACE *surf, FLOAT *v_split,
                                   EPI_PATCH *parent,
                           EPI_PATCH *small_child, EPI_PATCH *big_child);
  EPI_LEAF*    (*build_leaf) (EPI_SURFACE *surf, UINT block nr,
                                                  EPI PATCH *patch);
  VOID         (*bound_surface) (EPI_SURFACE *surf, EPI_PATCH *patch,
                                              VEC3 dir, BOXl *min_max);
  VOID         (*eval_surface)(EPI_SURFACE *surf, PNT2 uv, PNT3 *xyz,
                                                VEC3 *d_u, VEC3 *d_v);
  VOID         (*end_surface) (EPI_SURFACE *surf);
  VOID         (*exit_surface) (VOID);
 } EPIPED_FUNCT;
```

Die einzelnen Komponenten haben dabei folgende Bedeutung:

`CHAR *name`: Um unterscheiden zu können, welche Art von Fläche den Epipedbaum nutzt, wird dieser Name verwendet.

`BOOLEAN externtransform`: Es gibt Flächen, die geschlossen transformiert werden können. Dazu gehören etwa Bezier- und B-Spline-Flächen, die transformiert werden, indem ihre Stützpunkte neu berechnet werden. Andere Flächen, etwa Flächen in Parameterdarstellung können nur Punkt für Punkt transformiert werden. Ist externtransform TRUE, so wird die Transformation punktweise durch den Epipedbaum

ausgeführt, ansonsten muß die Fläche ihre innere Datenstruktur nach einem Copy_Surface selbst an die neue Lage im Raum anpassen.

`VOID (*init_surface) (VOID)`: Hier kann eine Initialisierungsroutine für einen Flächentyp stehen. Wenn dies nicht benötigt wird, so wird dies durch einen NULL-Pointer dargestellt.

`EPI_SURFACE* (*read_surface) (UINT block_nr)`: Durch diese Routine wird die Definition einer Fläche vom Datenfile eingelesen. Das Datenformat hängt dabei von der Fläche ab. Um die gelesenen Daten zu speichern, kann lokaler Speicher über die Memory-ID block_nr angefordert werden. Dieser wird bei Programmende automatisch wieder freigegeben.

`VOID (*write_surface)(EPI_SURFACE *surf)`: Zu Kontrollzwecken und für Log-Files ist es günstig, wenn die gelesenen Daten auch wieder geschrieben werden können. Die Funktion write_surface gibt die Daten in jenem Format aus, das read_surface lesen kann.

`EPI_SURFACE* (*copy_surface) (EPI_SURFACE *source, UINT block_nr, MAT3 *obj2world, MAT3 *world2obj)`: Die Fläche source wird kopiert. Dabei kann für lokale Daten wieder Speicher unter der Memory-ID block_nr abgefordert werden. Wenn die Fläche geschlossen transformiert werden kann (vgl. externtransform) so muß dies hier erfolgen.

`VOID (*setup_surface) (EPI_SURFACE *surf, BOX2 *uv, EPI_PATCH *patch)`: Eine einzelne Fläche wird initialisiert. Dabei wird ein Patch generiert, der die gesamte Fläche erfaßt. Weiters wird dem Epipedbaum mitgeteilt, über welchem (u,v)-Parameterbereich die Fläche definiert ist.

`VOID (*uv_split_surface) (EPI_SURFACE *surf,SPLIT_DIR *dir,FLOAT *split_pos, EPI_PATCH *parent, EPI_PATCH *small_child, EPI_PATCH *big_child)`: Einige Flächen können selbstständig entscheiden, an welcher Stelle und in welche Richtung sie am besten zu unterteilen sind, wenn möglichst schnell ebene Flächenstücke entstehen sollen. Dabei wird das parent Flächenstück in zwei Teile zerlegt. Die Schnittrichtung wird in dir gemeldet (SPLIT_U bzw. SPLIT- V), der Parameterwert, an dem gespalten wurde, ist in split_pos vermerkt. Für Flächen, die hier nicht selbst entscheiden können, wird der Pointer auf diese Routine auf NULL gesetzt.

`VOID (*u_split_surface) (EPI_SURFACE *surf,FLOAT *u_split,EPI_PATCH *parent, EPI PATCH *small child, EPI PATCH *big child);`

`VOID (*v_split_surface) (EPI_SURFACE *surf~FLOAT *v_split,EPI_PATCH *parent, EPI_PATCH *small_child, EPI_PATCH *big_child);`

Diese beiden Routinen verhalten sich ähnlich wie uv_split, allerdings ist hier die Richtung der Trennung fest vorgegeben. Auch für die Trennstelle existiert ein Vorschlag, der jedoch von der Fläche auch verändert werden darf. Im Gegensatz zu uv_split müssen diese Routinen von allen Flächen unterstützt werden.

`EPI_LEAF* (*build_leaf) (EPI_SURFACE *surf, UINT block_nr, EPI_PATCH *patch)`: Einige Flächen benötigen für die Newton-Iteration in einem lokalen Bereich spezielle, lokale Informationen. Für die unterste Ebene des Epipedbaums wird daher diese Funktion aufgerufen, mit der Flächen solche lokalen Daten speichern können. Flächen die diese Funktion nicht benötigen, tragen für diese Routine einen NULL-Pointer ein.

`VOID (*bound_surface) (EPI_SURFACE *surf, EPI_PATCH *patch, VEC3 dir, BOX *min_max)`: Beim Aufbau des Epipedbaums ist es nötig, ein Flächenstück möglichst eng zwischen zwei Ebenen, die normal auf einen vorgegebenen Vektor sind, einzusperren. Bound_Surface bestimmt solche Ebenen und liefert in Min_Max die inneren Produkte zwischen dem Normalvektor und den Ebenenpunkten.

`VOID (*eval_surface) (EPI_SURFACE *surf, PNT2 uv, PNT3 *xyz, VEC3 *d_u, VEC3 *d_v)`: Für die Newton-Iteration ist es nötig einen Flächenpunkt und seine Ableitungen nach U und V zu berechnen. Diese Aufgaben wird von Eval_Surface wahrgenommen.

`VOID (*end_surface) (EPI_SURFACE *surf)`: Für jede einzelne Fläche wird am Ende des Programms end_surface aufgerufen. Hier können Statistiken ausgegeben werden. Weiters kann das System aufgeräumt werden, wobei nur jener Speicher explizit freigegeben werden muß, der nicht über die Memory_ID block_nr angefordert wurde. Wenn eine Fläche diese Funktion nicht benötigt, kann NULL eingetragen werden.

`VOID (*exit_surface) (VOID)`: Nachdem für alle Flächen end_surface aufgerufen wurde, wird für jeden Flächentyp noch exit_surface aufgerufen. Die Funktion von exit_surface entspricht dabei jener von end_surface.

7.2 XYZParse.D

XYZParse.D definiert alle Typen und Konstanten, die in einem Modul, das Formeln auswerten soll, benötigt werden. Dazu gehören der Typ für die Knoten des Parse-Baums (PARSE_NODE) und des Postfix-Codes (PCODE). Weiters werden die Typen für die Auswertung des Postfix-Codes mit Ableitungen und Intervallarithmetik definiert. (EVAL_VARS...)

7.3 XYZ_UV.D

XYZ_UV.D definiert die Typen für Flächen in Parameterdarstellung. Es sind dies je ein Typ für die gesamte Flächendefinition (XYZ_UV), für ein Flächenstück während des Aufbau des Epipedbaums (XYZ_PATCH) und für die Newton-Iteration auf einem Flächenstück (XYZ_LEAF). Flächen in Parameterdarstellung benötigen zur Newton-Iteration keine speziellen, von der bearbeiteten Region abhängigen Daten. Theoretisch könnte der Typ für die Blätter des Epipedbaum also auch VOID sein. Da dieser aber Bestandteil einer Union ist, in der keine Komponente VOID sein darf, ist XYZ_LEAF als INT (ein beliebiger kleiner Datentyp) definiert.

7.4 XYZ_Priv.D

XYZ_Priv.D wird nur von den Modulen XYZ*.C verwendet. Es ist nicht dafür gedacht, von außenstehenden Modulen eingesetzt zu werden. Dafür stehen XYZParse.D und/oder XYZ_UV.D zur Verfügung.

In XYZ_Priv.D werden eine Reihe von Switches gesetzt, die steuern wieviel Debug-Code erzeugt werden soll. Daneben sind verschiedene Vielfache von Pi definiert (vgl. Nullstellen und Extremwerte der Winkelfunktionen). Weiters sind gleichartige Operatoren, die in Switch Statements ähnlich zu behandeln sind, zu Gruppen zusammengefaßt (Variable, Binäre Operatoren, Unäre Operatoren, Vergleiche, Boolesche Operatoren).

7.5 XYZParse.F

XYZParse.F deklariert alle Funktionen die ein Modul kennen muß, um den Formel auswerten zu können. Diese stammen alle aus XYZParse.C, XYZ_Opt.C und XYZ_Eval.C und sind später bei der Beschreibung der *.C Files erläutert.

7.6 XYZ_UV.F

XYZ_UV.F deklariert die Funktionen und globalen Variablen, die für die Verwaltung der Flächen in Parameterdarstellung erforderlich sind.

Bei der Beschreibung von Epiped.D wurde das Interface zwischen dem Epiped- und dem Flächencode erklärt. XYZ_UV.F enthält nun die Deklaration der Funktionen aus XYZ_UV.C, die dieses Interface ausfüllen.

7.7 XYZParse.C

```
DOUBLE read_realexpr(VOID);
```

Read_RealExpr dient dazu einen reellen Wert aus der Bilddatei zu lesen. In der Datei kann dabei entweder direkt eine Zahl stehen, oder aber ein von runden Klammem umschlossener, konstanter mathem. Ausdruck.
Bisher konnten reelle Werte in der *.FTL Datei nur als Zahlen angegeben werden, die durch Leerzeichen von einander getrennt waren. Dabei war stets klar, wo eine Zahl aufhörte und die nächste anfing. Bei der Auswertung mathematischer Ausdrücke ist dies nicht immer klar (vgl. unäres vs. binäres Minus). Daher wurde es erforderlich, durch die Klammerung eine explizite Trennung der Ausdrücke zu erzwingen.

Read_RealExpr rechnet intern mit FLOAT Präzision. Die Tatsache, daß der Retourwert als DOUBLE definiert ist, ist nur mit der Kompatibilität zur alten Funktion readreal() begründet.

```
FLOAT eval_const_expr(UCHAR *expr);
```

Eval_Const_Expr dient dazu eine mathematischen Ausdruck auszuwerten. Die Funktionen des Ausdrucks dürfen nur von Konstanten und Benutzervariablen (Attributen) abhängig sein, nicht aber von den

Systemvariablen (T, U, V, W, X, Y, Z). Eval_Const_Expr erledigt alle nötigen Schritte, um die Zeichenkette in einen Parse-Baum umzuwandeln, diesen zu optimieren und den so erhaltenen Baum in Postfix-Notation überzuführen. Der Postfix-Ausdruck wird dann ausgewertet und das Ergebnis der Auswertung als Ergebnis von Eval_Const_Expr geliefert.

Der Formelstring darf nur die Formel selbst enthalten. Abschließende Symbole hinter der Formel (etwa ein Strichpunkt) sind nicht erlaubt und werden als Fehler gemeldet.

```
VOID read_equation(UCHAR *equ, UINT max_len, UCHAR open_brace,
                                             UCHAR close_brace);
```

Read_Equation liest eine Formel aus der Bild-Datei in einen String. Wenn Open_Brace das Null-Zeichen ist, wird so lange gelesen bis zum ersten Mal Close_Brace auftritt. Ist hingegen Open_Brace verschieden von Null, so muß das erste gelesene Zeichen gleich Open_Brace sein, und es wird solange gelesen, bis gleich viele Close_Brace wie Open_Brace Zeichen gelesen wurden. Die Begrenzungszeichen werden mitgelesen und auch in den Zielstring geschrieben. In beiden Fällen wird mit einer Fehlermeldung abgebrochen, wenn bereits Max_Len Zeichen gelesen wurden und die Abbruchbedingung nicht erfüllt ist. Dadurch wird erreicht, daß im Falle eines Tippfehlers in der Bilddatei der Speicher hinter dem String nicht unkontrolliert überschrieben wird.

```
VOID parse_error(UCHAR *msg, INT posl, INT pos2);
```

Parse_Error gibt die Meldung Msg auf dem Bildschirm aus. In der Zeile darunter werden an zwei Stellen Pfeile gezeichnet, um gezielt Zeichen in der Botschaft zu markieren. Dies ist nützlich, wenn ein Tippfehler in einer Gleichung möglichst genau angezeigt werden soll. Eine Positionsangabe kleiner Null führt zu keiner Markierung.

```
VOID fatal_op(UCHAR *pos, OPERATOR op);
```

Wenn bei der Manipulation mit einem Parse-Baum oder mit PostfixCode ein unerlaubter Operator auftritt, so wird FataI_Op verwendet, um den Fehler anzuzeigen. Der Name der Funktion, in der der Fehler aufgetreten ist, wird in Pos übergeben; Wenn der störende Operator Op zwar grundsätzlich existiert, aber im konkreten Zusammenhang verboten war, so wird sein Name ausgegeben. Wenn Op nicht bekannt ist, wird Op als Integer ausgegeben.

```
VOID init_hash(PARSE_NODE *hash_table);
```

Für die Verwaltung der Knoten von Parse-Bäumen werden Hash-Tabellen eingesetzt. Bevor der erste Ausdruck in einen Baum umgewandelt werden kann, muß Init_Hash aufgerufen werden, um alle Knoten der Hash- Tabelle als unbenutzt zu markieren.

```
PARSE_NODE *new_node(PARSE_NODE *node, PARSE_NODE *hash_table);
```

New_Node trägt einen Knoten in eine Hash-Tabelle ein. Wenn der Knoten bereits vorhanden war, wird ein Zeiger auf den alten Eintrag zurückgeliefert. Wenn der Operator des neuen Knotens kommutativ (z.B. Addition, Multiplikation, Boolesche Operatoren) ist, so wird auch nach dem Knoten mit vertauschten Operanden gesucht und gegebenenfalls dessen Adresse zurückgeliefert. Wenn der Knoten nicht gefunden wurde, wird ein neuer Eintrag angelegt.

```
VOID free_tree(PARSE_NODE *tree, BOOLEAN free_top, INT cnt);
```

Free_Tree wandert einen Parse-Baum rekursiv ab und gibt alle Knoten frei. Wenn Free_Top False ist, wird der Wurzelknoten des Baumes nicht freigegeben. Da ein Knoten auch mehrfach verwendet sein kann, kann es auch nötig sein, ihn mehrfach freizugeben. Cnt gibt an, wie oft der Baum freigegeben wird. (Hinweis: Wenn man Cnt auf -1 setzt, so wird der Baum -1 mal freigegeben, also einmal mehr belegt.)

```
VOID free_node(PARSE_NODE *tree, INT cnt);
```

Free_Node gibt einen Knoten eines Parse-Baums Cnt fach frei.

```
VOID skip_blank(UCHAR **parse);
```

Skip_Blank überspringt alle Leerzeichen am Anfang des Strings *Parse und setzt den Stringanfangs auf das erste, von einem Space verschiedene Zeichen.

```
VOID init_parse(VOID);
```

Init_Parse initialisiert eine Tabelle mit den Namen von Operatoren. Es muß aufgerufen werden bevor das erste Mal eine Formel mit Parse_Expr in einen Parse-Baum umgewandelt wird.

```
PARSE_NODE *parse_expr(PARSE_NODE *hash_table, UCHAR *equation,
                UCHAR **parse, PARSE_PERMIT permit, PARSE_MACRO *macro);
```

Parse_Expr wandelt eine als Zeichenkette gegebenen mathem. Formel in einen Parse-Baum um. Die Knoten des Baumes werden dabei in einer Hash- Tabelle abgelegt, sodaß gleiche Knoten an der selben Stelle im Speicher liegen. Equation gibt die erste Stelle des Formelstrings an und wird für Fehlermeldungen genutzt. Parse zeigt auf das nächste zu bearbeitende Zeichen. Permit gibt an, welche Arten von Operatoren und Systemvariablen in der Formel auftreten dürfen. So sind Vergleiche, Boolesche Operationen und Bedingungen einfach unterdrückbar. Weiters ist es möglich, bereits früher erstellte Bäume zu benennen und als Macro in einen neuen Baum einzubinden. Dabei ist es wichtig, daß diese Teilbäume mit der selben Hash- Tabelle erstellt wurden. (Definition von PARSE_PERMIT und PARSE_MACRO in XYZParse.D)

Parse_Expr liest solange Zeichen der Formel, wie diese interpretierbar sind. Am Ende der Auswertung zeigt *Parse auf das erste, noch nicht bearbeitete Zeichen. Der Benutzer dieser Routine sollte prüfen, ob dort ein korrektes Abschluß-Symbol (z.B. Strichpunkt oder Null-Zeichen je nach Syntax) steht.

Um einen Ausdruck einzulesen wird folgende Grammatik verwendet:

Parser-Grammatik:

```
EXPR      = EXOR_EXPR [ OR_OP EXOR_EXPR]
OR_OP     = or | nor | | | ||
EXOR_EXPR = AND_EXPR [ EXOR_OP AND_EXPR]
EXOR_OP   = exor | eor | xor | eqv
AND_EXPR  = EQUAL_EXPR [ AND_OP EQUAL_EXPR]
AND_OP    = and | nand | & | &&
EQUAL_EXPR= COMP_EXPR [ EQUAL_OP COMP_EXPR]
EQUAL_OP  = = | == | != | <> | >< | #
COMP_EXPR = ADD_EXPR [ COMP_OP ADD_EXPR]
COMP_OP   = < | <= | =< | >= | => | >
ADD_EXPR  = MUL_EXPR [(+ | -) MUL_EXPR]
MUL_EXPR  = POW_EXPR [(* | / | % | \) POW_EXPR]
POW_EXPR  = TERM [(^ | ´) TERM]
TERM      = SIGN CONST | NEG VARIABLE |
            NEG ( EXPR ) | NEG FUNCT ( EXPR ) |
            NEG MINMAXPOW ( EXPR , EXPR ) |
            NEG cond ( EXPR , EXPR , EXPR )
SIGN      = [+ | -]
NEG       = [-]
CONST     = Float-Konstante nach C-Stil
VARIABLE  = VAR | LOCAL_VAR | USER_VAR | pi
VAR       = u | v | w | x | y | z | t
LOCAL_VAR = Name einer lokalen Variable (Macro-Ausdruck)
USER_VAR  = Name einer globalen Variable (Typ: Float)
FUNCT     = neg  |  not  |  inv  |  abs  |  sgn  |
            sqr  | cubic |  sqrt |  cbrt |
            exp  |  ln   |
            sin  |  cos  |  tan  |  cot  |
            asin |  acos |  atan |  acot |
           floor |  ceil |
           twice |  half | pimul | dg2rd | rd2dg |
            rand |  seed | crc_rand | crc_seed
MINMAXPOW = min | max | pow | quadric
```

Bei dieser Grammatik sind alle Terminalsymbole fett geschrieben. Der senkrechte Strich '|' steht für die exklusive Auswahl. Ausdrücke in eckigen Klammem können beliebig oft, auch kein Mal, verwendet werden. Zwischen zwei Terminalsymbolen sind beliebig viele Leerzeichen zulässig. Zwischen Groß- und Kleinbuchstaben wird nicht unterschieden.

Die Abarbeitung eines Ausdrucks mit einer solchen Grammatik ist in Kapitel 6.1 detailliert beschrieben.

```
VOID ptree_to_postfix(PARSE_NODE *tree, OPERATOR prev_op,
                      PARSE_NODE *hash_table, UINT *reg_cnt, BOOLEAN use_reg,
                      PCODE *postfix, UINT *len, UINT max_len);
```

PTree_To_Postfix wandelt einen Parse-Baum in Postfix-Code um. Dabei werden folgende, gemeinsam mit der Umwandlung in Kapitel 6.3 ausführlich beschriebene Optimierungen durchgeführt:

- Für mehrfach verwendete, zusammengesetzte Ausdrücke wird nur einmal Code erzeugt. Hinter diesem Code wird ein Store-Befehl eingefügt, der bei der Ausführug dafür sorgt, daß das Ergebnis der Ausdrucks zwischengespeichert wird. Dadurch können spätere Anwender das Ergebnis des Ausdrucks verwenden, ohne es selbst berechnen zu müssen. Wenn die Funktionen Sin und Cos auf das gleiche Argument angewendet werden, so werden diese Auswertungen zusammengefasst.
- Operationen zwischen einer Konstanten und einem Ausdruck werden nicht als zwei Befehle (Laden der Konstanten / Operation) abgebildet, sondern zu einer Operation zusammengefasst, die beide Aktionen zusammen ausführt. Für Multiplikationen mit einigen speziellen Konstanten wurden eigene Operatoren definiert (Half, Twice, PiMul, Rd2Dg, Dg2Rd). Auch die Potenzierung mit kleinen ganzen Zahlen wird durch eigene Operatoren bewerkstelligt (Inv, Sqr, Cubic).
- Die Auswertung Boolescher Operatoren geschieht unter Ausnützung von 'Short Cuts'. Das heißt, daß die Berechnung abgebrochen wird, wenn nach der Ermittlung eines Operanden das Ergebnis bereits feststeht. Dort wo es erforderlich ist werden Bool-Operatoren eingefügt um sicherzustellen, daß das Ergebnis einer Booleschen Operation stets 0.0 oder 1.0 ist.

Die Parameter von PTree_To_Postfix haben über folgende Bedeutung:
`Tree`: Der Parse-Baum, der in Postfix-Code umgewandelt werden soll.
`Prev_Op` : Übergeordneter Operator im Baum. Beim Aufruf sollte hier stets DONE stehen.
`Hash Table`: Hashtabelle in der der Parse-Baum steht.
`Reg_Cnt`: Gibt an, wie viele Register bereits belegt sind.
`Use_Reg`: Gibt an, ob mehrfach verwendete Ausdrücke durch Einsatz von Zwischenspeichern optimiert werden sollen. Beim Aufruf von außen sollte hier immer TRUE stehen.
`Postfix`: Adresse des Zielarrays für den Postfix-Code
`Len`: Nummer des nächsten freien Eintrags im Postfix-Array
`Max_Len` : Arraygröße für den Speicherschutz

Nach der Ausführung von PTree_to_Postfix steht in Len die Anzahl der Befehle des Postfix-Codes. Um später festzustellen, wieviele Befehle der Code umfasst, kann man entweder Len speichern oder nach dem letzten Knoten suchen. Dieser ist stets ein Done-Knoten, bei dem die Union-Komponente Jump den Wert -1 hat. Bei allen Done Knoten im Inneren des Codes hat Jump den Wert 0.

```
VOID ptree_to_asc(PARSE_NODE *tree, UCHAR *out_string, INT strg_len);
```

PTree_To_ASC wandelt den Parse-Baum Tree in eine Formel um. Die erstellte Zeichenkette wird dabei maximal Strg_Len Zeichen lang, sodaß ein unkontrolliertes Überschreiben von Speicherbereichen verhindert werden kann. Diese Funktion dient vor allem dazu, die Fehlersuche bei der Formelmanipulation zu vereinfachen.

```
VOID postfix_to_asc(PCODE *pcode, UCHAR *out_string, INT strg_len);
```

PostFix_To_ASC erstellt zu einem Postfix-Ausdruck in interner Darstellung eine für Menschen lesbare Zeichenkette mit maximal Strg_Len Zeichen. Diese Funktion dient vor allem dazu, die Fehlersuche bei der Manipulation von Postfix-Ausdrücken zu erleichtern.

Benutzervariablen:

Für verschiedene Zwecke ist es nützlich, eine Formel mit einem Parameter zu versehen, ohne daß dieser expliziter Bestandteil der Parameterliste ist. Dies wird durch Benutzervariable (Attribute) erreicht. Es handelt sich dabei um benannte Variable vom Typ Real, deren Adresse im Postfix-Code festgehalten wird. Der Benutzer kann nun die Werte der Attribute vor der Auswertung des Postfix-Codes geeignet setzen (im C-Code), und so eine Formel vielfältig parametrisieren. Der Code für die globale Verwaltung der Attribute wurde von Wolfgang Stürzlinger erstellt.

```
VOID readvariables(VOID);
```
ReadVariables liest die Definition aller Attribute aus der Bilddatei. Ein Attribut besteht dabei aus einem Namen und einem optionalen Initialisierungswert. Ist kein Anfangswert angegeben, so wird 0.0 verwendet.

```
FLOAT *readnamevariable(VOID);
```
ReadNameVariables liest aus dem *.FTL File den Namen eines (bereits definierten) Attributes, und liefert dessen Adresse.

```
VOID resetvariables(VOID);
```
ResetVariables setzt den Inhalt aller Attribute auf die Anfangswerte zurück.

Für die Speicherung der Attribute wurden in XYZParse.C folgende, globale Variablen angelegt:

```
#define MAX VARIABLES 100
UINT num_variables;              /* Anzahl der Variablen */
FLOAT variables[MAX_VARIABLES]; /* Tabelle der Variablen */
```

Der Code für die Prozeduren zur Verwaltung der Attributlisten befindet sich in Symbols.C.

7.8 XYZ_Opt.C

XYZ_Opt.C enthält den Code, um einen Parse-Baum zu optimieren. Dazu werden folgende Funktionen verwendet.

```
PARSE_NODE *strip_macro (PARSE_NODE *tree, PARSE_NODE *hash_table)
```
Es ist möglich in einem Parse-Baum benannte Unterbäume als Macros einzubinden-: Diese enthalten außer dem Unterbaum auch noch ihren Namen und eine Stringdarstellung des Unterbaumes. Diese Information ist zum Debuggen nützlich, für die Optimierung und Auswertung aber hinderlich. StripMacro wandert einen Parse- Baum rekursiv ab, und ersetzt alle Macroknoten durch den Teilbaum, den sie enthalten.

```
PARSE_NODE *opt_ptree(PARSE_NODE *tree, PARSE_NODE *hash_table)
```
Opt_PTree ist jene Funktion, die von außen aufgerufen werden soll. Sie verwaltet die intern in mehreren Schritten über Opt_PTree_Const und Opt_PTree_Neg ablaufende Optimierung und formt den Parse-Baum so lange um, bis keine weiteren Vereinfachungen mehr möglich sind.

```
PARSE_NODE *opt_ptree_const(PARSE_NODE *tree, PARSE_NODE *hash_table,
                                                    BOOLEAN *success)
```
Opt_PTree_Neg wandert einen Parse-Baum rekursiv ab und ersetzt dabei konstante Teilausdrücke durch ihren Wert. Dabei werden die optimierten Knoten stets korrekt in der Hash-Tabelle eingetragen. Wenn eine Vereinfachung stattgefunden hat, wird dies in Success vermerkt, sodaß es möglich ist, Opt_PTree_Const so lange aufzurufen, bis keine Optimierung mehr möglich ist.

Bei der folgenden Darstellung der optimierbaren Ausdrücke steht $C_1..C_N$ stets für eine Konstante und X sowie Y für einen beliebigen (Teil-)Ausdruck:

C_1 op $C_2 \rightarrow C_3$ für op = Grundrechnungsoperator, Boolescher oder Vergleichsoperator

Fn(C_1) $\rightarrow C_2$ für alle unären Operationen Fn, mit Ausnahme von Random und Seed (Seiteneffekte!)

```
X + 0 = 0 + X → X       X - 0 → X        0 - X → -X
X * 0 = 0 * X → 0       X / 0 → Error    0 / X → 0
X * 1 = 1 * X → X       X / 1 → X        1 / X → Inv(X)
X *-1 =-1 * X →-X       X /-1 →-X

        0 ^ X → 0      1 ^ X → 1      X ^ 1 → X
                  C ^ X → Exp(Ln(C)*X)
```

```
X  And 0 = 0  And X → 0          X  And C = C  And X → Bool(X)
X  Or  0 = 0   Or X → Bool(X)    X  Or  C = C  Or  X → 1
X NAnd 0 = 0 NAnd X → 1          X NAnd C = C NAnd X → Not(X)
X  NOr 0 = 0  NOr X → Not(X)     X  NOr C = C  NOr X → 0
X ExOr 0 = 0 ExOr X → Bool(X)    X ExOr C = C ExOr X → Not(X)
X  Eqv 0 = 0  Eqv X → Not(X)     X  Eqv C = C  Eqv X → Bool(X)
          für beliebige Konstante C ungleich Null

          X + X → Twice(X)
          X * X → Sqr(X)
          X op X → 0 für op aus {-, %, ExOr, <, >, !=}
          X op X → 1 für op aus {/, Eqv, <=, >=, ==}
          X op X → X für op aus {Min, Max}
          X op X → Bool(X) für op aus {And, Or}
          X op X → Not(X) für op aus {NAnd, NOr}

   Cond(X,Y,Y) → Y      Cond(1,X,Y) → X      Cond(0,X,Y) → Y
```

Wegen der besonderen Bedeutung der Potenz eines Wertes und der Probleme, die bei der Intervall-Arithmetik entstehen können, wenn diese durch fortgesetztes Multiplizieren errechnet werden, gibt es eine Sonderregel, die über zwei Ebenen des Baumes optimiert.

```
      Power(X,C) * X → Power(X,C+1)
      X * Power(X,C) → Power(X,C+1)
(Power(X,C) * Y) * X → Power(X,C+1) * Y
(Y * Power(X,C)) * X → Y * Power(X,C+1)
(X * Y) * Power(X,C) → Power(X,C+1) * Y
(Y * X) * Power(X,C) → Power(X,C+1) * Y
Power(X,C) * (X * Y) → Power(X,C+1) * Y
Power(X,C) * (Y * X) → Power(X,C+1) * Y
X * (Y * Power(X,C)) → Power(X,C+1) * Y
X * (Power(X,C) * Y) → Power(X,C+1) * Y
```

Dabei steht Power für eine der Funktionen Sqr, Cubic, Raise oder die Identität.

```
PARSE_NODE *opt-ptree_neg(PARSE_NODE *tree, PARSE_NODE *hash_table,
                                                    BOOLEAN *success)
```

Opt_PTree_Neg versucht, das unäre Minus und die logische Negation so weit wie möglich aus einem Parse-Baum zu verdrängen. Dazu werden bei jedem Operator zunächst alle Operanden optimiert, indem Opt_PTree_Neg rekursiv aufgerufen wird, und dann untersucht, ob lokal eine Vereinfachung möglich ist. Selbst wenn es lokal keinen Vorteil bringt, wird versucht, Negationsopertoren im Baum nach oben zu ziehen, um sie eventuell auf einer höheren Ebene im Baum entfernen zu können.

Im Rahmen dieser Diplomarbeit werden folgende Fälle erkannt und optimiert:

```
--X → X

-X + -Y → -(X+Y)        -X - -Y → Y - X
 X + -Y → X - Y          X - -Y → X + Y
-X +  Y → Y - X         -X -  Y → -(X+Y)

(-X) * (-Y) → X * Y      (-X) / (-Y) → X / Y

X*(-Y) → -(X*Y)   X/(-Y) → -(X/Y)   (-X)*Y → -(X*Y)   (-X)/Y → -(X/Y)
C*(-X) → (-C)*X   C/(-X) → (-C)/X   (-X)*C → X*(-C)   (-X)/C → X/(-C)
     Min(-X,-Y) → -Max(X,Y)    Max(-X,-Y) → -Min(X,Y)
```

```
F(-X) → -F(X) für F aus {Inv, Sgn, Cubic, Cbrt,
                         Sin, Tan, Cot, ASin, ATan}
F(-X) →  F(X) für F aus {Abs, Cos, Sqr, Not, Bool}

-X op -Y → X op Y für op aus {And, Or, NAnd, Nor, ExOr, Eqv}
 X op -Y → X op Y für op aus {And, Or, NAnd, Nor, ExOr, Eqv}
-X op  Y → X op Y für op aus {And, Or, NAnd, Nor, ExOr, Eqv}

-X < -Y → X > Y    -X <= -Y → X >= Y    -X == -Y → X == Y
-X > -Y → X < Y    -X >= -Y → X <= Y    -X != -Y → X != Y

!!X → Bool(X)      !Bool(X) → !X

!X And  !Y → X NOr Y        !X Or  !Y → X NAnd Y
!X NAnd !Y → X  Or Y        !X NOr !Y → X And Y
!X ExOr !Y → X Exor Y       !X Eqv !Y → X Eqv Y

!X ExOr  Y → X Eqv Y        !X Eqv  Y → X ExOr Y
 X ExOr !Y → X Eqv Y         X Eqv !Y →X ExOr Y

!(X<Y) → X>=Y     !(X<=Y) → X>Y     !(X==Y) → X!=Y
!(X>Y) → X<=Y     !(X>=Y) → X<Y     !(X!=Y) → X==Y

!(X And Y)  → X NAnd Y      !(X NAnd Y) → X And Y
!(X Or Y)   → X NOr Y       !(X Nor Y)  → X Or Y
!(X ExOr Y) → X Eqv Y       !(X Eqv Y)  → X Exor Y

Bool(X op Y) → X op Y für op aus {>, <, >=, <=, !=, ==,
                                  And, Or, NAnd, NOr, ExOr, Eqv}
Bool(F(X)) → F(X) für F aus {Not, Bool}

Cond(F(X),Y,Z) → Cond(X,Y,Z) für F aus {Neg, Bool, Abs, Sgn,
                                        Sqr, Sqrt, Cbrt, Cubic}
```

7.9 XYZ Eval.C

Das Modul XYZ_Eval.C enthält drei Prozeduren für die Auswertung von Postfix-Code. Sie haben alle einen ähnlichen Programmaufbau und vergleichbare Parameter. In Code wird der abzuarbeitende Code übergeben. Vars bestimmt die Werte der Systemvariablen (T,..,Z).
Stack wird als Speicher für die Zwischenergebnisse verwendet. Reg zeigt auf jenen Speicherbereich, der verwendet wird, um Mehrfachauswertungen zu vermeiden. Eine detailliertere Erklärung zur Abarbeitung von Postfix-Code befindet sich in Kapitel 6.4.

`FLOAT eval_postfix(PCODE *code, EVAL_VARS *vars, FLOAT *stack, FLOAT *reg)`

Eval_Postfix wertet Postfix-Code für eine feste Stelle aus. Es wird nur der Funktionswert und keine Ableitung berechnet. Dadurch sind in dieser Funktion auch Operatoren zulässig, für die keine Ableitung besitzen. Dazu gehören etwa die Booleschen Operatoren, Vergleiche, Bedingungen und Zufallsfunktionen.

```
VOID eval_duv_postfix(PCODE *code, EVAL_VARS_DUV *vars, PNT_DUV *stack,
                                                 PNT_DUV *reg, PNT_DUV *dest)
```

Eval_dUV_Postfix arbeitet wie Eval_Postfix. Allerdings werden zusätzlich zum Funktionswert auch die Ableitungen nach zwei Variablen berechnet. Dadurch ist es nicht mehr möglich, die Operatoren Remainder, Random und Seed zu verwenden.

```
VOID eval_duv_range_postfix(PCODE *code, EVAL_RANGE_VARS_DUV *vars,
         PNT_DUV_RANGE *stack, PNT_DUV_RANGE *reg, PNT_DUV_RANGE *dest)
```

Eval_dUV_Range_Postfix arbeitet wie Eval_dUV _Postfix. Allerdings werden der Funktionswert und die Ableitungen nicht für eine Stelle berechnet, sondern durch Intervallarithmetik für ein ganzes Intervall.
Wegen der Verwendung der Intervallarithmetik können folgende Operatoren nicht mehr verwendet werden: Remainder, Random, Seed, Condition sowie alle Booleschen- und Vergleichsoperatoren.

Weiters sind in XYZ_Eval zwei allgemein nützliche mathematische Operationen definiert:

```
FLOAT raise(FLOAT base, INT nth)
```

Raise erhebt eine reelle Zahl durch fortgesetztes Quadrieren und Multiplizieren zu einer ganzzahligen Potenz. Dabei ist es egal, ob die Basis positiv oder negativ ist.

```
FLOAT quad(FLOAT x, FLOAT y)
```

Quad berechnet eine vorzeichenerhaltende Potenz. Dabei ist Quad(X, Y) äquivalent zu Sgn(X) * (Abs(X)^Y). Diese Funktion ist bei der Definition von Superquadric sehr nützlich.

Das Modul XYZ_Eval.C enthält zur internen Verwendung eine Reihe von Funktionen, die die üblichen mathematischen Funktionen für Intervalle definieren. Diese Funktionen sind nicht dazu gedacht, von außen aufgerufen zu werden, sondern sollen nur die Funktion von Eval_dUV_Range_Postfix unterstützen.

7.10 XYZ_UV.C

XYZ_UV.C enthält die Schnittstelle zwischen den Epipedbäumen und den Flächen in Parameterdarstellung.

```
EPIPED_FUNCT xyz_uv_functions;
EPIPED_FUNCT funct_uv_functions;
```

In Epiped.D wurde der Typ EPIPED_FUNCT deklariert. Er dient dazu, die Funktionen der Flächen für den Epiped-Baum nutzbar zu machen. Es wurden zwei Typen von Flächen in Parameterdarstellung unterschieden. Bei ersten Typ (XYZ_UV) definiert der Benutzer für jede der drei Raumrichtungen eine Funktion über U und V. Im anderen Fall (Funct_UV) sind die Funktionen für die X und die Y Achse fest vorgegeben (X = U; Y = V;) und der Benutzer definiert nur noch eine Funktion F(u,v) für die Z-Richtung. Dieser Sonderfall einer Fläche ist praktisch wichtig und seine getrennte Behandlung bringt Geschwindigkeits-Vorteile.

```
VOID xyz_init(VOID);
```

XYZ_Init ist zur Zeit eine Dummy-Funktion, die nur aus Gründen der Kompatibilität existiert. Es bestünde die Möglichkeit, hier lnit_Parse(..) aufzurufen und die Tabelle der Operatornamen zu initialisieren, doch da der Parser allgemein von FLIRT genutzt wird, geschieht dies bereits in der Startphase des Programms.

```
EPI_SURFACE *xyz_read(UINT mem_id);
```

XYZ_Read liest aus der Bilddatei die Definition einer Fläche in Parameterdarstellung. Allgemein wird eine Fläche, die vom Epipedbaum verwaltet wird, wie folgt deklariert:

```
Typ EpipedTree SURFACE_NAME
  (MaxDepth UINT)+
  (MaxIterations UINT)+
  (DisplayDepth UINT)+
     SURFACE
  (Transform TRANS)+
   Material MATERIAL
  (Texture TEXTURE)+
```

Ausdrücke der Form $(...)^+$ sind optional.

Dabei ist SURFACE_NAME der Name der Flächentyps, so wie in der Name-Komponente des EPIPED_FUNCT-Record angegeben wurde. SURFACE ist die vom Flächentyp abhängige Deklaration der

Fläche. Die Deklaration von Transformationen, Material und Oberflächentextur erfolgt für Flächen genauso wie für alle anderen Körper in FLIRT. Eine Definition der Syntax befindet sich in FLIRT.DOC.

Für Flächen vom Typ XYZ(u,v) sind SURFACE_NAME und SURFACE nun wie folgt definiert:

```
        SURFACE_NAME = XYZ_UV

SURFACE = (Local [NAME = EXPR;])+
          Equations
            X(u,v) = EXPR;
            Y(u,v) = EXPR;
            Z(u,v) = EXPR;
          (U_Range CONST_EXPR CONST_EXPR)+
          (V_Range CONST_EXPR CONST_EXPR)+
```

Oft ist es günstig, eine Formel nicht als eine lange Kette anzuschreiben, sondern sie in kleinere, überschaubare Einheiten (Macros) zu zerlegen und die Gesamtformel dann aus diesen Macros aufzubauen. Unter Local ist es möglich, solche Macros zu erstellen, indem einem Namen eine Formel zugewiesen wird. Diese Teilausdrücke werden intern mit einem Klammernpaar umschlossen und bei der Verwendung des Macros statt des Macronamens in die Zielformel eingesetzt. Durch die folgende Umwandlung der Formel in einen Postfix-Code mit Registern tritt keine Mehrfachauswertung auf. Bei der Definition eines Macros können auch weiter oben definierte Macros verwendet werden. Macronamen beginnen mit einem Buchstaben und bestehen aus maximal 32 Buchstaben, Ziffern oder Unterstrichen ('_').

Unter Equations werden die Formeln für die drei Raumkoordination der Flächenpunkte definiert. Dem EXPR liegt die in Kapitel 7.7 beschriebene Grammatik zugrunde. Allerdings dürfen nicht alle Möglichkeiten genutzt werden. So sind Zufallszahlen, Boolesche Operatoren, Vergleiche und Bedingungen verboten, da es sich dabei nicht um stetige Funktionen handelt. Diese sind allerdings Voraussetzung für glatte Flächen. Weiters dürfen von den Systemvariablen nur U und V verwendet werden.

U_Range und V_Range geben an, innerhalb welcher Grenzen sich die Parameter bewegen. Wenn keine Grenzen angegeben werden, so wird das Einheitsintervall ([0..1]) verwendet.

Ein CONST _EXPR kann entweder eine FLOAT-Zahl sein, oder aber ein in runden Klammem eingeschlossener mathem. Ausdruck, der nur von Konstanten abhängig ist.

```
EPI_SURFACE *funct_uv_read(UINT mem_id);
```

Für Funct_UV_Read gilt im wesentlichen das bei XYZ_UV_Read gesagte, doch enthält die Definition nur noch eine Formel.

SURFACE_NAME = **Funct_UV**

```
SURFACE = (Local [NAME = EXPR;])+
          Function
            F(u,v) = EXPR;
          (U_Range CONST_EXPR CONST_EXPR)+
          (V_Range CONST_EXPR CONST_EXPR)+
```

Die durch Funct_UV_Read erstellte Datenstruktur unterscheidet sich nicht von der durch XYZ_UV Read erstellten. Die Gleichungen für X und Y werden einfach intern auf X(u, v) = U; Y(u, v) = V; und Z(u,v)=F(u, v); gesetzt. Dadurch sind alle übrigen Funktionen, die für XYZ_UV Flächen erstellt wurden, für F(u,v)-Flächen mitverwendbar.

```
VOID xyz_write(EPI SURFACE *surface);
VOID funct_uv_write(EPI_SURFACE *surface);
```

XYZ_Write und Funct_UV-Write geben die Definition einer Fläche im gleichen Format aus, wie es XYZ_Read bzw. Funct_UV_Read lesen.

```
EPI_SURFACE *xyz_copy'(EPI_SURFACE *source, UINT mem_id,
                       MAT3 *obj_to_world, MAT3 *world_to_obj);
```
XYZ_Copy kopiert eine Fläche in Parameterdarstellung. Dabei werden die Parameter Obj_to_World und World_to_Obj nicht genutzt, da der Parameter extern_transform in EPIPED_FUNCT auf TRUE gesetzt ist und daher der Epipedbaum die Transformationen erledigt.

```
VOID xyz_setup(EPI_SURFACE *surface, BOX2 *uv, EPI_PATCH *patch);
```
XYZ_Setup erstellt ein Patch, in dem die gesamte Fläche enthalten ist. Weiters teilt es dem Epipedbaum in uv mit, über welchem Parameterbereich die Fläche definiert ist.

```
VOID xyz_uv_split(EPI_SURFACE *surface, SPLIT_DIR *dir, FLOAT *splitpos,
          EPI_PATCH *parent, EPI_PATCH *small_child, EPI_PATCH *big_child);
VOID xyz_u_split(EPI_SURFACE *surface, FLOAT *u_split, EPI_PATCH *parent,
          EPI_PATCH *small_child, EPI_PATCH *big_child);
VOID xyz_v_split(EPI_SURFACE *surface, FLOAT *v_split, EPI_PATCH *parent,
          EPI_PATCH *small_child, EPI_PATCH *big_child);
```
XYZ..Split zerteilt Flächenstücke entlang von U oder V-Linien in kleinere Teile, und erstellt für diese kleineren Teile wieder Patches. Eine Beschreibung der Parameter ist bereits bei Epiped.D erfolgt.

```
EPI_LEAF *xyz_build_leaf(EPI_SURFACE *surface, UINT mem_id,
                                               EPI_PATCH *patch);
```
Flächen in Parameterdarstellung benötigen keine lokale Daten, um Flächenpunkte mit Ableitungen für die Newton-Iteration zu berechnen. Daher liefert diese Funktion stets einen NULL-Pointer zurück.

```
VOID xyz_bound(EPI_SURFACE *surface, EPI_PATCH *patch, VEC3 dir,
                                                 BOXl *min_max);
```
XYZ_Bound berechnet die Lage von zwei Ebenen, die normal auf dir stehen und zwischen denen ein Patch vollständig eingeschlossen ist. Wie dabei im Detail vorgegangen wird, ist in Kapitel 5 beschrieben.

```
VOID xyz_duv_eval(EPI_SURFACE *xyz_uv, PNT2 uv, PNT3 *xyz,
                                                 VEC3 *du, VEC3 *dv);
VOID funct_uv_duv_eval(EPI_SURFACE *xyz_uv, PNT2 uv, PNT3 *xyz,
                                                 VEC3 *du, VEC3 *dv);
```
Diese Funktionen berechnen die Koordinaten eines Flächenpunkts sowie die Ableitungen in U- und V-Richtung an dieser Stelle. Da die Auswertung der Formeln für die Newton-Iteration einen spürbaren Teil der Rechenzeit eines Bildes verbraucht, wurden für XYZ(u,v) und F(u,v) Flächen zwei verschiedene Auswerteroutinen geschrieben. Dadurch ist es möglich, die X und Y Komponente eines Punktes für F(u, v)-Flächen direkt in C zu berechnen ohne Postfix-Code auswerten zu müssen.

```
VOID xyz_end(EPI_SURFACE *surface);
```
XYZ_End gibt eine Statistik für eine Fläche aus. Dabei wird ausgegeben, wie oft die Fläche gespalten wurde, wie oft Flächenstücke zwischen Ebenen eingesperrt wurden und wie oft ein Flächenpunkt mit seinen Ableitungen berechnet wurde.

```
VOID xyz_exit(VOID);
```
XYZ_Exit existiert nur aus Kompatibilitätsgründen und führt keine Operation aus.

7.11 Externe Anwendungen des Parsers

Im Rahmen dieser Diplomarbeit wurde ein sehr mächtiger Parser mit schneller Auswertung implementiert, der für viele Zwecke eingesetzt werden kann. Das folgende Kapitel beschreibt, wie andere Module die Funktionen des Parsers nutzen können.

Im einfachsten Fall wird der Parser dazu benutzt, mathematische Ausdrücke bei der Angabe von reellen Konstanten in der Bilddatei zu zulassen. Dazu existiert die Funktion Read_RealExpr, die entweder eine Zahl oder einen in runden Klammern eingeschlossen mathematischen Ausdruck von der Datei liest, und den Wert zurückliefert. Wenn

der Formel-String nicht aus der Bilddatei, sondern aus einer anderen Quelle stammt, so bestimmt Eval_Const_Expr den Wert des Ausdrucks.

Diese beiden Funktionen arbeiten nur mit Konstanten und Benutzervariablen, nicht aber mit den Systemvariablen (T, U, V, W, X, Y, Z) zusammen. Weiters gilt, daß diese alle Arbeiten von der Zeichenkette bis zur Auswertung bei jedem Aufruf erneut durchführen. Wenn mehrfach der gleiche Ausdruck ausgewertet werden soll, sind diese Routinen daher zu langsam.

Um einen Ausdruck oder eine Gruppe zusammengehöriger Ausdrücke schnell auszuwerten, ist es nötig die Umwandlung von der Zeichenkette in einen Postfix-Code und die Auswertung dieses Codes zu trennen. Dabei wird wie folgt vorgegangen:

1.Schritt: Formeln → Postfix

```
#include "xyzparse.d"
#include "xyzparse.f"

VOID step_one(UCHAR *equation[], PCODE *postfix[])
 {
  PARSE_NODE hash_table[HASH_SIZE];
  PARSE_PERMIT permit;
  PARSE_NODE tree[...];   /* Pro Formel ein PARSE_NODE */
  PCODE pfix[MAX_EQU_COMPLEXITY + 2];
  UCHAR *parse;
  UINT len, reg_cnt;

  init_parse();           /* Einmal zu Beginn des Programms */
  init_hash(hash_table);  /* Einmal vor jeder Formel-Gruppe */
  permit = ...            /* Gültige Operatoren definieren */
  for (i =...)            /* Für jede Formel (Reihenfolge frei) */
   {
    parse = equation[i];  /* Parse: Zeiger auf aktuelles Zeichen */

    tree[i].op = DONE;
    tree[i].u.operand = parse_expr(hash_table, equation[i],
                                   &parse, permit, NULL);

    if (*parse == ...)    /* Test auf Zeichen hinter dem Ende */
      fatal ("Unexpected Char '%c' after Expression", *parse);
   }

  for (i = ...)           /* Für jede Formel (Reihenfolge frei) */
    opt_ptree(&tree[i], hash_table);

  reg_cnt = 0;
  for (i = ...)           /* Für jede Formel (Reihenfolge wichtig) */
   {
    len = 0;
    ptree_to_postfix(&tree[i], DONE, hash_table, &reg_cnt, TRUE,
                     pfix, &len, MAX_EQU_COMPLEXITY);

    postfix[i] = (PCODE*) newmem(... , (len + 2) * sizeof(PCODE));
    memcpy(postfix[i], pfix, (len + 2) * sizeof(PCODE));
   }
 }
```

Schritt 2: Auswerten der Postfix-Codes

```
#include "xyzparse. d"
#include "xyzparse.f"

VOID step_two(PCODE *postfix[])
 {
  EVAL_VARS vars;
  FLOAT stack[MAX EQUATION COMPLEXITY / 2 + 3];
  FLOAT reg[MAX_EQUATION_COMPLEXITY / 2 + 3];

  vars = ...;    /* Werte der verwendeten Systemvariablen */

  for (i = ...)  /* Gleiche Reihenfolge wie bei PTree_To_Postfix */
    value[i] = eval_postfix(pfix, &vars, stack, reg);
           /* Wenn keine Systemvar. verwendet werden,
              kann auch NULL statt (&vars) übergeben werden */
 }
```

Weitere Informationen über die Vorgangsweise bei der Auswertung von Formeln sind in der Funktion Eval_Const_Expr zu finden. Diese führt nach obigem 'Strickmuster' die Schritte 1 und 2 direkt hintereinander aus.

8. Weitere Arbeitsgebiete

Bisher wurden stets Flächen in Parameterdarstellung, also in der Form XYZ(u, v), behandelt. Eine Möglichkeit, die Flächendefinition zu erweitern, wäre, implizite Flächen zuzulassen. Diese sind über eine Formel der Form F(X, Y,Z)=0 definiert, wobei es sinnvoll sein kann, zusätzlich Grenzen für die Werte von X, Y und Z anzugeben, um unendliche Flächen zu begrenzen. Als Beispiel für eine (endliche) implizit gegebene Fläche soll hier die Einheitskugel in ihrer Definition über den Abstand der Flächenpunkte vom Mittelpunkt angegeben werden: $X^2+Y^2+Z^2-1=0$. Um die Schnittpunkte eines Strahls mit einer solchen Fläche zu berechnen, kann die Parameterdarstellung des Strahls verwendet werden: $S = (x_0,y_0,z_0) + t*(\Delta x,\Delta y,\Delta z)$. In Komponenten zerlegt also $x = x_0 + t*\Delta x$, $y = y_0 + t*\Delta y$, $z = z_0 + t*\Delta z$. Diese Formeln für XYZ(T) können nun in die implizite Flächengleichung eingesetzt werden. Man erhält so eine Formel in T, deren Nullstellen aufzusuchen sind. Für ein endliches T -Intervall ist dies mit Intervallarithmetik leicht möglich. Eine genauere Beschreibung dieses Themenkreises befindet sich in [MIT90]. Danach bleibt noch das Problem offen, den Normalvektor auf die Fläche am Schnittpunkt zu bestimmen.

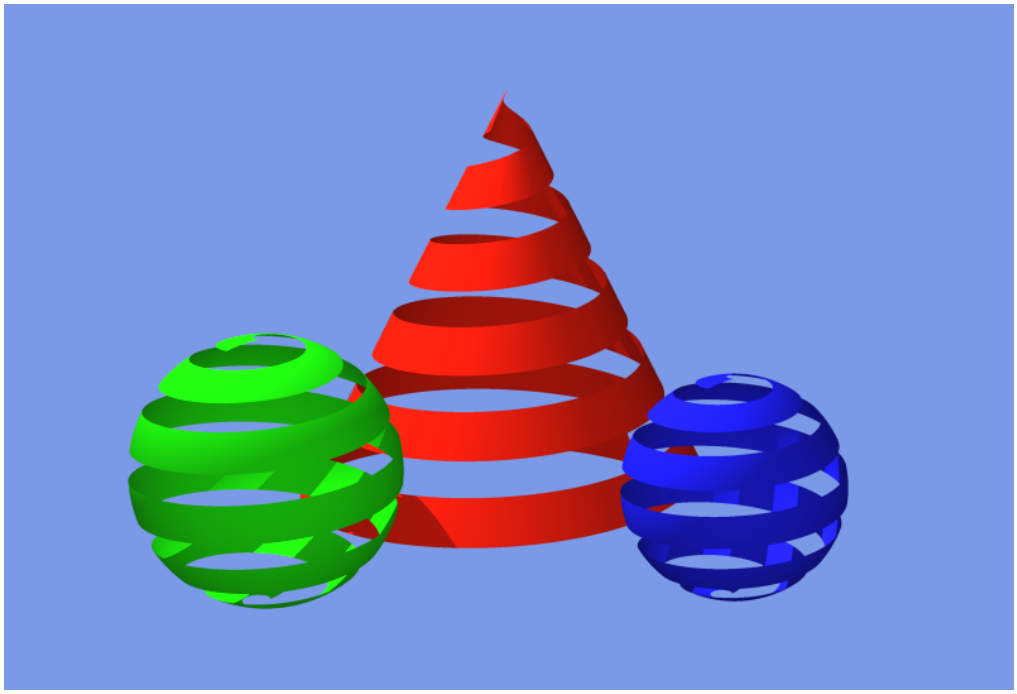
Bei der Berechnung der begrenzenden Ebenen in XYZ_Bound_Surface werden bisher die Intervalle der Funktionswerte und der ersten Ableitungen berücksichtigt. Dieses lineare Modell eignet sich gut für fast ebene Flächenstücke. Um auch stärker gekrümmte Flächenstücke eng umschließen zu können, wäre es möglich, auch Ableitungen höheren (n-ten) Grades zu berücksichtigen. Die Fläche in Parameterdarstellung wäre dann zwischen zwei Flächen n-ten (polynom-) Grades eingeschlossen, deren n-te Ableitungen der minimalen bzw. maximalen Ableitung der eingeschlossenen Fläche entsprechen. Ob Flächenstücke enger durch Näherungsflächen hohen oder niederen Grades umschlossen werden, wurde in Kapitel 5.2 für den Fall Funktionswerte vs. erste Ableitung besprochen. Bei der Verwendung höherer Ableitungen ergibt sich das Problem, daß die umschließenden Flächen selbst nicht immer einfach mit einer Ebene begrenzbar sein müssen, und daß Extremwerte auch im Inneren oder am Rand und nicht nur in den Ecken des Flächenstücks auftreten können. Außerdem sind zur Bestimmung der Schranken der höheren Ableitungen immer mehr Operationen notwendig. Da mit Intervallarithmetik gerechnet wird, und bei jeder Operation das Risiko einer Überschätzung besteht, werden die Schranken für die hohen Ableitungen immer gröber.

Die Ebenen der Epipede könnten aber auch auf eine grundsätzlich andere Art bestimmt werden. Dazu werden die Flächenpunkte mit minimalen und maximalen Abstand zu einer, zur (späteren) Epipedseite parallelen Ebene aufgesucht. Für diese Punkte gilt, daß die Ableitung der Anstandsfunktion sowohl in U als auch in V Richtung Null ist. Weiters müssen Rand- und Eckextrema gesucht werden. Alle diese Fälle lassen sich mit symbolischer Differentiation, Intervallarithmetik und Intervallschachtelung leicht lösen. Wenn nun die Epipedseiten durch die Punkte mit minimalem und maximalem Abstand gelegt werden, so schließen sie das Flächenstück, für die gegebene Ebenenrichtung optimal ein.

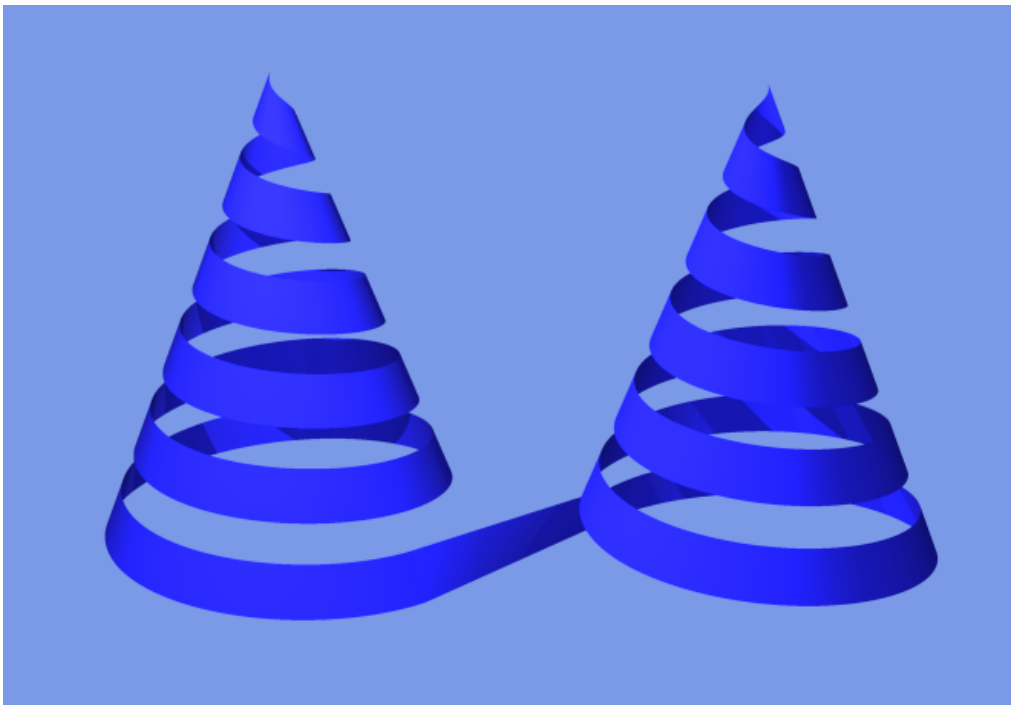
Weiters könnten die Flächen zu soliden Volumen erweitert werden. Dazu wäre es etwa möglich, einen achsenparallelen Quader an einer Fläche der Form Z(X, Y) in zwei Teile zu zerlegen und einen Teil zu verwerfen. Die so entstehenden 'Insel-Körper sind aus dem Bereich der Geschäftsgrafiken bekannt. Andererseits könnten Flächen, die ein Inneres haben, gefüllt werden. Dabei besteht das Problem, Flächen, für die eine eindeutige Definition von innen und außen existiert, zu erkennen.

9. Praktische Beispiele

9.1 Bilder

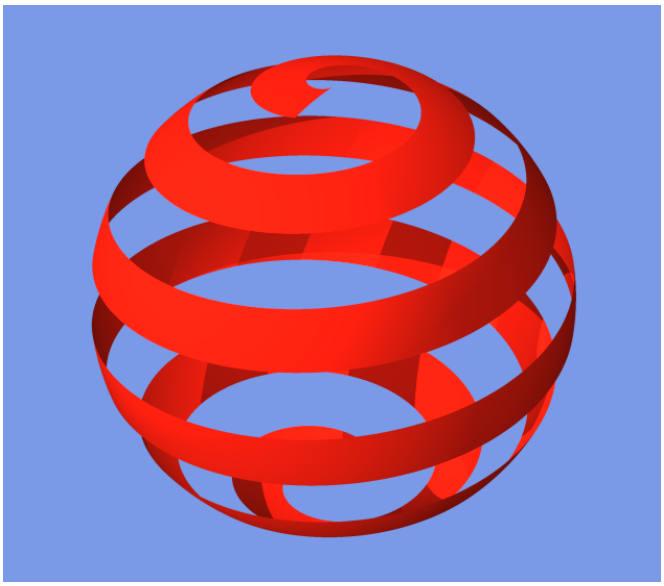


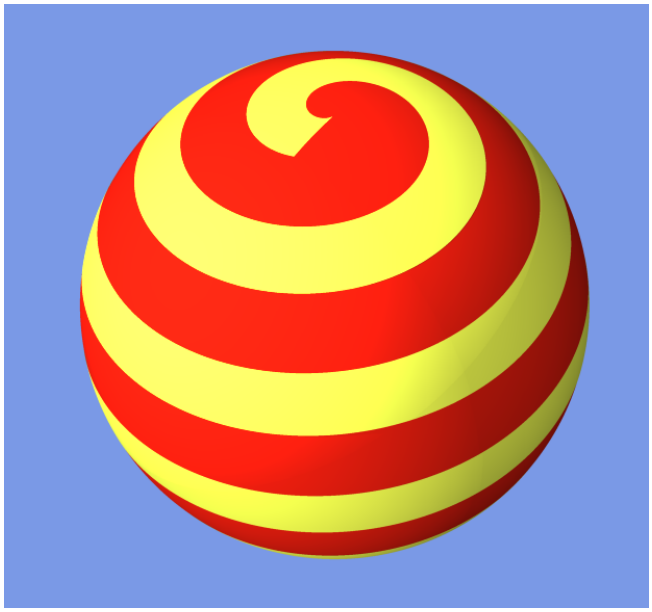
Scene #1



Twin Peaks

Ball

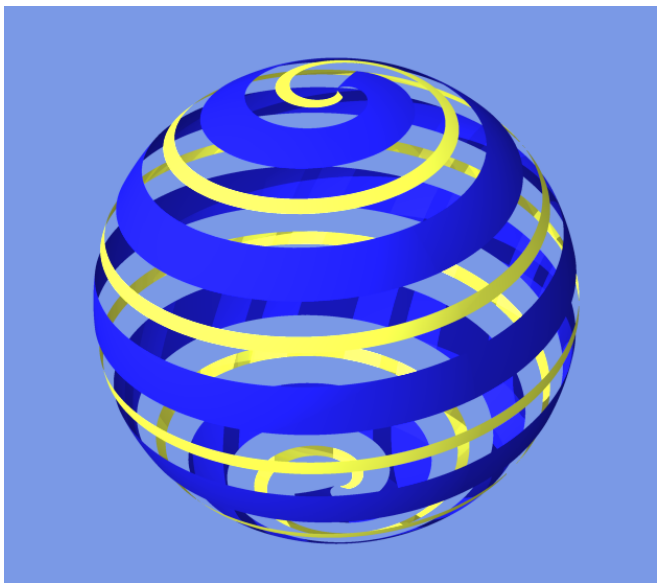




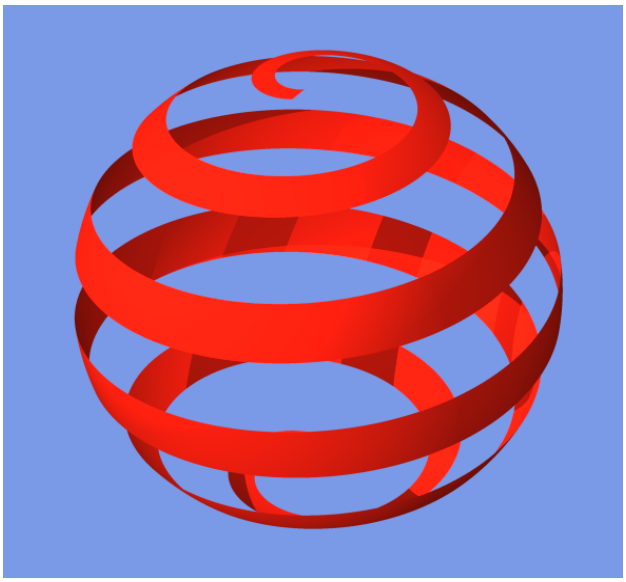
Sphere_Ball

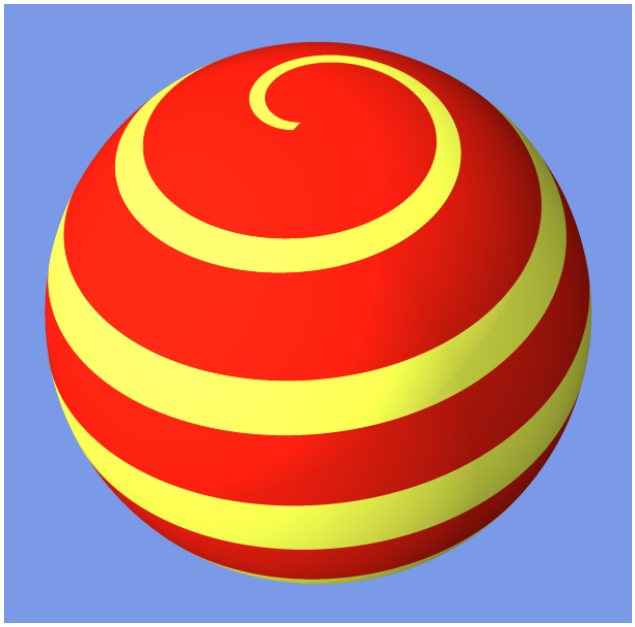
X(U, V) = Sin(12 * U) * Sin(U + V);
Y(U, V) = Cos(12 * U) * Sin(U + V);
Z(U, V) = Cos(U + V);
U = [0.12...3.02] ~ V = [-0.12...0.12]

Double_Ball



VBall

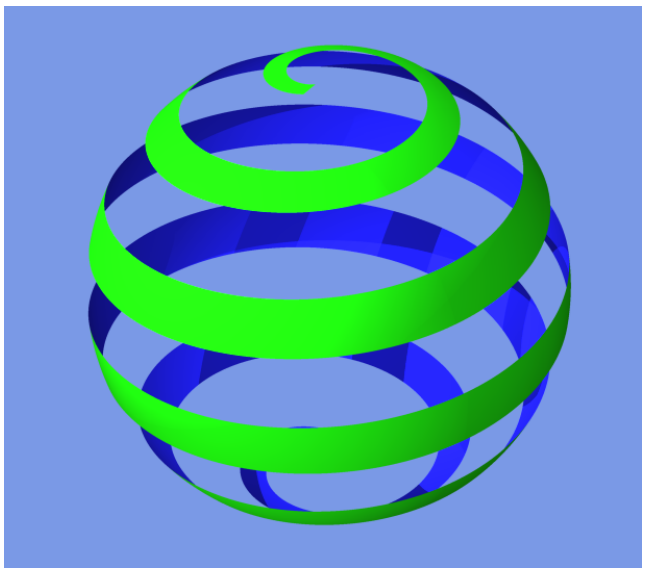




Sphere_VBall

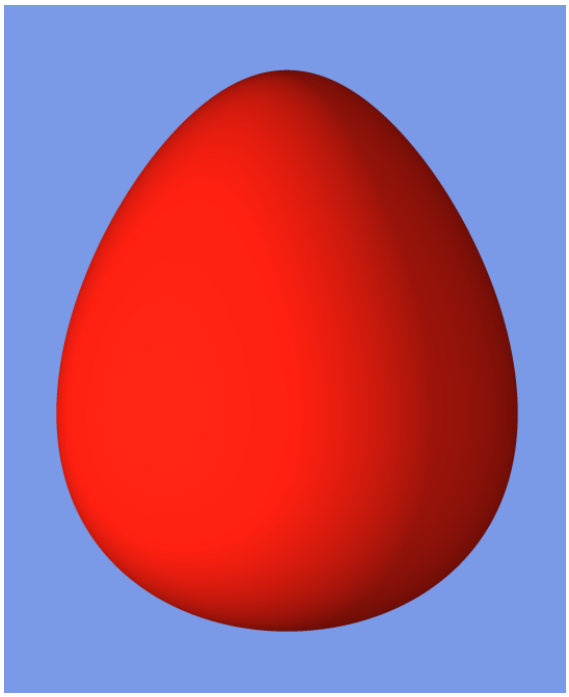
B = 1.1 - Abs(U –π/2) / (π/2)
X(U, V) = Sin(12 * U) * Sin(U + V *B);
Y(U, V) = Cos(12 * U) * Sin(U + V * B);
Z(U, V) = Cos(U + V * B);
U = [0.12...3.02] ~ V = [-0.12...0.12]

VBall_Pat



Ei

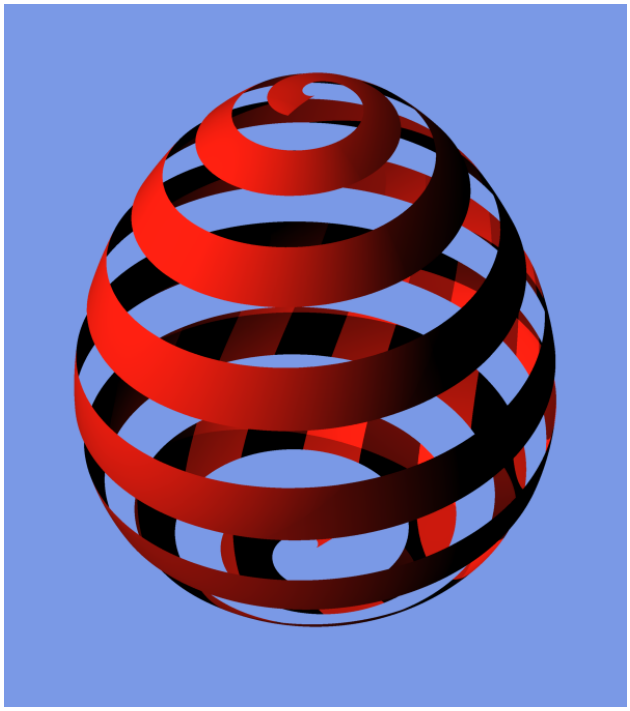
F = (4 - Cos(V)) /5;
X(U, V) = Cos(U) * (Sin(V) * F);
Y(U, V) = Sin(U) * (Sin(V) * F);
Z(U, V) = Cos(V);
U = [-π...π] ~ V = [0.0...π]

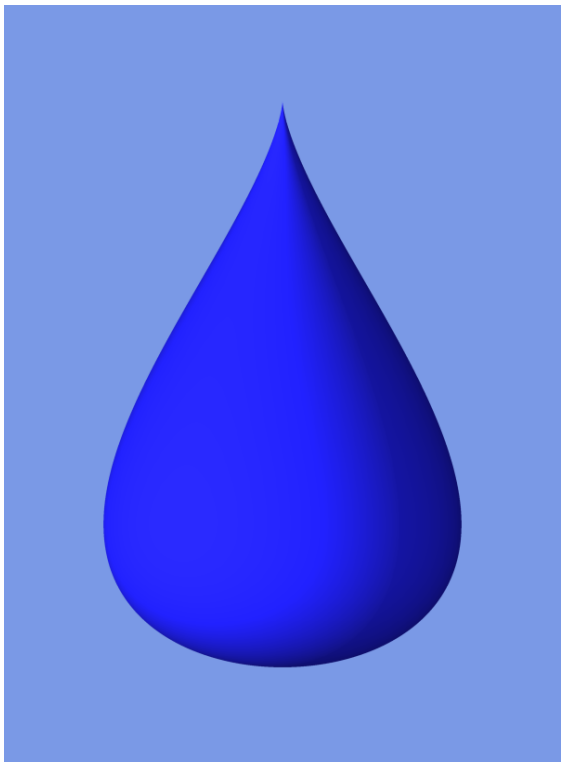


Egg_Ei

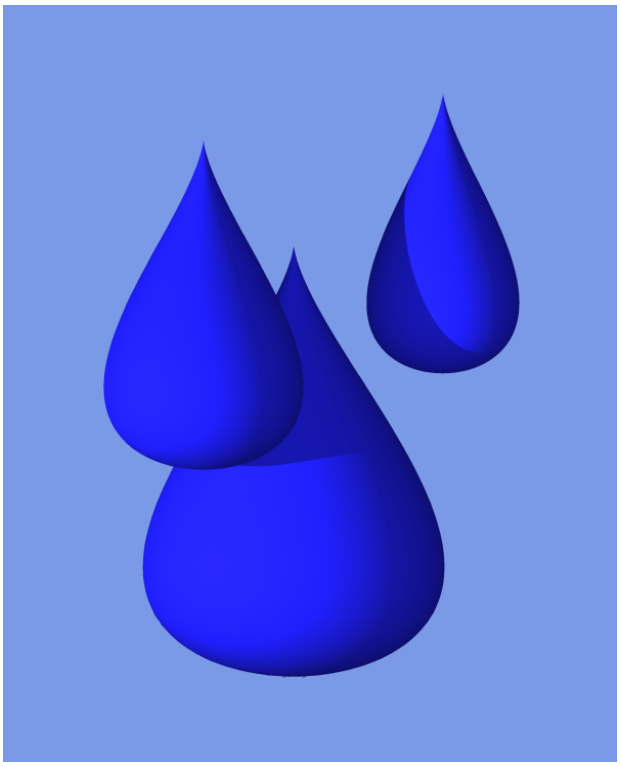
Spiral_Egg

F = (4 - Cos(U + V)) /5;
X(U, V) = Sin(16 * U) * (Sin(U + V) * F);
Y(U, V) = Cos(16 * U) * (Sin(U + V) * F);
Z(U, V) = Cos(U + V);
U = [0.1...3.04] ~ V = [-0.1...0.1]





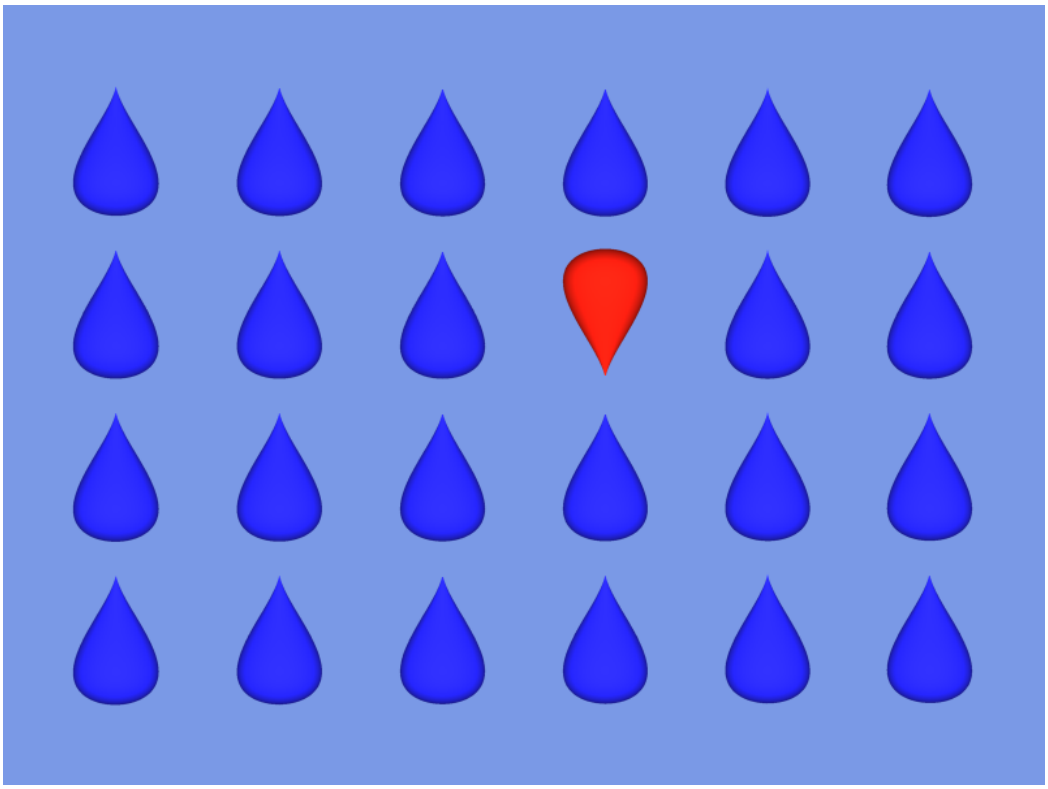
Drop



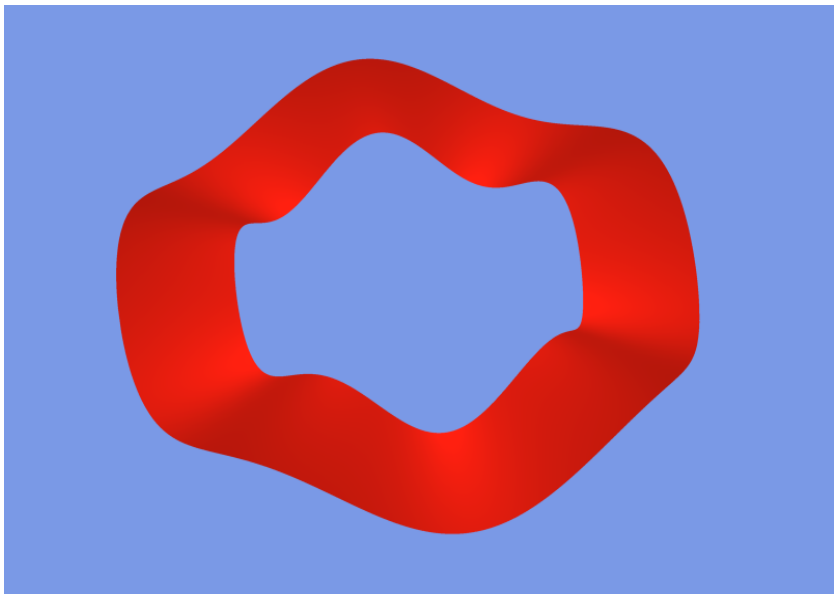
Rain

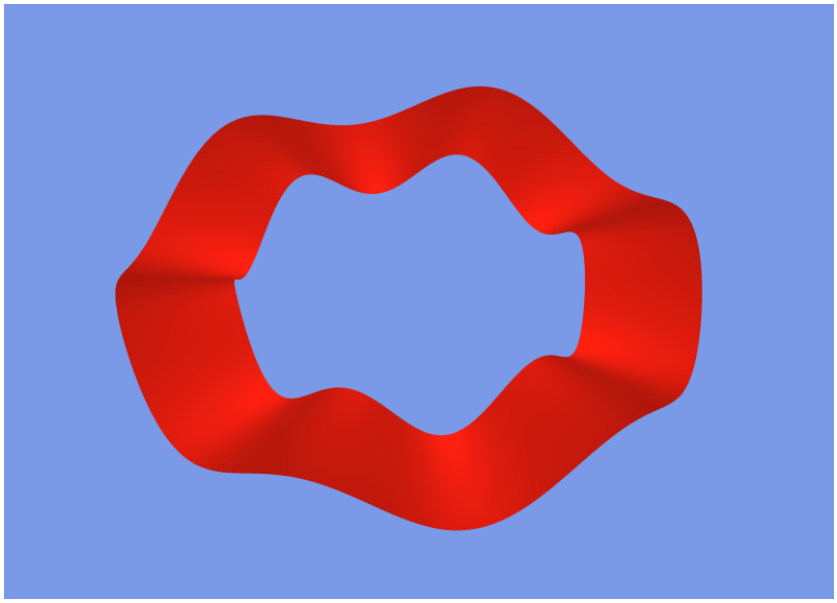
$$X(U, V) = \cos(U) * (\sin(V) * (1 - \cos(V) / 2);$$
$$Y(U, V) = \sin(U) * (\sin(V) * (1 - \cos(V) / 2);$$
$$Z(U, V) = \cos(V);$$
$$U = [-\pi...\pi] \sim V = [0...\pi]$$

Drip



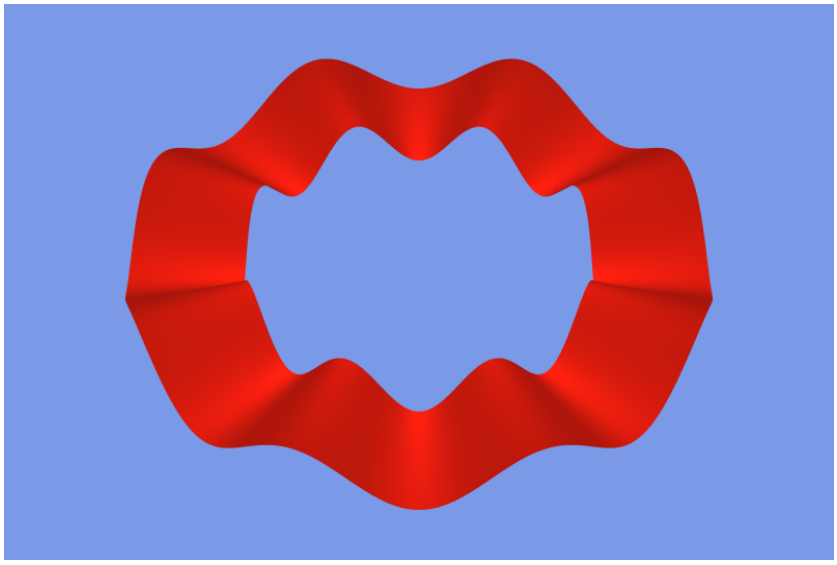
Wave 5
X(U, V) = Sin(U) * V;
Y(U, V) = Cos(U) * V;
Z(U, V) = Cos(5 * U) * 0.1;
U = [0.0...2π] ~ V = [0.6...1.0]





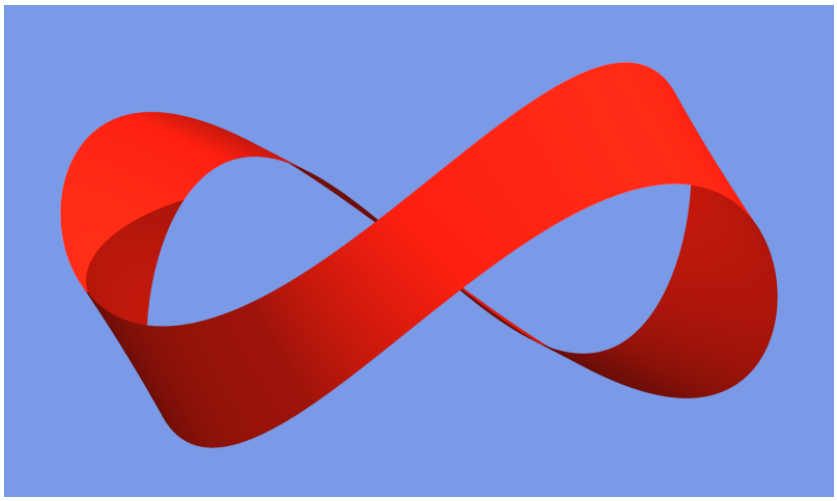
Wave 6
X(U, V) = Sin(U) * V;
Y(U, V) = Cos(U) * V;
Z(U, V) = Cos(6 * U) * 0.1;
U = [0.0...2π] ~ V = [0.6...1.0]

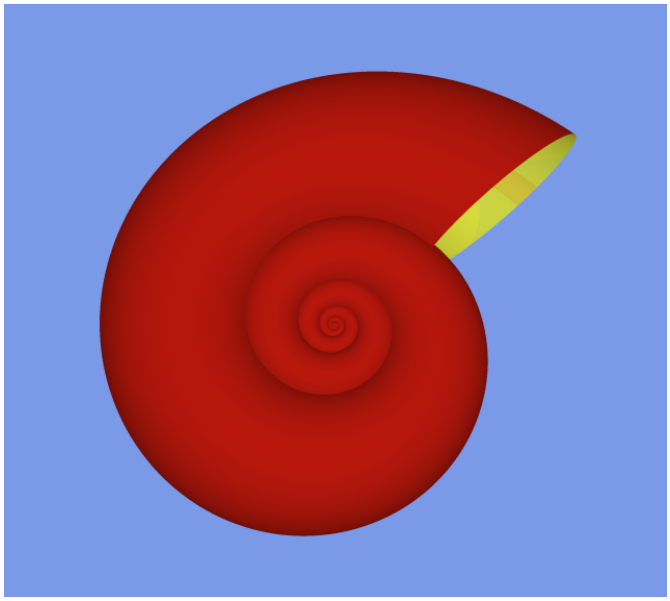
Wave 8
X(U, V) = Sin(U) * V;
Y(U, V) = Cos(U) * V;
Z(U, V) = Cos(8 * U) * 0.1;
U = [0.0...2π] ~ V = [0.6...1.0]



Möbius

X(U, V) = Cos(V) * (1 + Cos(V/2) * U);
Y(U, V) = Sin(V) * (1 + Cos(V/2) * U);
Z(U, V) = Sin(V/2) * U + 0.4 * Sin(2*V);
U = [-0.2...0.2] ~ V = [0.0...2π]



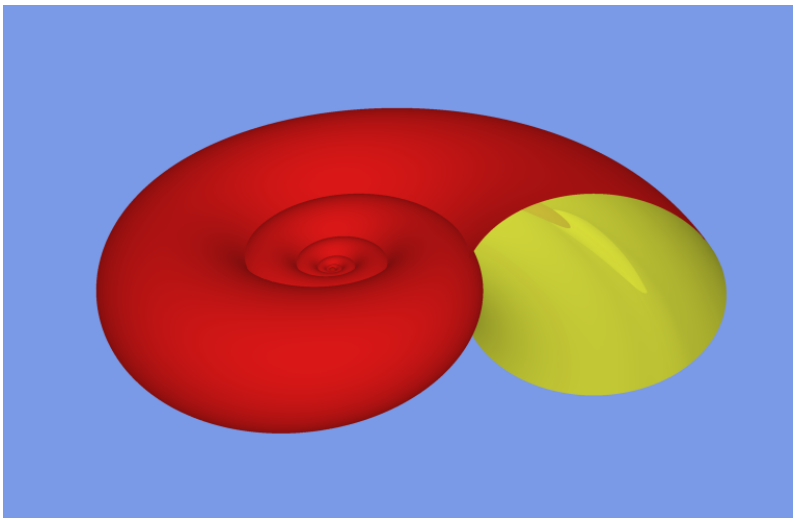


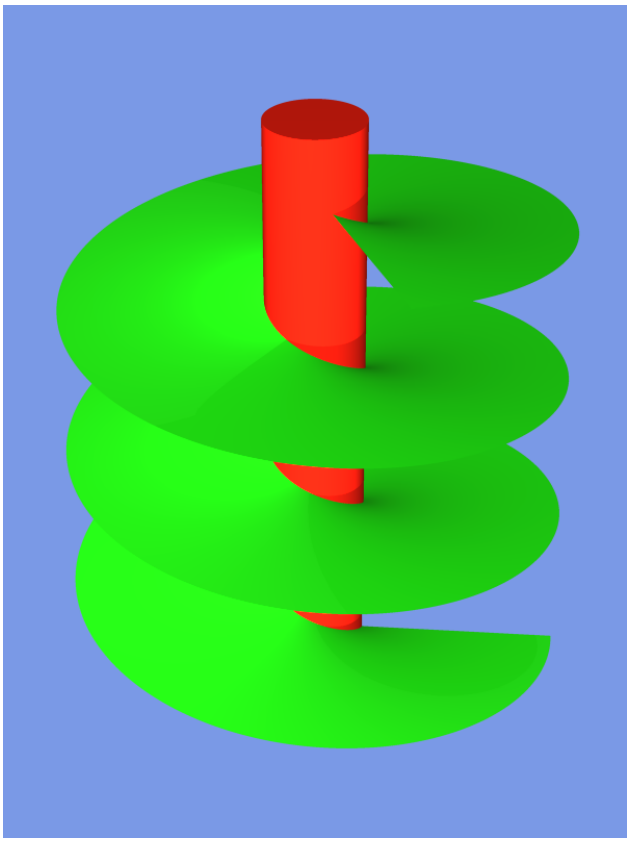
Nautilus #1

R=2 ^ (U / 5)
X(U, V) = Sin(U) * (R * (1 + Cos(V) * 0.47));
Y(U, V) = Cos(U) * (R * (1 + Cos(V) * 0.47));
Z(U, V) = R * Sin(V) * 0.47;
U = [-5.0...30.0] ~ V = [-π...π]

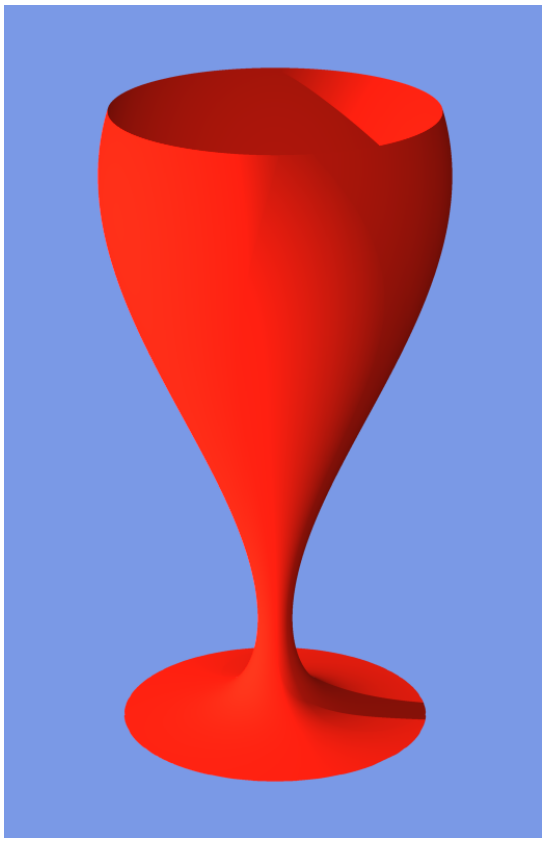
Nautilus #2

R=2 ^ (U / 5)
X(U, V) = Sin(U) * (R * (1 + Cos(V) * 0.47));
Y(U, V) = Cos(U) * (R * (1 + Cos(V) * 0.47));
Z(U, V) = R * Sin(V) * 0.47;
U = [0.0...31] ~ V = [-π...π]



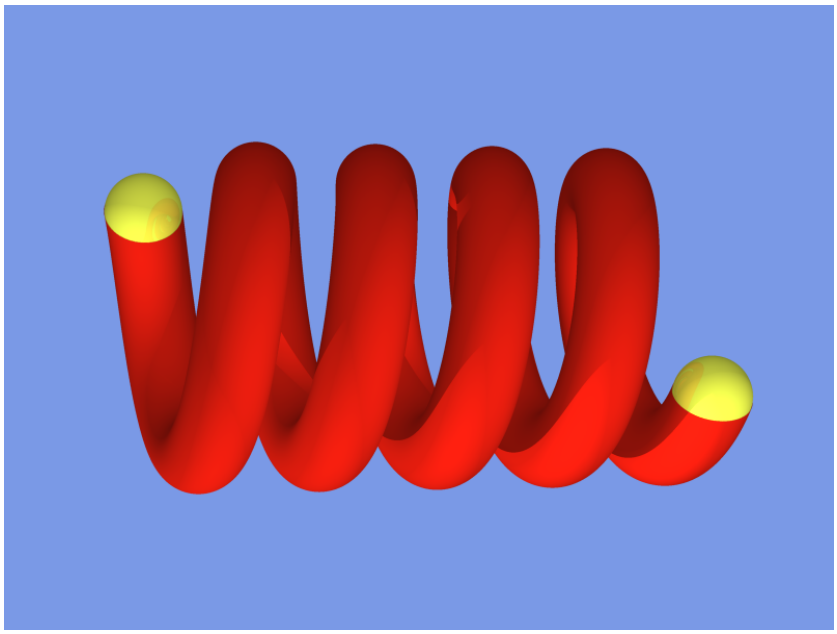


Screw
X(U, V) = Sin(2 * U) * V;
Y(U, V) = Cos(2 * U) * V;
Z(U, V) = U;
U = [-0.25...9.75] ~ V = [1.0... 5.0]



Kelch
R = (((U - 6) * U + 2) * U - 1) /20 –
0.07 / (U + 0.5);
X(U, V) = R * Sin(V);
Y(U, V) = R * Cos(V);
Z(U, V) = U;

Coil
X(U, V) = Sin(U) * (1 + 0.3 * Sin(V));
Y(U, V) = Cos(U) * (1 + 0.3 * Sin(V));
Z(U, V) = U /7 + 0.3 * Cos(V);
U = [1.4...30.0] ~ V = [-π...π]

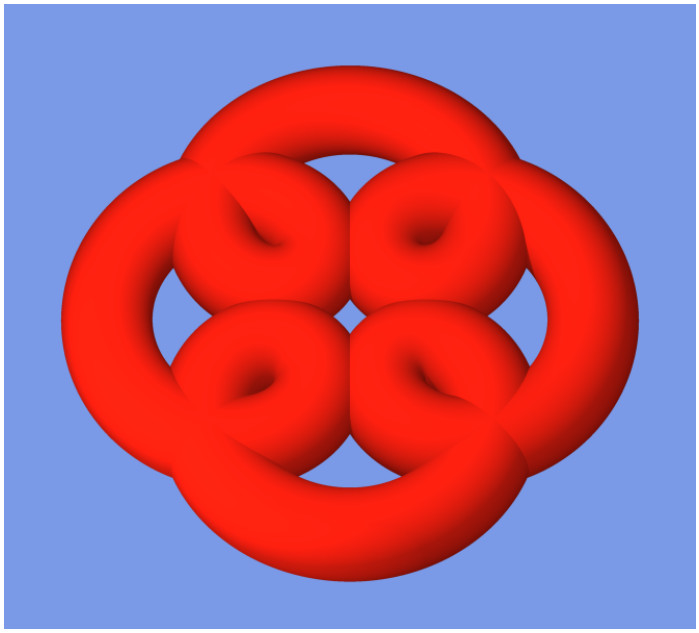


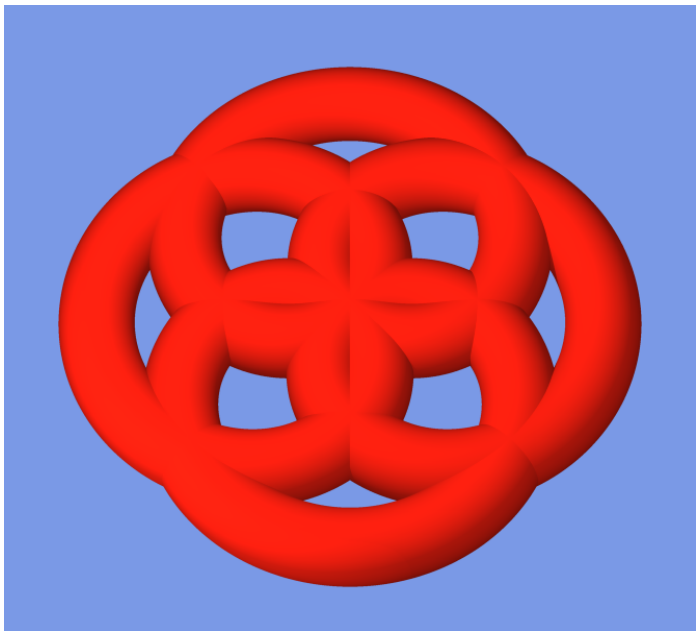
Kugel-Sweep entlang einer Trochoiden:

A = ...
B = ...
R = ...
X = Cos(U) + B * Cos(A * U);
Y = Sin(U) + B * Sin(A * U);
Dx = Neg(Sin(U) + A * B * Sin(A * U));
Dy = Cos(U) + A * B * Cos(A * U);
L = Sqrt(Dx * Dx + Dy * Dy + 0.001);

X(U, V) = X + Dy * (Cos(V) * R / L);
Y(U, V) = Y - Dx * (Cos(V) * R / L);
Z(U, V) = Sin(V) * R;
U = [-π...π] ~ V = [-π...π]

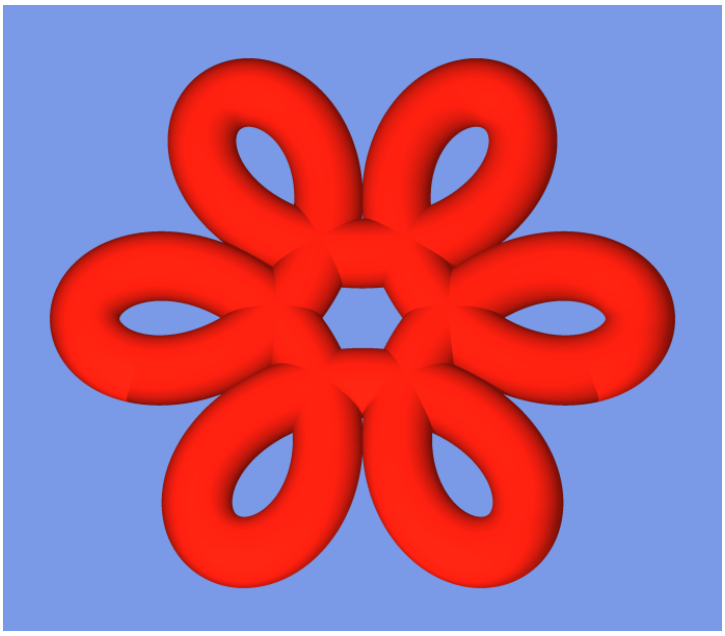
Sweep #1
A = 5;
B = 0.6;
R = 0.3;

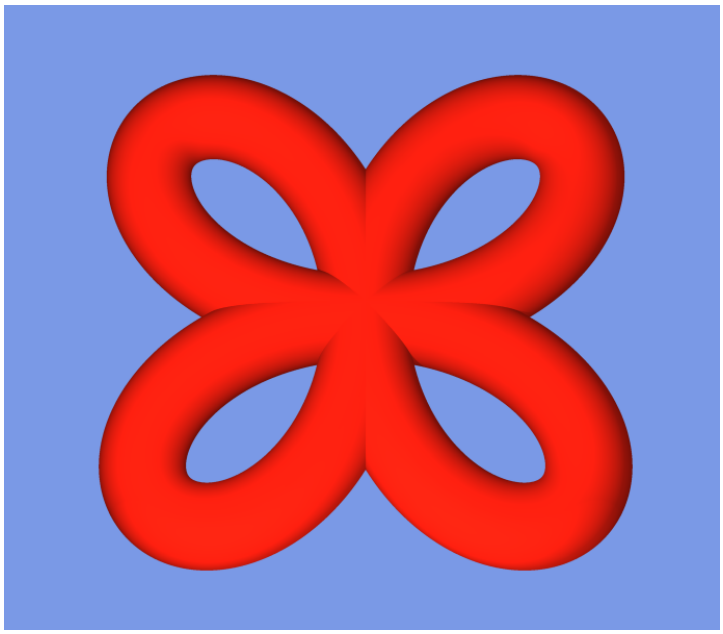




Sweep #2
A = 5;
B = 1;
R = 0.3;

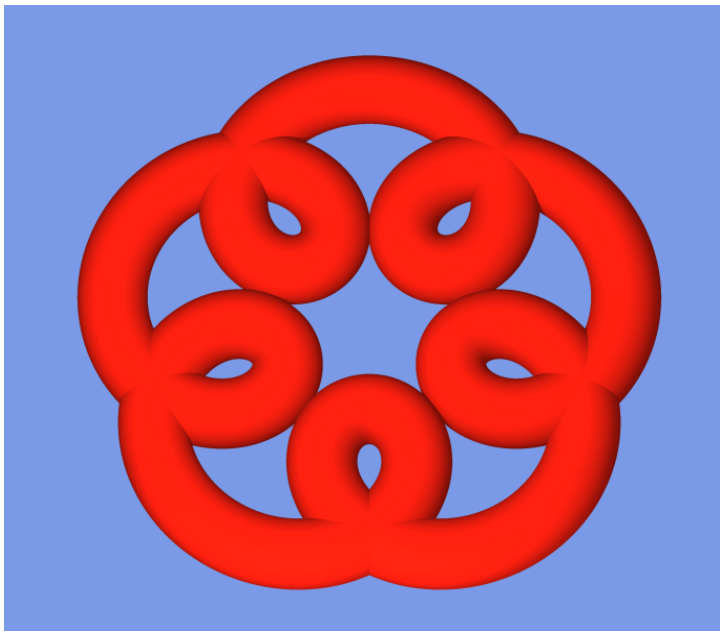
Sweep #3
A = -5;
B = 0.6;
R = 0.2;



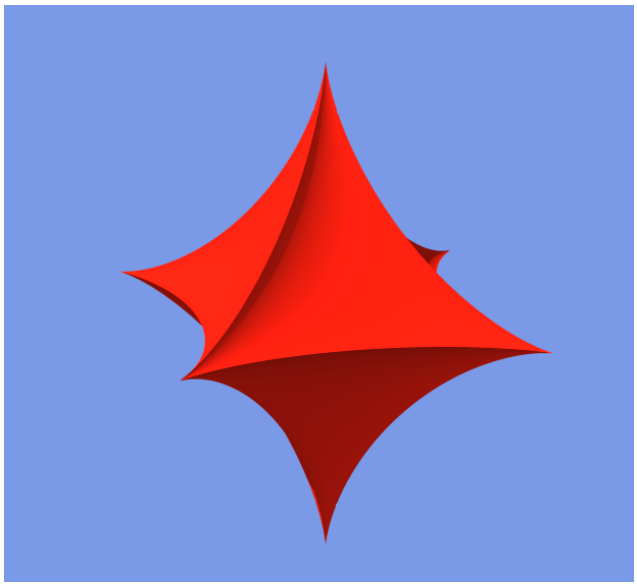


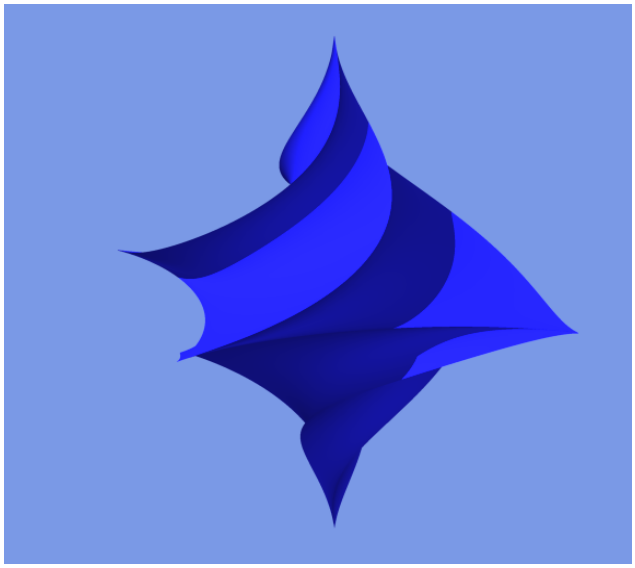
Sweep #4
A = -3;
B = 1;
R = 0.3;

Sweep #5
A = 6;
B = 0.5;
R = 0.2;



Quadric 3
X(U, V) = Quadric(Cos(U) * Sin(V), 3);
Y(U, V) = Quadric(Sin(U) * Sin(V), 3);
Z(U, V) = Quadric(Cos(V), 3);
U = [-π...π] ~ V = [0.0...π]

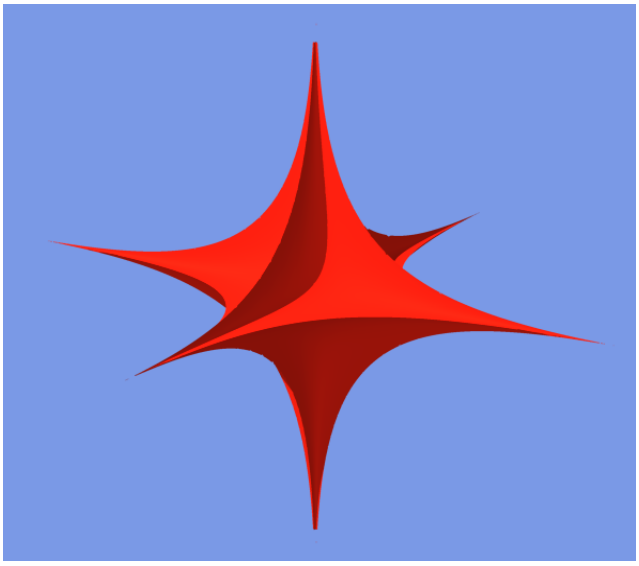




Twisted
X = Quadric(Cos(U) * Sin(V), 3);
Y = Quadric(Sin(U) * Sin(V), 3);
Z = Quadric(Cos(V), 3);
W = Z * π/2;

X(U, V) = X * Cos(W) + Y * Sin(W);
Y(U, V) = X * Sin(W) - Y * Cos(W);
Z(U,V) = Z;
U = [-π...π] ~ V = [0.0...π]

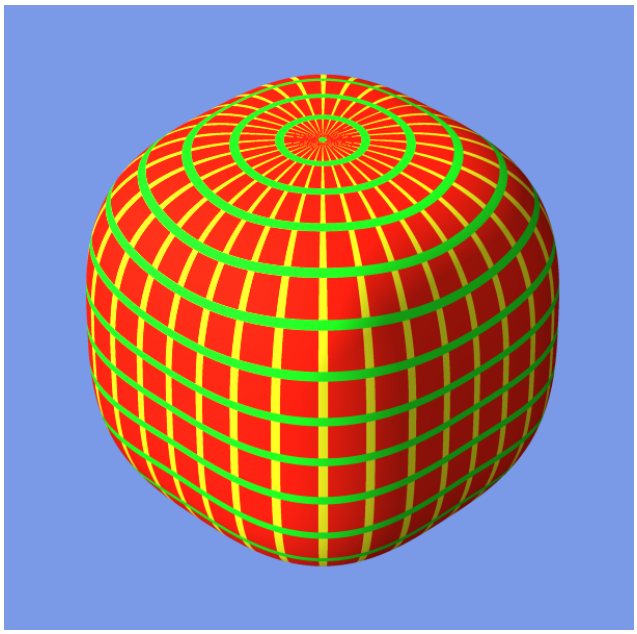
Quadric 5
X(U, V) = Quadric(Cos(U) * Sin(V), 5);
Y(U, V) = Quadric(Sin(U) * Sin(V), 5);
Z(U, V) = Quadric(Cos(V), 5);
U = [-π...π] ~ V = [0.0...π]

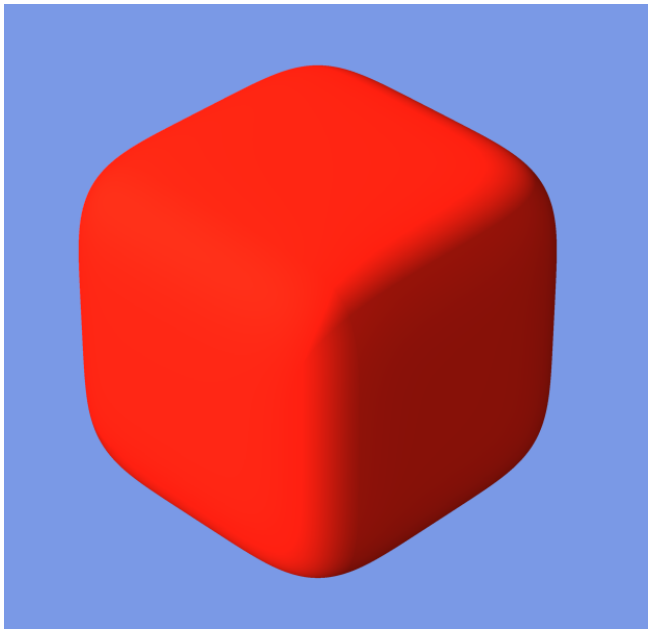


Quadric 0.333

X = Cos(U) * Sin(V);
Y = Sin(U) * Sin(V);
Z = Cos(V);
R = Cbrt(Abs(X) ^ 3 + Abs(Y) ^ 3 + Abs(Z) ^ 3);

X(U,V) = X / R;
Y(U,V) = Y / R;
Z(U,V) = Z / R;
U = [-π...π] ~ V = [0.0...π]





Quadric 0.20

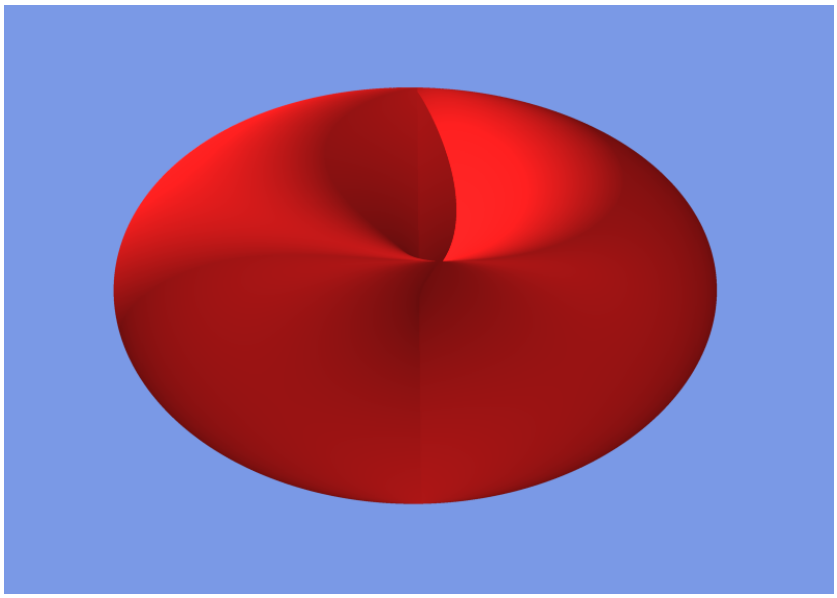
X = Cos(U) * Sin(V);
Y = Sin(U) * Sin(V);
Z = Cos(V);
R = (Abs(X) ^ 5 + Abs(Y) ^ 5 + Abs(Z) ^ 5) ^ (1 / 5);

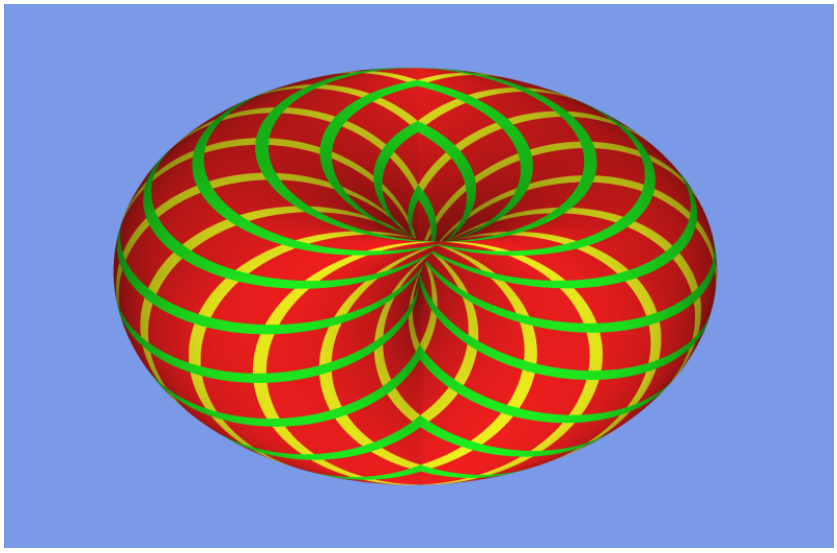
X(U, V) = X / R;
Y(U,V) = Y / R;
Z(U,V) = Z / R;
U = [-π...π] ~ V = [0.0...π]

Quadric 0.125

X = Cos(U) * Sin(V);
Y = Sin(U) * Sin(V);
Z = Cos(V);
R = (X ^ 8 + Y ^ 8 + Z ^ 8) ^ (1 / 8);
X(U,V) = X / R;
Y(U,V) = Y / R;
Z(U,V) = Z / R;
U = [-π...π] ~ V = [0.0...π]

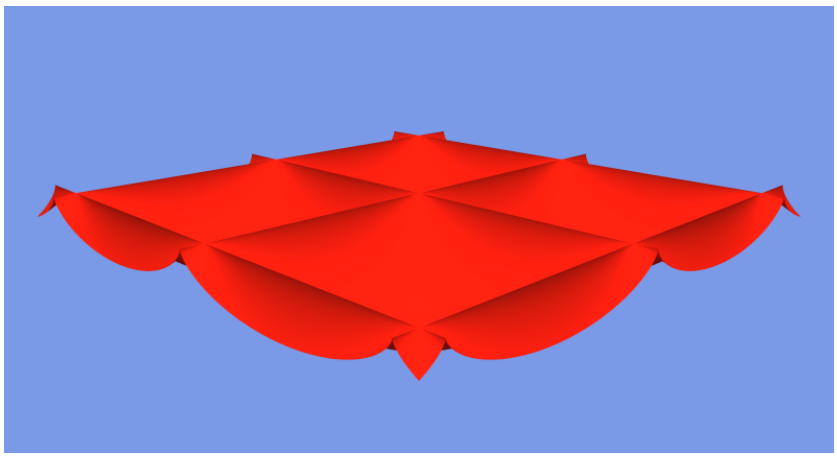
CCircle #1
X(U, V) = Sin(U);
Y(U, V) = Cos(V);
Z(U, V) = Cos(U) + Sin(V);
U = [0.0...2π] ~ V = [0.0...2π]





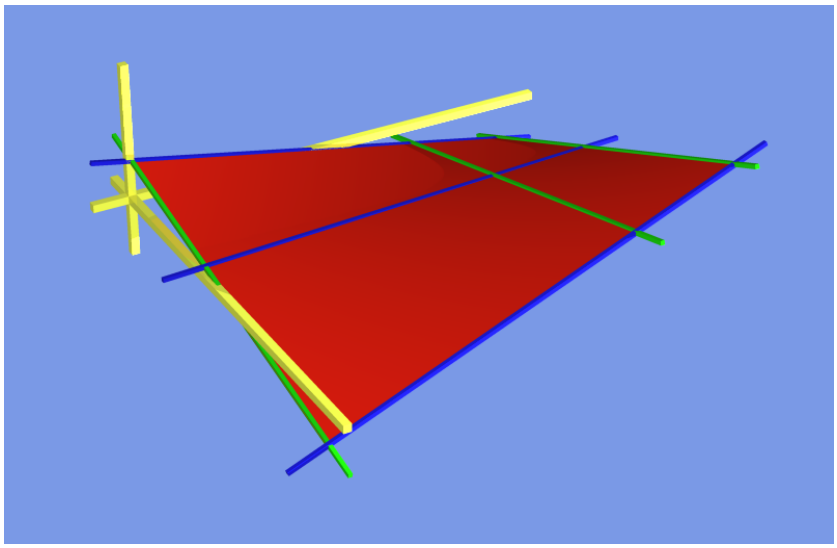
CCircle #2

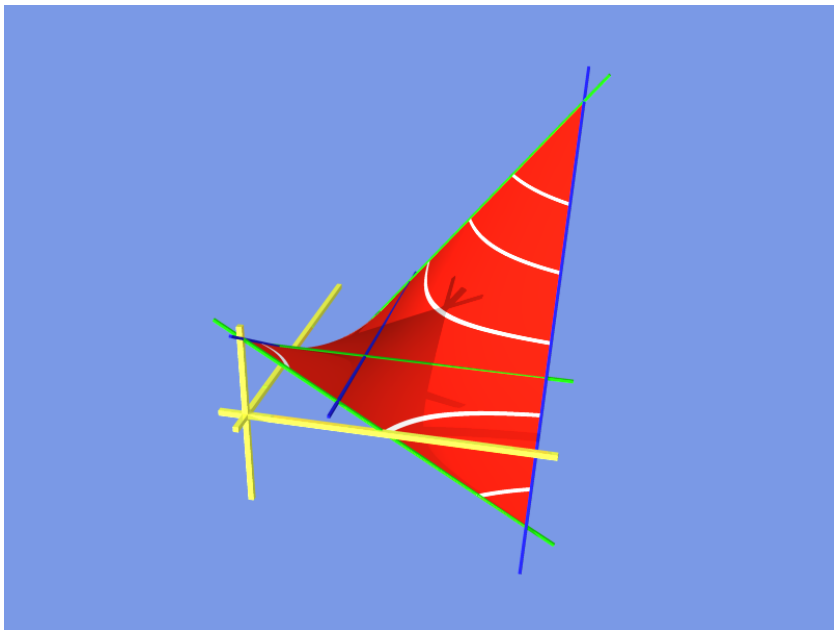
Crest
X(U, V) = U + Cos(U);
Y(U, V) = V + Cos(V);
Z(U, V) = -0.5 * (1 - Sin(U)) * (1 - Sin(V));
U = [-13.1415...3.1415]
V = [-13.1415...3.1415]



Bilinear Patch

Pnt_00 = 0.1; Pnt_01 = -0.1;
Pnt_10 = -0.1; Pnt_11 = 0.3;
F(U, V) = U*V*Pnt_00 + U*(1-V)*Pnt_01 +
(1-U)*V*Pnt_10 + (1-U)*(1-V)*Pnt_11;
U = [0.0...1.0] ~ V = [0.0...1.0]



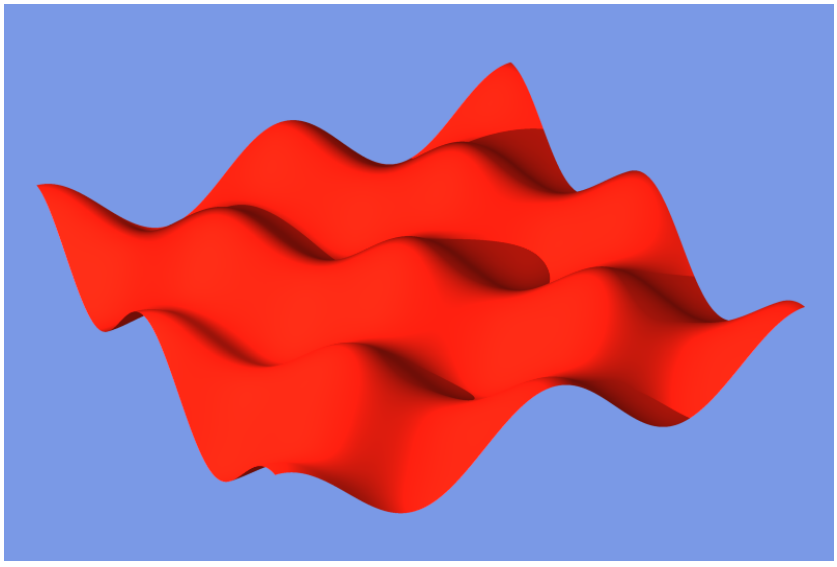


Bilinear_Height

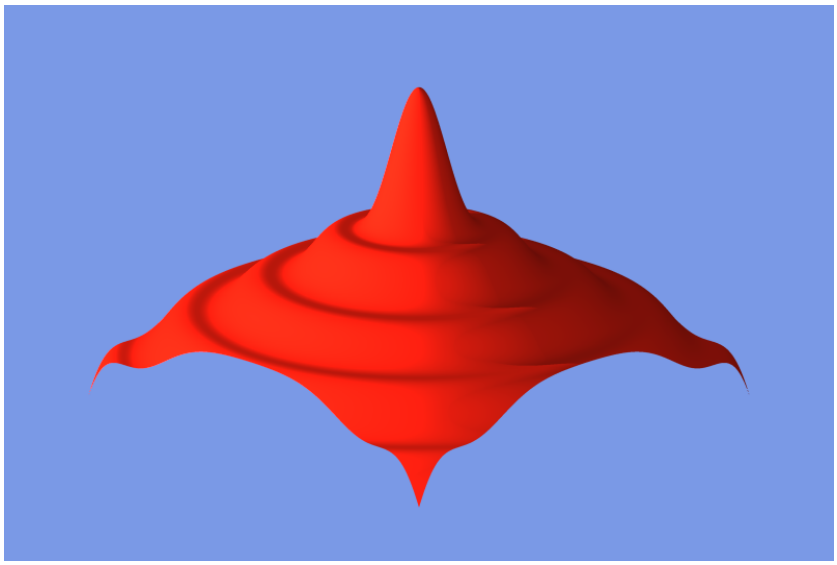
Pnt_00 = 0.3; Pnt_01 = -0.3;
Pnt_10 = -0.3; Pnt_11 = 0.9;
F(U, V) = U*V*Pnt_00 + U*(1-V)*Pnt_01 +
+ (1-U)*V*Pnt_10 + (1-U)*(1-V)*Pnt_11
U = [0.0...1.0] ~ V = [0.0...1.0]

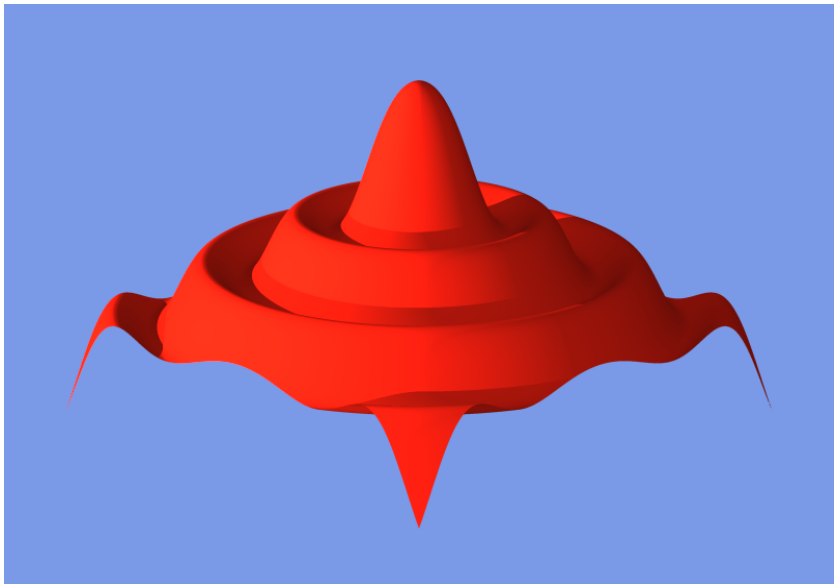
CosCos

F(U, V) = Cos(U) * Cos(V);
U = [-2π...2π] ~ V = [-2π...2π]



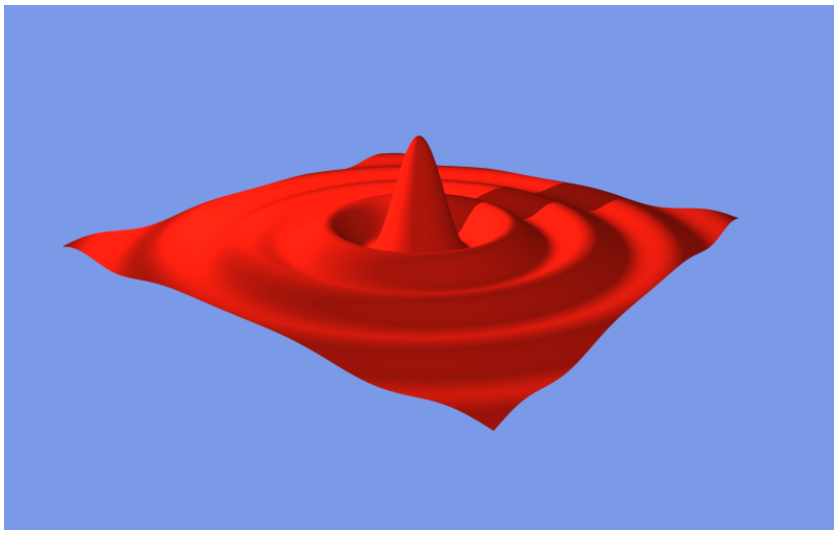
Splash #1
R = Sqrt(U*U + V*V) + π/2;
F(U, V) = Sin(R) - Sin(3R)/3 + Sin(5R)/5 - Sin(7R)/7 + Sin(9R)/9;
U = [-2.0...2.0] ~ V = [-2.0...2.0]





Splash #2
R = Sqrt(U*U + V*V) + π/2;
F(U, V) = Sin(R) - Sin(3R)/3 + Sin(5R/5) - Sin(7R)/7 - Sin(9R)/9;
U = [-2.0...2.0] ~ V = [-2.0...2.0]

Splish
R = Sqrt(U*U + V*V);
F(U, V) = 8 * Sin(R)/R;
U = [-20.0...20.0] ~ V = [-20.0...20.0]

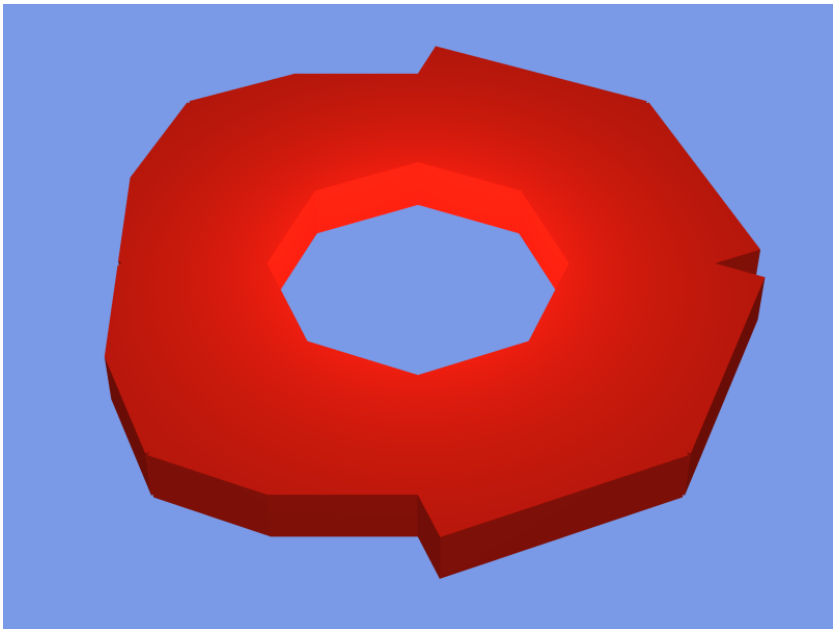


9.2 Hüllepipede

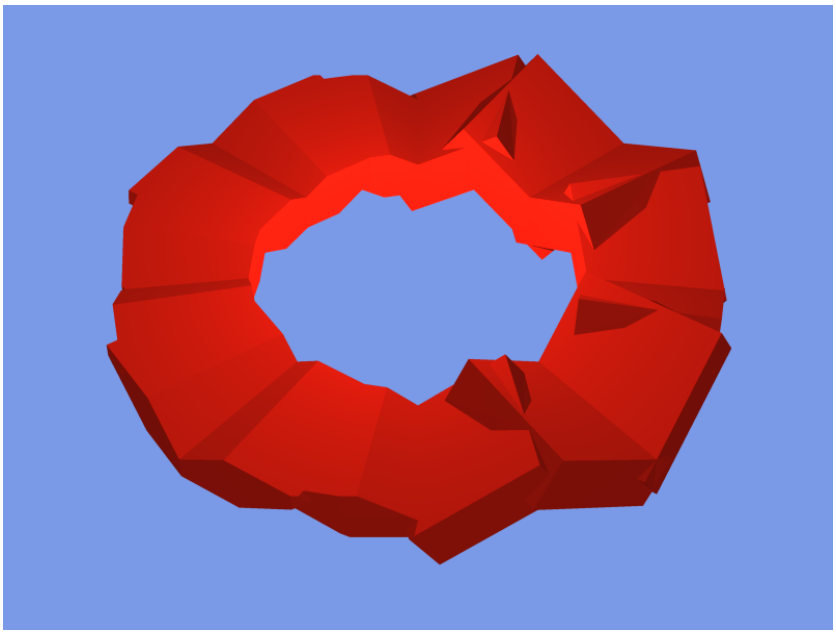
Die folgenden Serien von Bildern zeigen verschiedene Ebenen des Epipedbaums. In der rechten Bildhälfte werden dabei nur die Epipede einer Ebene dargestellt. Links ist der Durchschnitt dieser Ebene mit allen darüberliegenden Ebenen des Epipedbaums abgebildet.

Wave 8

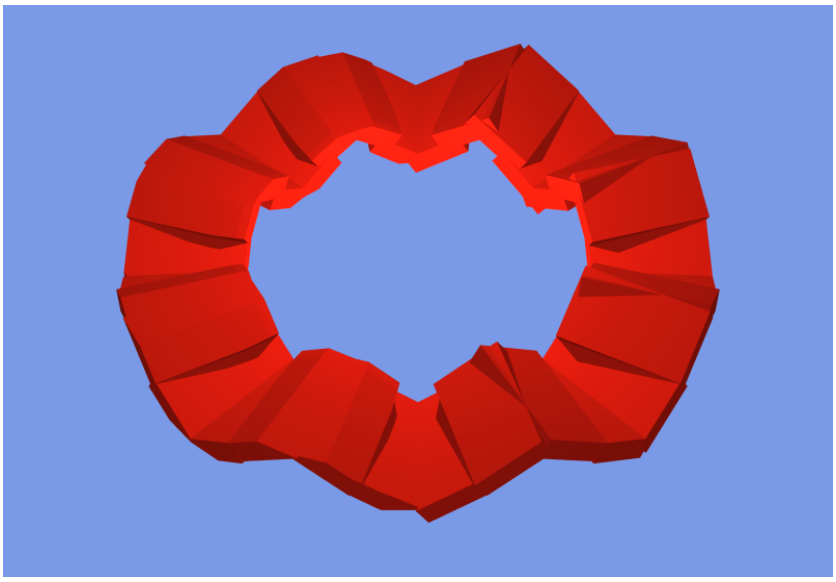
$X(U, V) = \text{Sin}(U) * V;$
$Y(U, V) = \text{Cos}(U) * V;$
$Z(U, V) = \text{Cos}(8 * U) * 0.1;$
$U = [0.0...2\pi] \sim V = [0.6...1.0]$



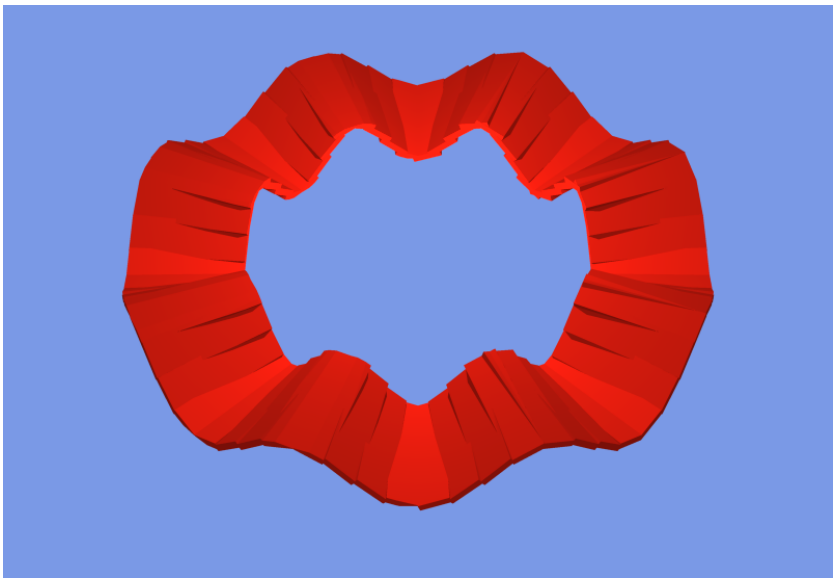
Ebene 3



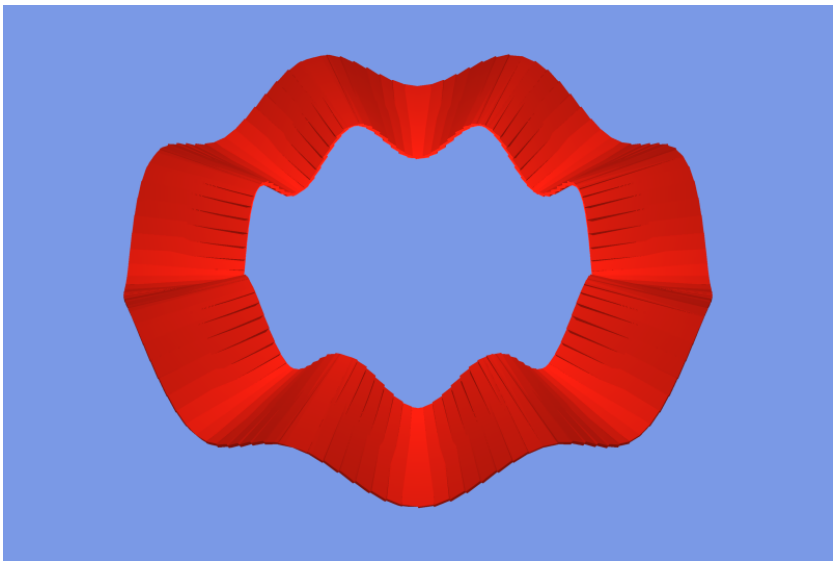
Ebene 4



Ebene 5



Ebene 6

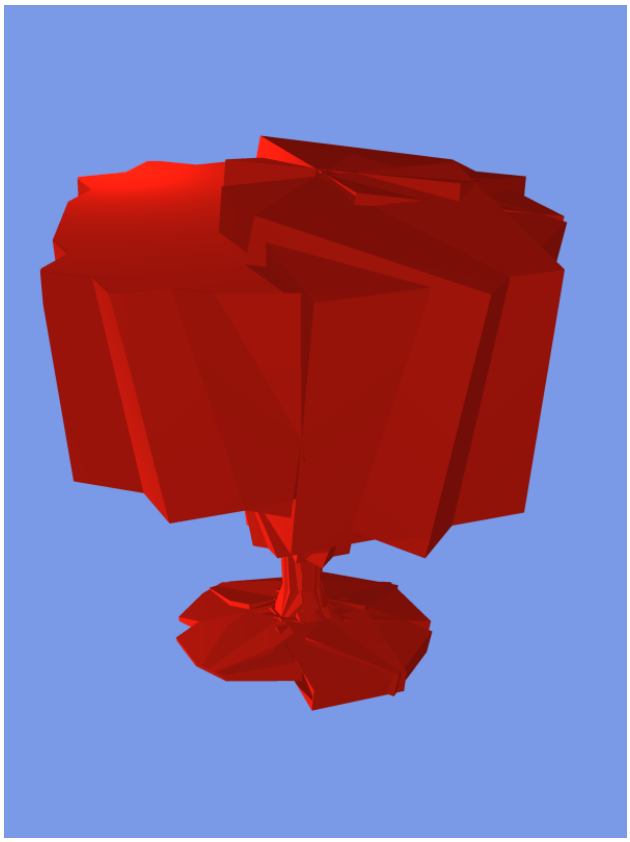


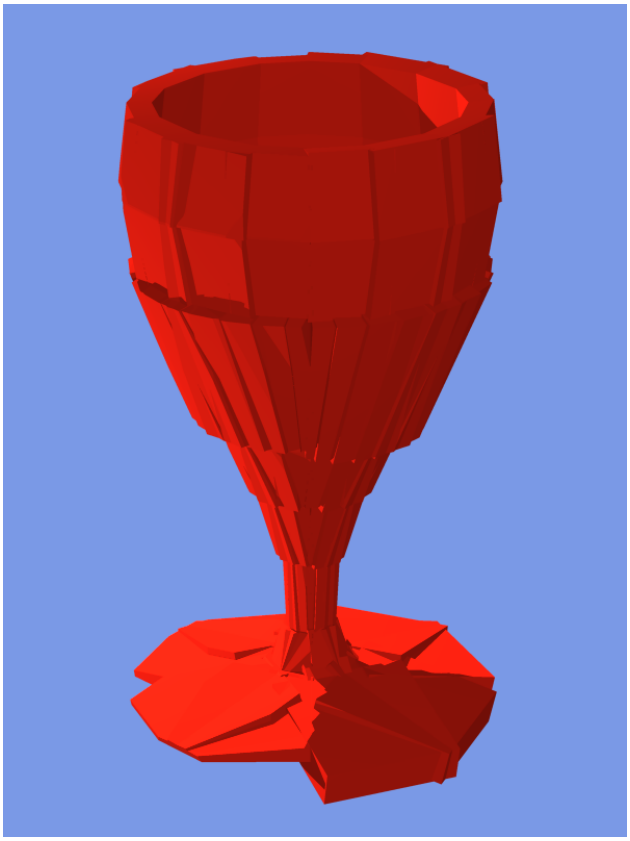
Ebene 7

Kelch

R = (((U - 6) * U + 2) * U - 1) /20 - 0.07 / (U + 0.5);
X(U, V) = R * Sin(V);
Y(U, V) = R * Cos(V);
Z(U,V) =U;
U = [-0.43...4.33] ~ V = [-π...π]

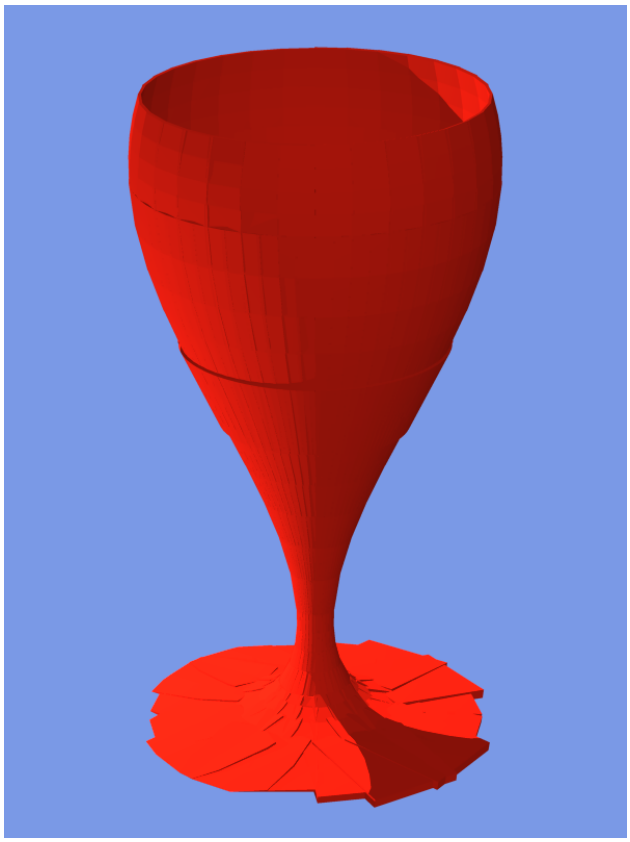
Ebene 3

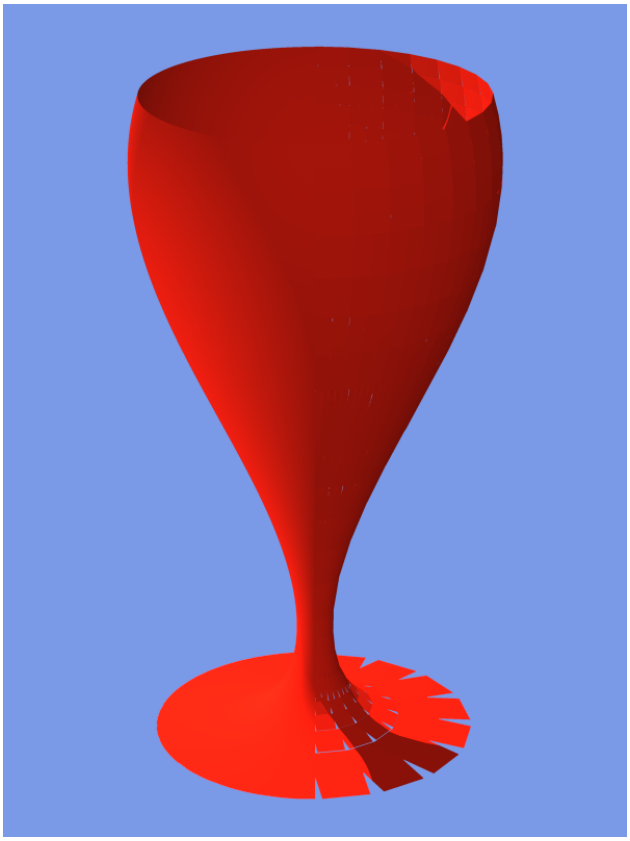




Ebene 6

Ebene 9





Links: fertiger Kelch
Rechts: Näherungsflächen

9.3 Statistik der Rechenzeiten

Titel	Blatt-Epipede	Setup Min:Sec	Eval's x1000	Newton x1000	Eval pro Newton	RunTime Std:Min	Memory KByte
Cone	64	0:01.08	5049	2181	2.31	2:03	88
TTwin	965*	0:19.53	3852	1880	2.05	2:02	349
Scene #1	704*	0:17.55	4998	2109	2.37	2:17	280
Scene #2	704*	0:16.57	5370	2333	2.29	2:52	280
Ball	719	0:14.93	5343	2520	2.12	2:32	275
Sphere_Ball	719	0:14.37	6854	3051	2.25	3:19	275
Double_Ball	602*	0:12.24	10344	4850	2.13	5:52	243
VBall	428	0:09.24	4567	2140	2.13	2:19	192
Sphere_VBall	428	0:09.10	5509	2413	2.28	2:56	193
VBall_Pat	231	0:04.56	5949	2563	2.32	2:46	137
Ei	480	0:08.44	7683	3077	2.50	2:12	206
Spiral_Egg	458	0:09.59	7449	3512	2.12	3:36	200
Egg_Ei	990*	0:17.35	15256	6505	2.35	5:17	325
Drop	440	0:07.89	8548	3434	2.49	2:26	195
Rain	1320*	0:22.47	11832	4864	2.43	3:27	431
Drip	10560*	3:08.02	23002	9400	2.45	6:34	3112
Wave 5	80	0:01.48	3380	1461	2.31	1:18	92
Wave 6	112	0:02.29	3937	1799	2.19	1:31	101
Wave 8	128	0:02.15	2418	1082	2.23	1:05	106
Nautilus #1	1024	0:20.62	15224	5271	2.89	5:36	362
Nautilus #2	4096	1:24.83	14052	5712	2.46	5:52	1239
Möbius	125	0:02.59	3525	1479	2.38	1:37	106
Möbius-Band	125	0:10.05	3523	1479	2.38	6:13	106
Kelch	1024	0:17.81	9057	3357	2.70	2:27	362
Screw	1024	0:19.70	3960	1811	2.19	1:45	362
Coil	4096	1:18.06	16457	6450	2.55	6:48	1239
Sweep #1	1024	0:24.42	47360	19526	2.43	17:56	364
Sweep #2	6976	2:52.67	38289	17280	2.22	16:38	2061
Sweep #3	6400	2:38.34	33986	15540	2.19	14:25	1897
Sweep #4	6432	2:29.01	22733	10055	2.26	9:49	1906
Sweep #5	6744	3:23.33	47495	21840	2.17	20:14	1995
Quadric 3	1800	0:31.49	7625	3157	2.41	2:24	583
Ouadric 5	1104	0:31.48	9788	4318	2.27	4:49	385
Twisted	2867	1:03.02	20460	10679	1.92	7:40	888
Ouadric 0.125	3792	1:48.07	14412	5741	2.51	5:30	1152
Ouadric 0.20	3616	1:46.13	11142	4427	2.51	4:25	1102
Ouadric 0.333	3632	1:40.00	20915	9370	2.32	9:00	1108
Crest	3437	1:03.72	4500	1836	2.45	1:52	1049
CCircle #1	1390	0:22.75	6543	2679	2.44	2:26	466
CCircle #2	1390	0:19.56	14857	6490	2.28	6:20	466
Bilinear Patch	32	0:00.44	5036	2020	2.49	1:31	83
Bilin. Height	32	0:01.41	6283	2525	2.49	2:07	102
CosCos	4096	1:15.12	20925	7585	2.76	8:01	1237
Splash #1	4096	1:14.90	13818	6174	2.24	7:41	1239
Splash #2	4072	1:52.61	15719	6692	2.35	8:57	1232

*..Alle Flächen des Bildes zusammen

Rechenzeiten auf einer VAX 3100 M38: Auflösung 1000x750, adaptives Oversampling

10. Literatur

- [AHO86]: Aho/Sethi/Ullman: Compilers Bell Laboratories 1986 deutsche Übersetzung: Compilerbau ~ Addison-Wesley 1988

- [ALE74]: Alefeld/Herzberger: Einführung in die Intervallrechnung Reihe Informatik/12 ~ Bibliograph. Institut, Zürich 1974

- [BAR90]: W.Barth: Effizientes Ray Tracing für Bezier und B-Spline-Flächen; in Encarnaçao/Hoschek/Rix: Geometrische Verfahren der Graphischen Datenverarbeitung, Springer Verlag 1990

- [GRO91]: Robert Groß: Ray Tracing für Bezier- und B-Spline-Flächen: Diplomarbeit am Inst. für Computergrafik der TU Wien, 1991

- [KIE91]: Kiendl: Ray Tracing für Bezier- und B-Spline-Flächen: Diplomarbeit am Inst. für Computergrafik der TU Wien, 1991

- [MIT90]: Mitchell: Robust Ray Intersection with Interval Arithmetic in: Graphics Interface '90, Seite 68-74

- [MIT91]: Mitchell: Three Applications of Interval Analysis in Computer Graphics: in SIGGRAPH 91