Polygonale Netze impliziter Oberflächen
(Studienarbeit)

Frank Pagels

28. März 2003

Abstract

Diese Studienarbeit beschäftigte sich mit dem Erzeugen und Darstellen von impliziten Oberflächen. Dabei wird ausführlich auf die Grundlagen impliziter Funktionen und ihre Erweiterung zu impliziten Oberflächen eingegangen. Es wird das Modellieren mit impliziten Oberflächen erläutert und Methoden zur Darstellung der Oberflächen gezeigt. Die Darstellung mittels Polygonaler Netze und dem Marching Cube Algorithmus wird ausführlich besprochen. Hierbei wird speziell Wert auf adaptive Algorithmen gelegt. In der Arbeit wird eine Implementation der vorgestellten Algorithmen anhand eines Moduls für den Modellierer Ayam von Randolf Schultz realisiert. Dabei wird auf Details der Implementation eingegangen und eine Bewertung der mit dem Modul erreichten Ergebnisse vorgenommen. Mit diesem Modul ist ein realistisches Modellieren in Echtzeit möglich.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Implizite Oberflächen zur Modellierung wurden vor 20 Jahren entwickelt. Seitdem wurden sie kontinuierlich weiter entwickelt. Insbesondere zur Beschleunigung der Berechnung und zur Entwicklung adaptiver Verfahren, die auch scharfe Kanten ermöglichen, gibt es viele Arbeiten. Der besondere Vorteil impliziter Oberflächen ist, dass mit ihnen leicht weiche und runde Körper erzeugt werden können. Auch ist es sehr einfach, Körper zu erzeugen, die durch das Verschmelzen (Blending) einzelner Grundkörper entstehen. In dieser Studienarbeit soll die Implementation eines Metaball Moduls für den Modellierer Ayam von Randolf Schultz [30] gezeigt werden. Abbildung 2 zeigt ein Beispiel für ein mit dem Metaballs Modul erzeugtes Objekt.

In dieser Arbeit werden zuerst Grundlagen zum Thema Implizite Oberflächen erklärt. Anschließend wird der Marching Cube Algorithmus zur Erzeugung von Polygonen erläutert. Hierbei wird auf Probleme und Verbesserungen, wie sie im Modul verwendet wurden, eingegangen. Zum Schluss werden mehrere adaptive Verfahren zur Polygon-Erzeugung vorgestellt, von denen eins im Rahmen der Studienarbeit implementiert wurde.

2 Grundlagen impliziter Oberflächen

Bisher bestand die Modellierung von Körpern in der Zusammensetzung von primitiven Körpern wie Linien, Kugeln, Boxen und Flächen. Hiermit war es aber schwierig, glatte, runde Oberflächen zu erzeugen. Weiterhin gab es keine einfache Möglichkeit des so genannten Blendings, des weichen Verschmelzens von Körpern. Eine Möglichkeit zum Erzeugen von gekrümmten runden Oberflächen ist z.B. die Béziertechnik. Sie wurde schon in den 60er Jahren von Bézier und de Casteljau [5, 2] vorgestellt und für den Automobilbau benutzt. Catmull und Clark stellten in [13] sowie Doo und Sabin in [14] mit Subdivision Surfaces eine weitere Möglichkeit zum Erzeugen von weichen Oberflächen vor. Hierbei wird ein Polygonales Netz rekursiv unterteilt. Abbildung 1 zeigt ein Beispiel hierfür. Das Netz wird rekursiv nach der Formel: $newmesh = F(oldmesh)$ erzeugt. Es wird hier also von einem Polygonalen Modell ausgegangen das dann iterativ verfeinert wird.

Subdivision Surfaces sind einfach zu benutzen und man kann Netze beliebiger Topologie erzeugen. Ein weiterer Vorteil ist, dass man die Kontinuität auch lokal steuern kann. Das ist z.B. auch in Abbildung 1 zu erkennen. Eine Spitze des Ausgangsnetzes bleibt hier eckig. Ein Nachteil von Subdivision Surfaces ist aber, dass man keine zwei Körper ohne größeren Aufwand verschmelzen kann. Dieses ist hingegen mit impliziten Oberflächen sehr leicht möglich. Diese wurden erstmals von Blinn [6] 1982 vorgestellt. Abbildung 2 zeigt ein Beispiel dafür. Es verschmelzen hier mehrere verformte Kugeln zu einem neuen Objekt.

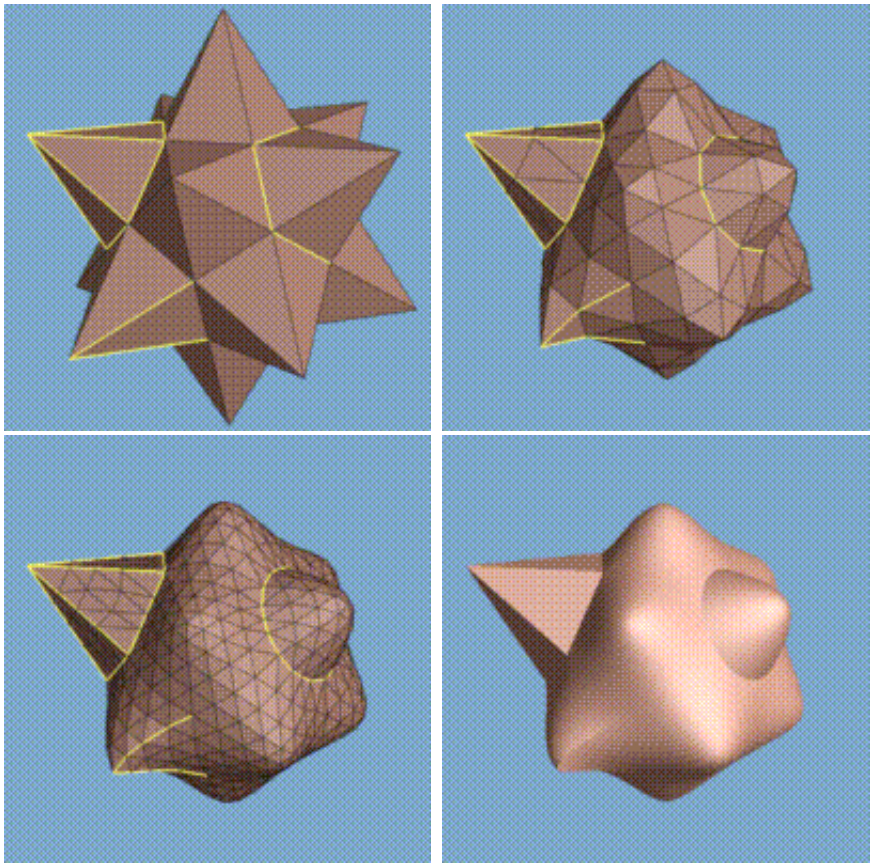

Abbildung 1: Subdivison Surfaces

Im Weiteren soll nun die Funktionsweise impliziter Oberflächen genauer erläutert werden. Hierfür wird zunächst der Begriff implizite Funktionen eingeführt.

2.1 Implizite Funktionen

Betrachten wir den Einheitskreis in Abbildung 3. Dieser Kreis kann auf mehrere Arten beschrieben werden. Als erstes trigonometrisch mit

$$p = (\cos(\alpha), \sin(\alpha)), \alpha \in [0, 2\pi]$$

oder parametrisch mit

$$p = (\pm(1 - t^2)/(1 + t^2), 2t/(1 + t^2)), t \in [-1, 1]$$

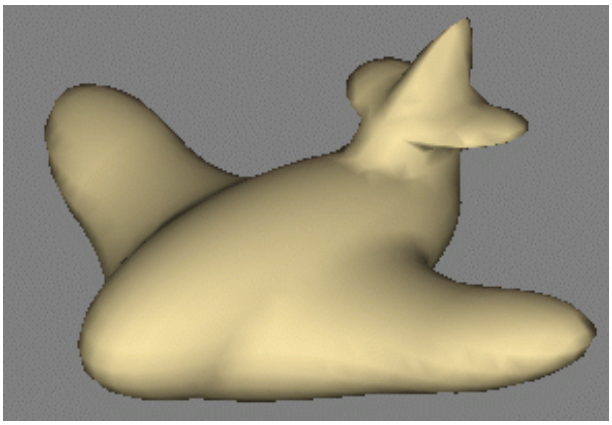

Abbildung 2: Metaball Objekt

und letztendlich kann der Kreis implizit beschrieben werden:

$$x^2 + y^2 - 1 = 0$$

Oder allgemein mit:

$$x^2 + y^2 - r = 0$$

Die letzte Gleichung beschreibt die unendliche Anzahl von Punkten (x, y), die auf dem Kreisumfang eines Kreises mit dem Radius r liegen. Der Kreisumfang ist hier also implizit beschrieben. Diese Vorgehensweise kann auf den 3D-Raum wie folgt erweitert werden:

$$x^2 + y^2 + z^2 - r = 0$$

Diese Gleichung beschreibt eine Kugel. Die Oberfläche wird durch die Nullstellen der Funktion beschrieben. Wenn ein Punkt auf der Oberfläche liegt, hat die Funktion einen Wert von Null, sonst einen Wert ungleich Null. Damit kann dann leicht bestimmt werden, ob ein Punkt innerhalb, außerhalb oder auf der Oberfläche liegt. Die Gleichung für eine Kugel kann somit als

$$F(x, y, z) - Iso = 0$$

geschrieben werden. Für verschiedene Werte von Iso und unterschiedliche Funktionen F können nun Oberflächen erzeugt werden. Hierfür müssen alle Punkte (x, y, z) gefunden werden, die mit der Funktion F den Wert Null ergeben. Solche Oberflächen heißen auch Isoflächen und die Funktion F skalare Feldfunktion. Also kann man eine implizite Funktion folgendermaßen beschreiben:

$$f : R^3 \rightarrow R$$

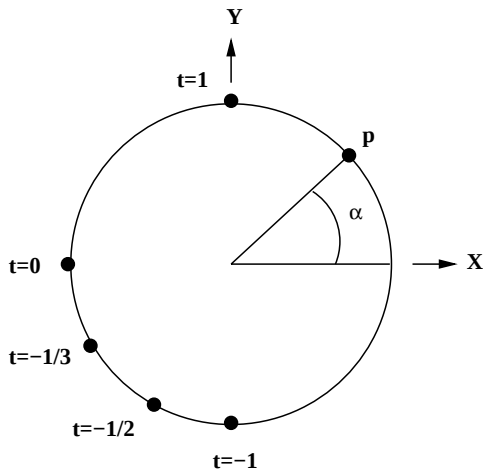

Abbildung 3: Einheitskreis

2.2 Implizite Oberflächen

Zur Darstellung von impliziten Oberflächen ist ein skalares Feld nötig, das entweder mittels einer Feldfunktion berechnet wird oder anderweitig zur Verfügung gestellt wird. Medizinische CT-Aufnahmen (CT: Computertomografie) in der Medizin können zum Beispiel ein skalares Feld beispielsweise des Gehirns enthalten. Zum Modellieren mit impliziten Oberflächen werden aber Feld- oder auch Dichtefunktionen verwendet. Dadurch ist ein Modellieren mit Deformationen möglich. Das Prinzip der impliziten Oberflächen besteht darin, dass Grundkörper nachbarschaftliche Beziehungen untereinander ausüben. Eine implizite Oberfläche besteht also im Prinzip aus Grundkörpern, die sich gegenseitig beeinflussen. Dieses wird mit der Feldfunktion erreicht. Eine Feldfunktion könnte z.B. $f(r) = 1/r^2$ sein. r ist hier der Abstand eines Punktes im Raum zu einem Kontrollpunkt in einem Universum. Je weiter man sich von dem Kontrollpunkt entfernt, desto kleiner wird der Funktionswert und er hat weniger Einfluss auf die Szene.

Das soll in Abbildung 4 mit einem Paar Kerzen verdeutlicht werden. Die Kerzen stellen punktförmige Wärmequellen dar. Die Temperatur in ihrer Umgebung kann man als Feldfunktion sehen. Je weiter man sich von der Wärmequelle entfernt desto weniger wird die Wärme spürbar. Das gleiche kann man auch bei elektrischen Punktladungen beobachten. Um jede Wärmequelle oder Punktladung breitet sich die Energie kugelförmig aus. Wenn man nun die Quellen dichter zueinander stellt, addiert sich die Energie der Quellen. Die selben durch die Addition entstandenen Temperaturen in Abbildung 4 bilden hier die Isofläche. Das soll durch den Bogen in der Mitte angedeutet sein. Der Temperaturwert legt hier fest, wie die Oberfläche aussieht. Er entspricht dem Iso-Wert impliziter Funktionen und legt die Form der Oberfläche fest. Abbildung 5 zeigt den

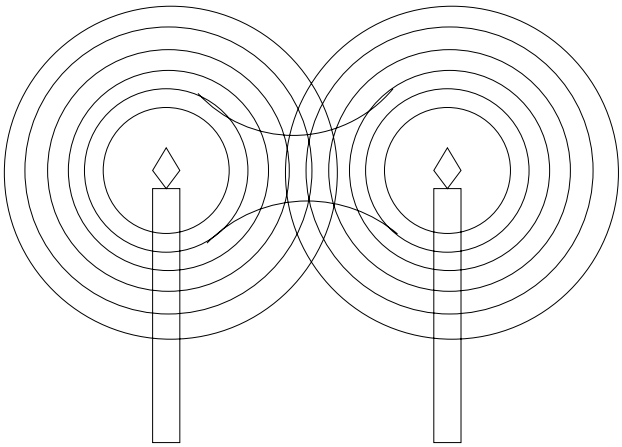

Abbildung 4: zwei Wärmequellen (nach Watt [35])

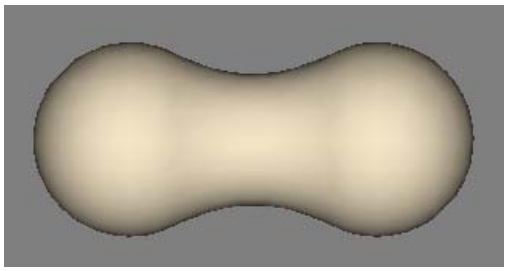

Abbildung 5: Das Dichte-Feld zweier Metaballs

Sachverhalt anhand zweier, mit Ayam gerenderter, *Metaballs*. Es wird hier also ein Dichte-Feld aus Potential-Funktionen benutzt.

$$f(S,p) = \sum_{s \in S} f_s(s,p)$$
$$p : f(S,p) - Iso = 0$$

Hierbei ist S eine Menge von Atomen und Iso ein Schwellwert (threshold). Unabhängig von der Komplexität der Szene bestimmt ein einziger skalarer Wert, der Schwellwert, die Oberfläche. Durch Verschieben der Grundkörper ist nun einfaches Modellieren von kontinuierlichen und weichen Verschmelzungen möglich. Bei Animation muss man allerdings darauf achten, dass es zu keinem unerwünschten Verschmelzen kommt. Dies kann zum Beispiel bei der Animation von Fingern vorkommen.

Zum Erzeugen von impliziten Oberflächen benötigt man nun also:

- Eine Erzeugende bzw. Grundkörper, für deren Punkte P in seiner Umgebung eine Entfernungsfunktion $d(P)$ definiert ist.
- Eine Potential-Funktion $f(d(P))$, die einen Skalarwert für die Entfernung $d(P)$ eines Punktes P von der Erzeugenden zurück gibt. Hier kann noch ein Einflussbereich gegeben sein, außerhalb dessen sie keine Auswirkung zeigt.
- Ein Skalar-Feld $F(P)$, das die kombinierten Auswirkungen der einzelnen Potenzial-Funktionen der Erzeugenden bestimmt. Hierfür ist eine Mischfunktion nötig, die im einfachsten Falle eine Addition ist. Es wird also für jede Erzeugende ein Wert für einen Punkt P berechnet und alle addiert. Um spezielle Effekte zu erreichen kann der Wert für einen Grundkörper auch subtrahiert werden. Damit können z.B. Löcher und Einbuchtungen einfach erzeugt werden.

Abbildung 6 zeigt ein Beispiel wie aus Grundkörpern eine Objekt modelliert wurde. Die blauen Kugeln haben hier ein negatives Potential.

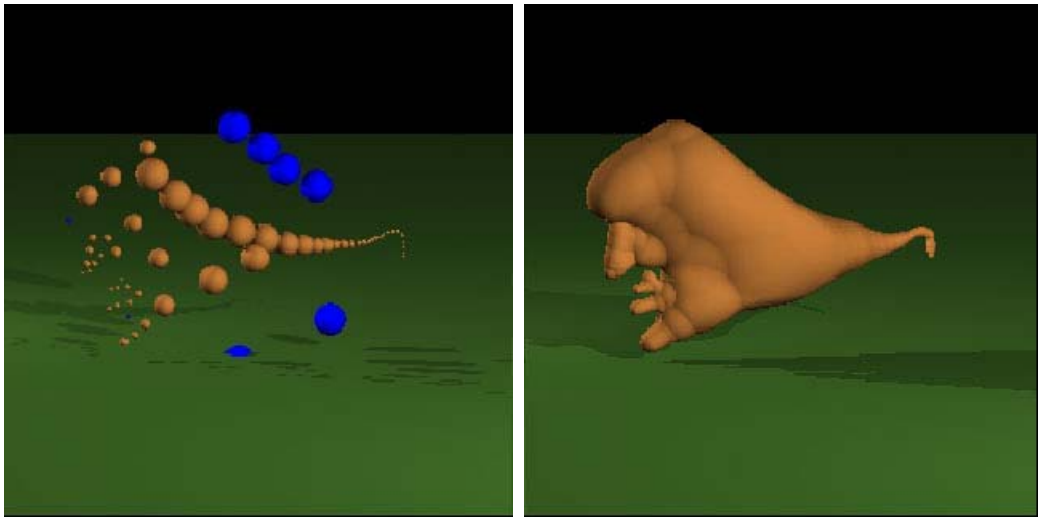

Abbildung 6: Implizite Oberfläche aus einzelnen Grundkörpern (aus [28])

2.3 Grundfunktionen

Das Aussehen der Oberflächen wird durch die implizite Funktion und den Iso-Wert bestimmt. Mit Verschiedenen Funktionen können somit unterschiedliche Oberflächen mit speziellen Eigenschaften erzeugt werden. Hier sollen einige grundlegende Funktionen vorgestellt werden.

2.3.1 Blobby Objects

Blinn verwendete 1982 als erster implizite Oberflächen in der Computer-Grafik [6]. Er benutzte dazu die Dichtefunktion von Atomen:

$$d(P) = be^{ar^2}$$

r ist der Abstand vom Punkt P zum Mittelpunkt des Atoms. Für mehrere Atome bestimmt man die Felddichte durch addieren:

$$d(P) = \sum_i b_i e^{a_i {r_i}^2} = T$$

r_i ist der Abstand von Punkt P zum i-ten Atom. Die Kurve ähnelt einer Gausskurve mit dem Mittelpunkt r_i, der Höhe a_i und der Standardabweichung b_i. Durch Verändern der Parameter a_i und b_i kann das Aussehen und die Form des Modells angepasst werden. Die Oberfläche ist definiert durch alle Punkte deren Dichte gleich dem Schwellwert T ist.

2.3.2 Metaballs

Ein Nachteil der *Blobby Objects* ist, das sich die Feldfunktion unendlich ausdehnt. Folglich muss für jeden Punkt im Raum und für jedes Primitiv in der Szene die Felddichte berechnet und addiert werden. Das kann bei vielen Primitiven sehr rechenintensiv sein. Metaballs, die von Nishimura et al. in [26] vorgestellt wurden, beschränken deswegen die Feldfunktion in der Reichweite. Ab einem bestimmten Abstand von einem Primitiv, trägt sie nicht mehr zur Gesamtfeldstärke bei. Die Dichte berechnet sich also folgendermaßen:

$$w(P) = \begin{cases} d_i \left(1 - 3\left(\frac{r_i}{b_i}\right)^2\right) & 0 \leq r_i \leq \frac{b_i}{3} \\ \frac{3d_i}{2}\left(1 - \frac{r_i}{b_i}\right)^2 & \frac{b_i}{3} \leq r_i \leq b_i \\ 0 & b_i \leq r_i \end{cases}$$

r_i ist der Abstand von Punkt P zum Zentrum des i-ten Metaball. d_i ist die Wichtung des i-ten Metabballs und b_i der Radius. Für mehrere Primitive gilt die Addition der

Feldstärken:

$$w(P) = \sum_{m_i \in M} w_i(P) \geq C$$

m_i ist der i-te Metaball, M eine Menge von Metaballs und C eine beliebige Konstante. Alle Punkte deren Dichtefunktionen einen Wert größer oder gleich C liefert, bilden die Oberfläche.

2.3.3 Soft Objects

Soft Objects wurden von Wyvill in [37, 38] vorgestellt. Hier wird die Exponentialfunktion von Blinn durch Polynome approximiert. Weiterhin wird die Felddichte ab einem bestimmten Abstand (dem Influenzradius) von Mittelpunkt zu Null abgeschnitten.

$$C(r) = \begin{cases} a\left(1 - \frac{4r^6}{9b^6} + \frac{17r^4}{9b^4} - \frac{22r^2}{9b^2}\right) & b \leq R \\ 0 & b > R \end{cases}$$

a ist ein Skalierungswert. r der Abstand von P zum Mittelpunkt und b der Radius ab dem abgeschnitten wird. Ein Vorteil ist hier, dass der Abstand quadratisch eingeht und keine Wurzel-Berechnungen notwendig sind. Wenn man mehrere Primitive nutzt, wird wieder addiert.

$$C(P) = \sum_{i=1}^{n} C(r_i)$$

2.3.4 Zusammenfassung

Mit *Blobby Objects*, *Metaballs* und *Soft Objects* wurden hier die Grundprinzipien für mögliche Dichtefunktionen gezeigt. Diese Methoden sind recht ähnlich haben aber doch Unterschiede. Die Dichtefunktion von Blinn breitet sich unendlich im Raum aus und es wird jedes Atom berücksichtigt. Das kann bei vielen Atomen sehr rechenintensiv sein. Man kann dem entgegenwirken, indem man eine Boundingsphere um jedes Atom legt. Bei den *Metaballs* und *Soft Objects* wurde dieses Problem durch die Wahl der Feldfunktion gelöst. Sie setzen den Feldwert ab einem bestimmten Abstand auf Null. Für *Metaballs* und *Soft Objects* werden Polynome zur Berechnung benutzt. Dies ist in der Berechnung billiger als die Exponentialfunktion von Blinn. Die *Soft Objects* haben weiterhin den Vorteil, daß die Distanz quadratisch in die Formel eingeht und keine rechenintensive Wurzel-Berechnung nötig ist. Alle drei Techniken wurden zusätzlich zu Kugeln, um komplexere Grundkörper wie z.B. Tori oder Würfel erweitert. Eine Dichtefunktion für einen Torus sieht folgendermaßen aus:

$$f(P) = (x^2 + y^2 + z^2 + R^2 - r^2)^2 - 4R^2(x^2 + z^2)$$

Hier ist R der äußere und r der innere Radius des Torus. Man sieht, daß Gleichungen für andere Objekte als Kugeln recht komplex werden können. Ein Würfel wird z.B. mit:

$$f(P) = x^n + y^n + z^n$$

erzeugt. n bestimmt hier, wie scharf die Kanten werden sollen. Je höher n ist, desto mehr nähert sich das Objekt einem exakten Würfel an. Ein guter Startwert ist hier $n = 8$.

Weiterhin sind skelettähnliche Funktionen, die auf Linien und Kurven basieren, möglich [9]. Damit können z.B. Menschen- und Tierkörper modelliert werden. Hier muss darauf geachtet werden, daß es z.B. bei der Modellierung von Fingern zu unerwünschten *blending*-Effekten kommen kann. Ein weiteres Problem entsteht beim Verbinden von Linien-Objekten. Wenn sich zwei Enden treffen, kann es zu unerwünschten Ausbuchtungen (*bulge*) kommen. In [8] stellt Bloomenthal eine Lösung für dieses Problem vor.

3 Darstellung impliziter Oberflächen

Implizite Oberflächen haben den Nachteil, dass es nicht leicht ist, diese in Echtzeit darzustellen. Es haben sich zwei Methoden zur Darstellung von impliziten Oberflächen etabliert. Diese sind zum einen das direkte Raytracing und zum anderen die Polygonisation, also die Verwendung Polygonaler Netze. Welches Verfahren benutzt wird, hängt auch davon ab, ob hohe Geschwindigkeit oder hohe Qualität Vorrang haben. Mit Raytracing wird die höchste Qualität erzielt, da hier die Oberfläche korrekt dargestellt wird. Bei einer Polygondarstellung kann die Oberfläche nur angenähert werden. Diese Methode ist aber weitaus schneller als Raytracing.

3.1 Raytracing

Mit Raytracing erreicht man die beste Qualität zur Darstellung von impliziten Oberflächen. Raytracing kann benutzt werden, wenn die Bestimmung eines Schnittpunkts und des Normalenvektors möglich ist. Für eine erste Annäherung des Schnittpunktes mit der impliziten Oberfläche kann ein Schnittpunkt mit einer umgebenden Kugel, *bounding sphere*, berechnet werden. Danach sind weitere Berechnungen nötig, um den exakten Schnittpunkt zu finden. Der Normalenvektor wird mit

$$N = \left(\frac{\partial F}{\partial x}, \frac{\partial F}{\partial y}, \frac{\partial F}{\partial z} \right)$$

bestimmt. Raytracing ist die aufwendigste und langsamste Methode zum Darstellen von impliziten Oberflächen. Blinn [6] und Nishimura [26] verwendeten Raytracing zum Visualisieren ihrer Modelle. John Hart beschreibt z.B. in [17] das Raytracing von impliziten Oberflächen.

3.2 Polygonale Netze

Eine Visualisierung von impliziten Oberflächen mittels Polygonaler Netze erfordert weniger Rechenaufwand und ist damit schneller als Raytracing. Polygone können außerdem von Grafikkarten besser verarbeitet werden. Für das Erzeugen von Polygonen sind zwei grundlegende Operationen nötig: Abtastung (*sampling*) und Zusammenfügen (*structuring*). Mit dem Abtasten werden Punkte auf der Oberfläche erzeugt. Diese werden dann im zweiten Schritt so miteinander verknüpft, dass sie ein Polygon-Netz ergeben. Die erste Operation beschäftigt sich mit der Geometrie, das Zusammenfügen ist jedoch eine topografische Operation. Die Algorithmen zum Erzeugen Polygonaler Netze lassen sich danach einteilen, wie diese Operationen implementiert sind. Die einfachste Methode ist eine einheitliche Unterteilung des Raumes. Um die Polygone zu erhalten, wird der 3D Raum in gleich große Zellen unterteilt. Danach wird festgestellt, welche Zellen die Oberfläche schneiden. Aus den Schnittpunkten mit den Zellen können dann Polygone erzeugt werden. Der so genannte Marching Cube Algorithmus [23] erzeugt nach diesem Verfahren Polygone. Es wurden aber weitere Algorithmen entwickelt. In [36] beschreiben Witkin und Heckbert die Verwendung von Partikeln zum Darstellen impliziter Oberflächen. Bei der Darstellung mittels Polygonen wird die Oberfläche nur angenähert. Für eine bessere Genauigkeit kann z.B. die Größe der Zellen verkleinert werden. Abbildung 7 zeigt die Auswirkung auf die Auflösung. Es existieren weiterhin adaptive Algorithmen die in Abschnitt 4 behandelt werden.

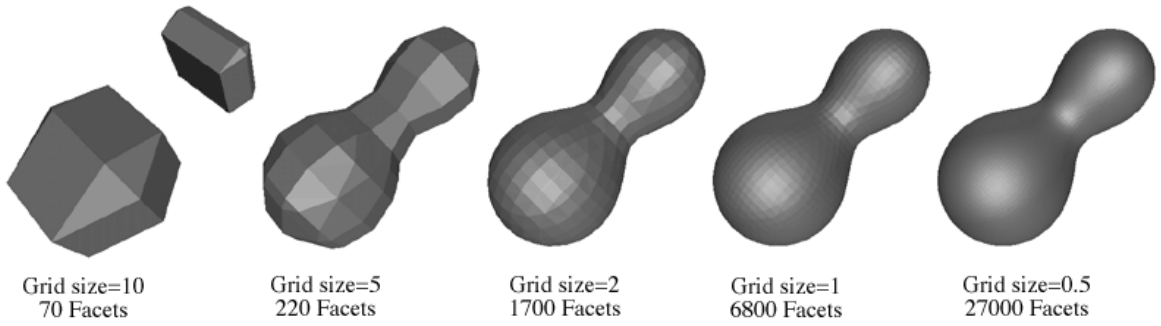

Abbildung 7: Verschiedene Gitter-Auflösungen (aus Bourke [12])

3.2.1 Marching Cube Algorithmus

Der Marching Cube Algorithmus wurde 1987 von Lorensen und Cline [23] vorgestellt. Er dürfte das bekannteste und meistverwendetste Verfahren zur Volumen-Visualisierung sein. Es wird auch z.B. zur dreidimensionalen Darstellung von CT-Schicht Bildern benutzt.

Zunächst wird der Raum (das Universum), in dem sich das Objekt befindet, in gleich große Voxel unterteilt (vgl. Abbildung 8). Nun werden die Voxel darauf hin untersucht, ob sie sich mit der Oberfläche schneiden. Dazu wird der skalare Wert für jede Ecke bestimmt. Dann wird verglichen, ob er größer oder kleiner als der Iso-Wert ist. Damit kann also festgestellt werden, ob sich eine Ecke inner- oder außerhalb der Oberfläche befindet. Siehe auch Seite 7 zur weiteren Erläuterung.

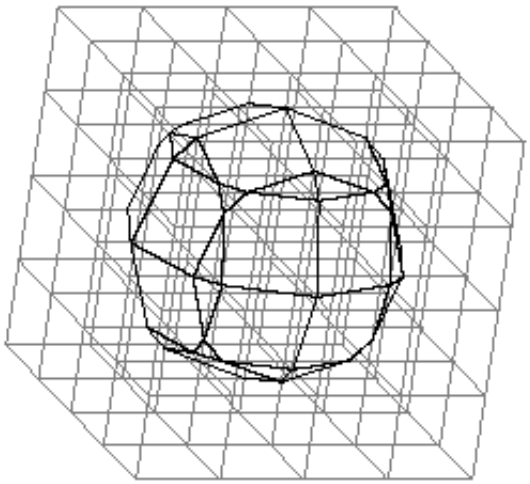

Abbildung 8: Unterteilter Raum (aus Bloomenthal [7])

Die Voxel werden nun anhand der Iso-Werte an den Ecken klassifiziert. Da ein Würfel 8 Ecken hat, gibt es theoretisch 256 Möglichkeiten zur Polygonbildung. Sie lassen sich aber durch Komplementbildung, Spiegelung und Rotation auf 15 Äquivalenzklassen reduzieren. Abbildung 9 verdeutlicht das. Es gibt jetzt zwei Spezialfälle, alle Ecken sind inner- oder außerhalb der Oberfläche. In diesem Fall entstehen keine Polygone.

Zur Bestimmung welche Kanten nun an der Polygonbildung beteiligt sind, wird im allgemeinen eine Tabelle verwendet. Dazu wird als erstes ein 8 Bit Ecken-Code erzeugt. Jedes Bit zeigt an, ob eine Ecke unterhalb des Iso-Wertes liegt. Wenn eine Ecke einer Kante unterhalb und eine andere oberhalb des Iso-Wertes liegt, schneidet die Kante die Oberfläche. Die genaue Position des Schnittpunktes mit der Kante, kann mittels linearer Interpolation gefunden werden. Für eine schnelle aber ungenaue Berechnung kann man

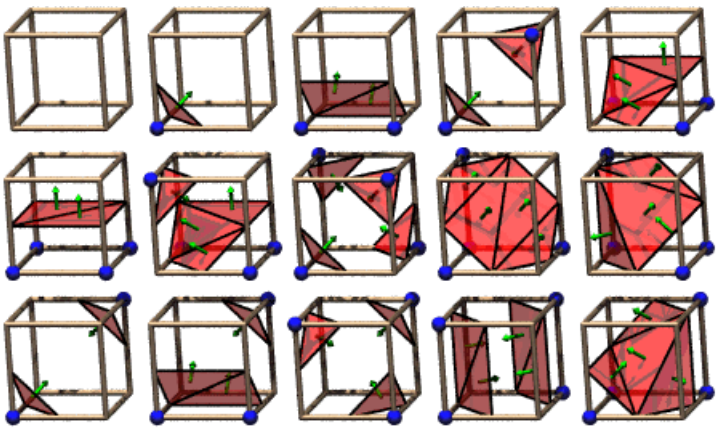

Abbildung 9: 15 Verschiedene Möglichkeiten zur Polygonbildung

auch den Mittelpunkt der Kante verwenden. Mit der Tabelle kann nun bestimmt werden welche gefundenen Punkte ein Polygon bilden. Pro Voxel können bis zu fünf Polygone entstehen. Abbildung 10 zeigt, wie der Ecken-Code gebildet wird. In diesem Beispiel liegt Ecke 3 unterhalb des Iso-Wertes und ergibt somit 0000 1000 als Code. Aus der Tabelle lässt sich folglich 1000 0000 1100 als Kanten-Code entnehmen. Dies bedeutet Kanten 2, 3 und 11 werden geschnitten.

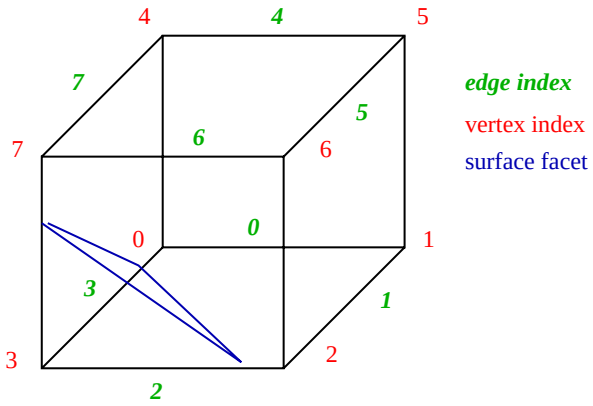

Abbildung 10: Schnittstelle mit Voxel (nach Bourke [12])

3.2.2 Bestimmung der Normalen

Für einen beliebigen Punkt auf der Oberfläche ist der normalisierte Gradient die Normale, welche für Schattierung und adaptive Verfahren nötig ist. Wir benötigen also die erste partielle Ableitung der Dichtefunktion. Das Tripel mit den Ableitungen bildet den Gradienten. Dieser Gradient wird normiert und ergibt die Oberflächennormale an dem betrachteten Punkt.

$$\nabla f = \left(\frac{\partial F}{\partial x}, \frac{\partial F}{\partial y}, \frac{\partial F}{\partial z} \right)$$

Zum Beispiel ist der Gradient einer Einheits-Kugel $f(x, y, z) = x^2 + y^2 + z^2 - 1$, $2x + 2y + 2z$. Der normalen Vektor an dem Punkt $(1, 0, 0)$ auf der Kugel hat demnach den Wert $(1, 0, 0)$. Dieser Vektor zeigt dann auch erwartungsgemäß nach außen.

Ist eine Funktion nicht oder nur sehr umständlich differenzierbar, kann der Gradient numerisch mit Vorwärts-Differenzen approximiert werden.

$$\nabla f \approx \begin{cases} (f(P) - f(P + \Delta x))/\Delta \\ (f(P) - f(P + \Delta y))/\Delta \\ (f(P) - f(P + \Delta z))/\Delta \end{cases}$$

$\Delta x, \Delta y$ und Δz sind Verschiebungen um Δ entlang der jeweiligen Achse. Für kleine Δ ist der Fehler proportional zu Δ. Wird ∇f mittels zentraler Differenzen berechnet:

$$\nabla f \approx \begin{cases} (f(P + \Delta x) - f(P - \Delta x))/2\Delta \\ (f(P + \Delta y) - f(P - \Delta y))/2\Delta \\ (f(P + \Delta z) - f(P - \Delta z))/2\Delta \end{cases}$$

ist der Fehler proportional zu Δ^2. Δ ist im Allgemeinen recht klein. Ein guter Wert ist ein hundertstel der Seitenlänge eines Voxels. Zu kleine Werte für Δ können allerdings zu numerischen Fehlern führen.

Ist der Gradient ungleich Null im Punkt P, wird P regulär genannt und $\nabla f(P)$ die Normale der Oberfläche im Punkt P. Ist der Gradient unbestimmt, ist der Punkt P Singular oder auch kritischer Punkt genannt. Die Normale eines singulären Punktes kann eventuell als Durchschnitt der Normalen der umgebenden Punkte bestimmt werden.

3.2.3 Surfacetracker

Die einfachste Umsetzung des Marching Cube Algorithmus besteht darin, in einem 3D Array von Cube Strukturen von jedem Würfel die Iso-Werte für jede Ecke zu bestimmen

und anschließend jeden Würfel zu untersuchen, ob er die Oberfläche schneidet.

Wenn man sich eine durchschnittliche Szene betrachtet, erkennt man, daß in dem Universum nur relativ wenige Cubes die Oberfläche schneiden. Zum Beispiel braucht das komplette Innere eines Objektes nicht betrachtet werden. Es werden also bei der ersten Methode zu viele Würfel betrachtet, die nichts zur Oberfläche beitragen. Ziel ist es also, nur die wirklich beteiligten Würfel zu berechnen.

Um das zu erreichen wird von einem Start-Cube aus untersucht, ob er die Oberfläche schneidet und wenn ja werden die Nachtbar-Cubes rekursiv untersucht. Hierbei wird ein Cube-Stack verwendet, in dem die aktuell gefundenen Nachbar-Cubes gespeichert werden. Aus dem Ecken-Code, siehe Seite 16, kann leicht gefunden werden, an welchen Kanten des Würfels die Oberfläche weiter verläuft. Diese angrenzenden Würfel werden dann auf den Stack gelegt. Der Prozess ist beendet, wenn der Stack leer ist.

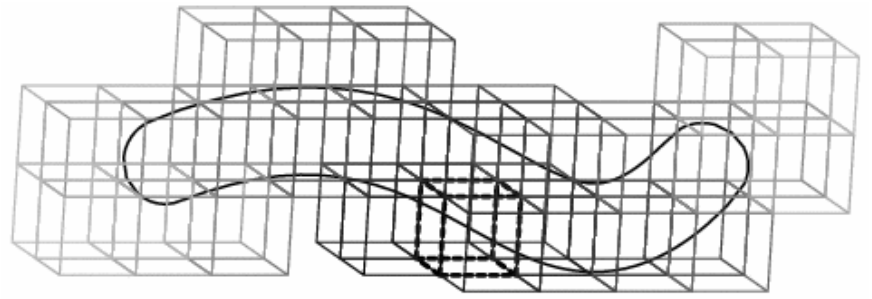

Abbildung 11: Prinzip des Surfacetrackers (aus Bloomenthal [7])

Ein Ausschnitt aus dem Pseudocode soll die Funktionsweise des Surfacestracker verdeutlichen.

```
RenderMetaballs()
{
  for each ball
  {
    pos = ball's position in the grid
    found = false
    while not found
    {
      pos = move one step up
      case = ComputeVoxel(pos)
```

```
      if case < 255
        found = true
    }

    if voxel is computed
      continue with next ball

    AddNeighborsToOpenList(case,pos)
    while open list is not empty
    {
      pos = position of last voxel in open list
      case = ComputeVoxel(pos)
      AddNeighboursToOpenList(case,pos)
    }
  }
}
```

Alle Objekte der Szene hängen in einer Liste und werden nacheinander abgearbeitet. Das erste Objekt wird so lange nach oben bewegt bis ComputeVoxel() feststellt, dass die Oberfläche des Objektes erreicht wurde. Case ist hier der Ecken-Code der auf Seite 7 erläutert wurde. Je nach Form des Objektes muss auch nach unten oder seitlich nach einem Start-Cube gesucht werden. Wurde der gefundene Cube schon betrachtet, kann das Objekt übergangen werden, da es schon von einem anderen Objekt ein Nachbar war und deshalb schon betrachtet wurde. Wurde der Cube noch nicht betrachtet und schneidet er die Oberfläche, werden in AddNeighborsToList() die Nachbar-Cubes bestimmt und in einem Stack gespeichert. Dieser wird dann abgearbeitet wobei dann neue Nachbar-Cubes zum Stack hinzukommen. Ist der Stack leer, wird das nächste Objekt in der Liste untersucht. Wenn die Oberfläche, die aus den einzelnen Objekten erzeugt wurde, zusammenhängend war, wurden die neuen gefundenen Start-Cubes schon betrachtet und der Algorithmus endet.

Wenn man ein skalares Feld z.B. von einer CT vorgegeben hat und eine Isofläche zur Visualisierung erzeugen möchte, macht es eventuell keinen Sinn den Surfacetracker zu verwenden. Es müssen ja in diesem Fall keine Iso-Werte mehr berechnet werden. Außerdem kann es Probleme geben, wenn die Oberfläche nicht zusammenhängend ist. Die Würfel können und müssen in diesem Fall alle betrachtet werden.

3.2.4 Probleme des Marching Cube Algorithmus

Mit dem Marching Cube Algorithmus entstehen bei bestimmten Situationen Probleme wie z.B. das Auftreten von Mehrdeutigkeiten wenn ein Satz von zwei Punkten, sich diagonal gegenüber stehen. Diese Mehrdeutigkeiten zeigt Abbildung 12.

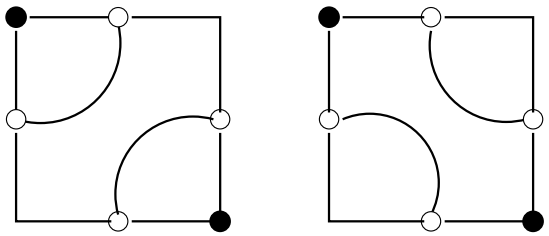

Abbildung 12: mögliche Mehrdeutigkeiten beim Marching Cube Algorithmus

Es kann nicht genau gesagt werden, welche Kombination hier zu verwenden ist. Eine mögliche Lösung hierfür ist es, den Würfel in Tetraeder aufzuspalten. Hier gibt es dann nur noch $2^4 = 16$ Möglichkeiten zur Polygonbildung. Diese lassen sich weiter auf nur 8 Äquivalenzklassen reduzieren.

Beim originalen Marching Cube Algorithmus besteht weiterhin das Problem, daß man kein zusammenhängendes Polygon Netz erhält. Es werden nur einzelne Polygone erzeugt, ohne daß man eine Information über die Position der Polygone zueinander erhält. Dem könnte man durch Speichern der Polygon-Daten in einem Octree entgegenwirken. Für eine Darstellung mit OpenGL wären außerdem so genannte *Triangle-Strips* günstiger, da diese von Grafikkarten schneller verarbeitet werden können.

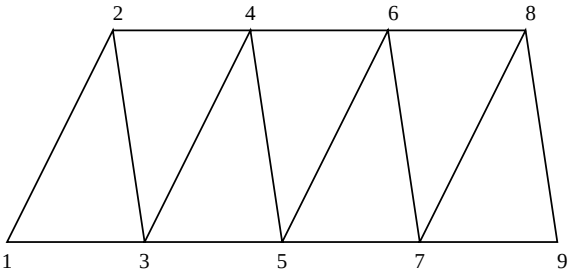

Abbildung 13: Triangle-Strip

Abbildung 13 zeigt so ein *Triangle-Strip*. Die Darstellung ist mit OpenGL wesentlich schneller. Weiterhin werden weniger Punkte benötigt und die Struktur wird kompakter. Luiz Velho stellt in [34] eine Methode zum Erzeugen von *Triangle-Strips* von parametrischen und impliziten Oberflächen vor. Hierbei wird gleich beim Erzeugen der Polygone versucht, *Triangle-Strips* zu bilden. Aus einem Polygon-Modell optimale *Triangle-Strips* zu bilden ist NP-Vollständig [3]. Akeley et al. haben in [21] einen Algorithmus als Quellcode veröffentlicht, der dieses Problem löst. Es gibt weitere Veröffentlichungen,

die mit Heuristiken an der Optimierung der Geschwindigkeit zur Generierung der *Strips* arbeiteten. Einen guten Überblick geben z.B. [19, 15].

4 Adaptive Verfahren

Die mit dem Marching Cube Algorithmus erzeugten Polygon-Netze sind nicht immer optimal. Ein Schwachpunkt ist die einheitliche Größe des den Raum unterteilenden Gitters. Hierdurch kommt es zu einer Überabtastung in Bereichen mit geringer Krümmung und zu einer Unterabtastung bei einer hohen Krümmung der Oberfläche. Es wurden mehrere Verfahren entwickelt, um dies zu optimieren. Oft werden viele Polygone erzeugt, die eigentlich unnötig sind. Eine Reduzierung der Anzahl der Polygone um bis zu 80% durch ein Octree basiertes Verfahren wird in [31] beschrieben. Polygone werden nach bestimmten Kriterien zusammengefasst, um die Reduzierung zu erreichen. Hierbei wird ein ähnlicher Surfacetracker-Algorithmus, wie auf Seite 17 beschrieben, verwendet.

Andere Methoden führen eine Glättung (*smoothing*) des Polygon-Netzes durch verschiedene Filterungen wie Median- oder Gauss-Filter durch. Ein Filterung durch ein Median-Filter wird in [18] vorgestellt. Einen Vergleich verschiedener Methoden zum Glätten Polygonaler Netze stellen Belyaev et al. in [4] vor.

Der Schwachpunkt des originalen Marching Cube Algorithmus ist, daß man damit nicht ohne weiteres scharfe Kanten erzeugen kann. Abbildung 14 verdeutlicht das. Hier wurde aus einem Kugelobjekt mittels eines negativen Objektes ein Teil ausgeschnitten. Im linken Bild sieht man eindeutig Artefakte. Diese kann man verkleinern, indem man die Auflösung des Gitters, siehe auch Abbildung 7, vergrößert. Dabei werden aber unnötig viele Polygone an glatten Stellen erzeugt. Es sind also adaptive Verfahren nötig, um nur an kritischen Stellen die Gitter-Auflösung zu erhöhen. Das rechte Bild zeigt das Ergebnis eines adaptiven Verfahrens, wie es für Ayam implementiert wurde. Man erkennt hier eine deutliche Verbesserung in der Darstellung. Im linken Bild kann man noch an den Rändern der Einbuchtung die Gitterstrukturen erkennen. Durch Erhöhen der Gitterauflösung könnte man eine gewisse Verbesserung erreichen, dies würde dann aber die Anzahl der Polygone in allen Bereichen des Objektes erhöhen. Auch da, wo eine höhere Auflösung gar nicht nötig wäre. Für das mit dem adaptive Verfahren erzeugte Bild, wurde die Gitterauflösung nicht verändert. Es wurde die gleichen Auflösung wie für das linke Bild verwendet.

Adaptive Algorithmen sind komplexer als einfache, mit einem festen Gitter arbeitende Algorithmen, denn sie müssen zwei voneinander abhängige Probleme lösen:

- für eine optimalen Abtastung sorgen

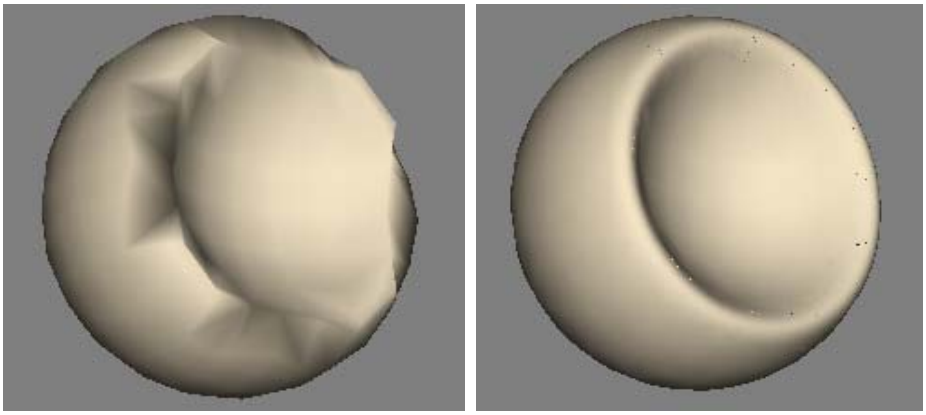

Abbildung 14: links ohne und rechts mit einem adaptiven Verfahren erzeugtes Objekt

- dabei eine korrekte Topologie erreichen

Eine optimale Abtastung sorgt für eine gute geometrische Approximation und hängt von den Adaptions-Kriterien ab. Das Achten auf eine korrekte Topologie sichert die Konsistenz des Polygonalen Netzes und hängt von der Qualität des Algorithmus zum Verbinden der erzeugten Punkte ab.
Die Schwierigkeit besteht nun darin, dass wenn die Abtastrate *lokal* geändert wird, die Netz-Topologie *global* beeinflusst werden kann. Es ist also ein Mechanismus nötig, der dieses verhindert. Anderenfalls können durch falsche Verbindungen der Punkte Löcher im Polygon-Netz entstehen. Unterschiedliche Abtastraten bei zwei benachbarten Voxeln im Marching Cube Gitter können zu diesen Problemen führen.

Im Folgenden sollen nun zwei unterschiedliche Verfahren zum Erreichen der Adaptivität vorgestellt werden. Sie unterscheiden sich darin, dass im einen die Voxel des Marching Cubes Gitter rekursiv unterteilt werden und im anderen die erzeugten Polygone unterteilt werden.

4.1 Adaptive Unterteilung des Raumes

Wie mit Abbildung 7 auf Seite 14 gezeigt wurde, kann durch Erhöhung der Auflösung des Marching Cube Gitters eine bessere Genauigkeit bei der Repräsentation der impliziten Oberfläche erreicht werden. Hierbei kann sich die Anzahl der Polygone und Eckpunkte massiv erhöhen. Es liegt also nahe, nur an den Stellen an denen es z.B. starke Krümmungen gibt, die Gitterauflösung zu verbessern. Dieses kann erreicht werden, indem ein Marching Cube Voxel geprüft wird, ob Probleme auftreten können. Wenn das der Fall ist, wird der Voxel in acht kleinere unterteilt und die entstehenden Voxel werden

dann wieder geprüft ob die Genauigkeit ausreichend ist. Ist das nicht der Fall, müssen die Voxel weiter unterteilt werden. Die Szene wird dann in einem Octree verwaltet. Abbildung 15 zeigt, wie die Unterteilung erfolgt. Die adaptive Unterteilung erfolgt hier mit dem Aufbau des Octrees.

Für eine Unterteilung eines Voxels gelten folgende Kriterien:

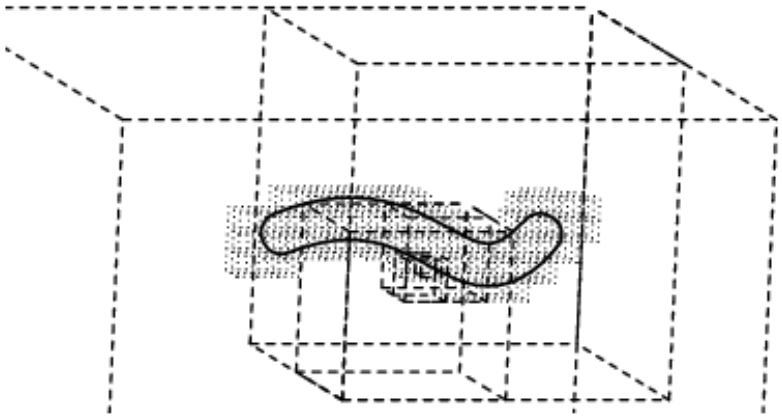

Abbildung 15: Beispiel eines Octrees für eine Szene (aus Bloomenthal [7])

- schneidet eine Ecke die Oberfläche
- ist eine maximale Unterteilung oder eine minimale Voxel-Größe erreicht
- entsteht mehr als ein Polygon aus dem Voxel
- ist das Polygon eben
- weicht die Normale der Eckpunkte von der Normale des Polygon-Zentrums ab?

Mit den Polygonecken P_i, ihren Einheitsnormalen N_i und der Einheitsnormale N im Polygonzentrum lässt sich bestimmen, wie plan das Polygon ist:

$$Max(V_i, N), i \in [1, \text{nPoints}] \text{ und } V_i \text{ der Einheitsvektor der Punkte } P_i \text{ und } P_{i+1}$$

Die Abweichung der Normalen wird mit

$$Max(N_i, N), i \in [1, \text{nPoints}]$$

bestimmt.

Der Algorithmus wird von Bloomenthal in [7] anhand von Pseudocode genauer erläutert. Es wird hier eine Art Surfacetracker (siehe Seite 18) mit einem Cube-Stack verwendet. Es werden hier also auch nur die beteiligten Voxel betrachtet.

4.2 Adaptive Unterteilung der Polygone

Im vorher beschriebenen Algorithmus erfolgt die Abtastung und adaptive Unterteilung in einem Schritt. Außerdem wird ein Octree zur Verwaltung der adaptiven Rekursion benötigt. Einen anderen Weg beschreibt Luiz Velho [33]. Sein Algorithmus wurde für Ayam implementiert und soll nun genauer beschrieben werden. Der Algorithmus verwendet zwei Schritte, um Adaptivität zu erreichen. Er wird dadurch einfacher und braucht weniger Speicherplatz. Diese Schritte sind:

- erzeuge Start-Polygone
- adaptive Verfeinerung

Die Start-Polygone werden mit einem normalen Marching Cube Verfahren erzeugt. Das entstehende Netz ist dann der Ausgangspunkt für eine adaptive Verfeinerung. Die Polygone des Netzes werden dann in Abhängigkeit von der lokalen Krümmung der Oberfläche rekursiv unterteilt. Für jedes Polygon des Startnetzes wird die Krümmung über die Abweichung der Polygonnormalen zur Oberflächennormalen bestimmt.

Für eine einfache Adaption ist eine Trennung von Polygonbildung und Abtastung nötig. Mit den Start-Polygonen ist der erste Schritt beendet. Hierbei ist die Abtastung eventuell noch etwas grob. Im zweiten Schritt werden neue Punkte erzeugt und auf die Oberfläche projiziert. Dabei wird der Gradient der impliziten Funktion verwendet. Das Start-Netz hilft dann bei der Erhaltung der geometrischen Konsistenz, da hier die Positionen der Polygone vorgegeben sind. Wenn die lokale Krümmung einen vorgegebenen Schwellwert überschreitet, wird ein Polygondreieck an den Mittelpunkten der Kanten aufgeteilt. Die Punkte werden auf die Oberfläche projiziert, wenn die Krümmung entlang der Kante einen Schwellwert überschreitet. Dieses garantiert die Konsistenz des Netzes wenn zwei Polygone sich eine Kante teilen.
Im weiteren werden die beiden Schritte zum Erzeugen eines optimalen Polygonnetzes, das Generieren der Start-Polygone und deren adaptive Verfeinerung, genauer erläutert.

4.2.1 Erzeugen der Start-Polygone

Wie schon erwähnt, wird zum Erzeugen der Start-Polygone eine Methode mit einer einheitlichen Raumunterteilung verwendet. Diese Methode ist sehr einfach umzusetzen und es gibt viele Veröffentlichungen zu dem Thema.
Im Gegensatz zum im Abschnitt 3.2.1 auf Seite 14 beschriebenen tabellarischen Verfahren zum Bestimmen der Schnittpunkte, wird hier der Voxel in sechs Tetraeder unterteilt. Bei einem Tetraeder gibt es weniger Kombinationsmöglichkeiten zur Bildung eines Polygons. Da nur vier Ecken vorhanden sind, gibt es $2^4 = 16$ Möglichkeiten. Mit Tetraedern kann es auch nicht zu Mehrdeutigkeiten, wie sie in Abbildung 12 auf Seite 20 erläutert

wurden, kommen. Die verwendete Aufteilung eines Voxel heißt Coxeter-Freudenthal-Triangulierung. Abbildung 16 zeigt diese und wie die Eckpunkte bezeichnet werden. Eine Coxeter - Freudenthal Triangulierung ist die einfachste mögliche Triangulierung des Euklidischen Raumes. Es wird hier wieder eine Raumaufteilung in Voxel unternommen. Für ein Voxel in R^3 mit den Eckpunkten $p0, ..., p7$, wird die Diagonale $p0p7$ auf jede Seite projiziert. Dieses trianguliert jede Seite des Würfels. Um jetzt die Tetraeder

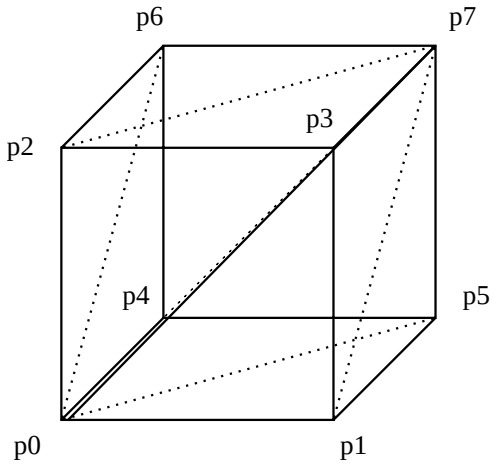

Abbildung 16: Coxeter-Freudenthal Aufteilung eines Voxels (nach Velho [33])

zu erhalten, werden zu jedem Dreieck einer Seite die Punkte der Diagonalen p0p7 hinzu gefügt. Man erhält dann eine Aufteilung des Voxels in 6 dreidimensionale Elemente:

$$\rho 0 = (p0, p1, p3, p7)$$
$$\rho 1 = (p0, p1, p5, p7)$$
$$\rho 2 = (p0, p2, p3, p7)$$
$$\rho 3 = (p0, p2, p6, p7)$$
$$\rho 4 = (p0, p4, p5, p7)$$
$$\rho 5 = (p0, p4, p6, p7)$$

Die implizite Oberfläche, die durch $f(x, yz) = 0$ gegeben ist, schneidet ein Voxel nur, wenn sich der Wert der Funktion f von einem positiven zu einem negativen Wert ändert. Dieser Übergang kann anhand der Funtionswerte in den Eckpunkten des Voxels leicht erkannt werden, wenn die Voxelgröße geeignet gewählt wurde. Abbildung 17 zeigt das nochmal für den 2D-Fall. Es muss also zuerst getestet werden, ob ein Voxel überhaupt die Oberfläche schneidet, bevor die einzelnen Tetraeder untersucht werden können. Hier

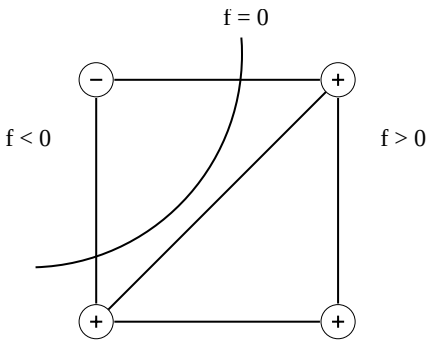

Abbildung 17: Eine 2D-Zelle schneidet eine implizite Kurve (nach Velho [33])

wurde der auf Seite 17 beschriebene Surfacetracker verwendet, damit nur die wirklich beteiligten Voxel betrachtet werden.
Wurde ein Voxel gefunden der an der Oberfläche beteiligt ist, wird für jeden Tetraeder, in die der Voxel zerlegt wurde, eine Prozedur `simplex` aufgerufen. Diese prüft ob und wie das Element die Oberfläche schneidet. Es müssen dabei nicht alle Elemente eines Voxels die Oberfläche schneiden.

Die implizite Oberfläche kann einen Tetraeder eines Voxels in drei oder vier Punkten schneiden. Dabei können zwei Fälle entstehen: entweder wird ein Dreieck oder ein Rechteck erzeugt. Abbildung 18 zeigt die beiden möglichen Fälle. Je nachdem, wie viele

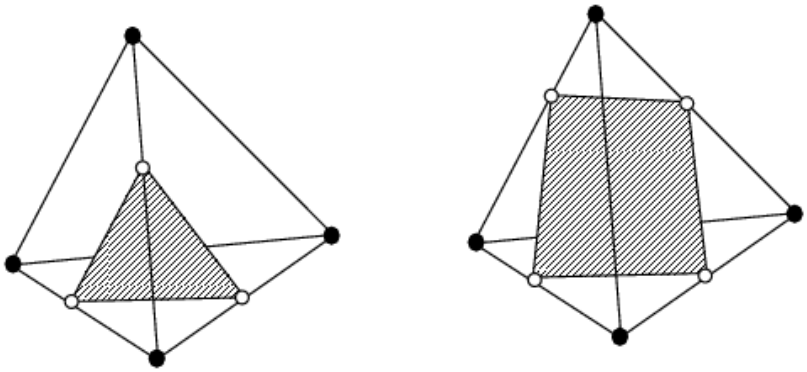

Abbildung 18: Mögliche Schnittpunkte mit einem Tetraeder (nach Velho [33])

Schnittpunkte es gibt, ruft `simplex` die Routinen `tri` oder `quad` auf, um die genauen

Schnittpunkte zu finden. Die entstehenden Dreiecke bilden die Start-Polygone, die für die weitere Verfeinerung benutzt werden. Dazu rufen die Prozeduren `tri` und `quad` die Prozedur `tri_adapt` auf. In Pseudocode sieht das so aus:

```
tri(v0, v1, v2, v3)
{
  i0 = intersect(v0,v1);
  i1 = intersect(v0,v2);
  i2 = intersect(v0,v3);
  tri_adapt(i0, i1, i2);
}
```

Die Prozedur `quad` erzeugt zwei Dreiecke.

```
quad(v0, v1, v2, v3)
{
  i0 = intersect(v0,v2);
  i1 = intersect(v0,v3);
  i2 = intersect(v1,v3);
  i3 = intersect(v1,v2);
  tri_adapt(i0, i1, i2);
  tri_adapt(i0, i2, i3);
}
```

Der Schnittpunkt einer Kante $v0v1$ mit der Oberfläche wird mit der Prozedur `intersect` ermittelt. Dazu wird zuerst mittels linearer Interpolation ein Punkt p gefunden und dann mit der Prozedur `project` auf die Oberfläche projiziert. Diese Prozedur wird später genauer beschrieben.

```
vertex intersect(v0, v1)
{
   t = v0.d/(v0.d-v1.d);
   p = v_add(v_scale(v0.p,1-t),v_scale(v1.p,t));
   v.p = project(1,p,f(p));
}
```

In $v0.d$ steht der Wert, den die implizite Funktion für diesen Punkt liefert. $v0.p$ enthält die Punkt-Koordinaten und $f(p)$ berechnet den Funktionswert für den Punkt p.

4.2.2 Adaptive Verfeinerung der Polygone

Beim Algorithmus zum adaptiven Verfeinern der Polygone werden Dreiecke rekursiv, entsprechend der Adaptions-Kriterien, unterteilt. Dabei werden die Start-Polygone einzeln behandelt. Es muss also nicht erst das gesamte Start-Netz erzeugt werden. Es kann gleich mit dem Unterteilen begonnen werden.

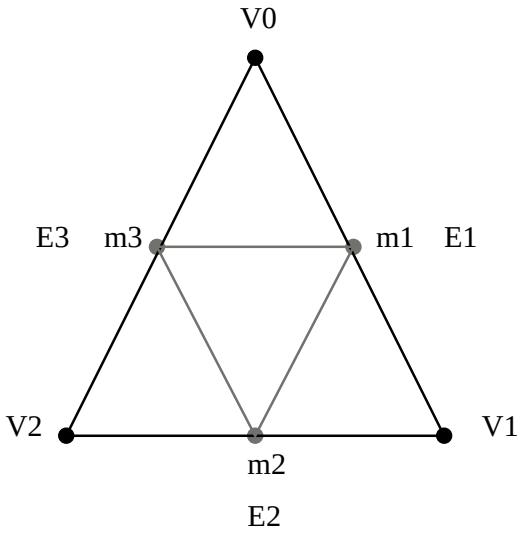

Abbildung 19: Unterteilungs-Schema für ein Dreieck (nach Velho [33])

Ein Dreieck mit dem Ecken v_0, v_1, v_2, wird in vier kleinere Dreiecke aufgeteilt. Dazu werden an den Kanten $e_1 = v_0v_1, e_2 = v_1v_2$ und $e_3 = v_2v_0$ die Mittelpunkte m_1, m_2 und m_3 gebildet und zu neuen Dreiecken zusammengefügt. Abbildung 19 verdeutlicht die Aufteilung. Ob eine Aufteilung erfolgt, hängt von der Krümmung der Oberfläche unter jeder Kante ab. Wenn alle Kanten einen bestimmten Grad an Ebenheit erreicht haben, wird das Dreieck ohne weitere Unterteilung ausgegeben. Wenn aber eine oder mehrere Kanten nicht eben genug sind, hat die Oberfläche eine lokale Krümmung und das Dreieck muss aufgeteilt werden. Dazu werden die Mittelpunkte der Kanten gefunden und die Prozedur `tri_adapt` wird rekursiv für jedes entstehende Teildreieck, also viermal, erneut aufgerufen. Hier die Prozedur in Pseudocode:

```
tri_adapt(v1, v2, v3)
{
   e1 = edge_code(v0, v1);
   e2 = edge_code(v1, v2);
   e3 = edge_code(v2, v0);
```

```
   if(e1 == FLAT && e2 == FLAT && e3 == FLAT)
   {
    tri_output(v0, v1, v2);
   } else
   {
     m1 = midpoint(e1, v0, v1);
     m2 = midpoint(e2, v1, v2);
     m3 = midpoint(e3, v2, v0);
     tri_adapt(v0, m1, m3);
     tri_adapt(v1, m2, m1);
     tri_adapt(v2, m3, m2);
     tri_adapt(m1, m2, m3);
   }
}
```

Die Krümmung unter einer Kante des Dreiecks wird mittels des Winkels α (Abbildung 20) zwischen den Oberflächennormalen n_0 und n_1 an den Kanten-Endpunkten bestimmt. Das erfolgt recht leicht mit dem Kosinus von α, da $\cos(\alpha) = n_0 \cdot n_1$ gilt. Ist der Winkel größer als eine vorher bestimmte Konstante, wird die Kante als nicht eben eingestuft und muss weiter unterteilt werden. Mit α hat man also eine Kontrollmöglichkeit über die Qualität der Polygonbildung. Dieser Parameter sollte vom Nutzer selbst zu bestimmen sein. Er hat damit Einfluss auf die Anzahl der entstehenden Polygone.

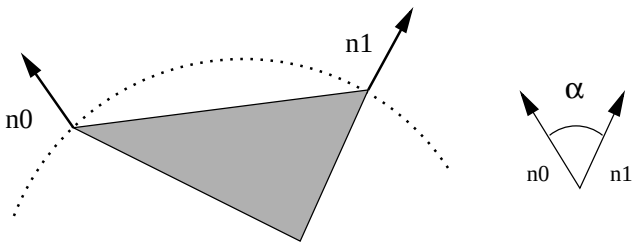

Abbildung 20: Krümmung unter einer Kante

Eine wichtige Prozedur, die in `tri_adapt` aufgerufen wird, ist `midpoint`. Mit ihr wird der Mittelpunkt einer Kante und damit ein neuer Punkt für ein Dreieck berechnet. Ein Mittelpunkt `m` einer Kante v_0v_1 wird zunächst linear berechnet und wenn die Kante nicht eben ist, auf die Oberfläche mit der schon erwähnten Prozedur `project` auf diese projiziert.

Es wird also ein neuer Punkt gesucht, der sich auf der Oberfläche befindet. Es sollte der Punkt auf der Oberfläche sein, der zu dem Punkt p_m einer Kante v_0v_1 am dichtesten

ist. Das heißt, wir suchen einen Punkt, der $f(p_0) = c$ ergibt und gleichzeitig $|p_0 - p_m|$ minimiert. Dieses Optimierungsproblem kann mit verschiedenen numerischen Verfahren gelöst werden. Luiz Velho [33] verwendet hierfür ein physikalisch basiertes Verfahren. Der Mittelpunkt p_m wird unter Nutzung der impliziten Funktion f auf die Oberfläche projiziert. f ist also eine Potenzialfunktion und der Gradient von $|f|$ ein Kraftfeld, welches ein Partikel an die Oberfläche steuert [22]. Man erhält folgende Bewegungsgleichung für ein Massepartikel:

$$\frac{dx}{dt} + \text{sign}(f)\nabla f = 0$$

Diese Differentialgleichung kann mittels einer einfachen Euler-Integration gelöst werden. Dieses erfolgt in der schon vorher benutzten `project` Prozedur. Hier wird die Position eines Partikels p zu dem Zeitpunkt $t + \Delta t$ von dem Startpunkt zur Zeit t berechnet.

$$p^{t+\Delta t} = p^t + \Delta t \text{sign}(f(p^t))\nabla f(p^t)$$

Diese Iteration wird so lange wiederholt, bis das Partikel nahe genug an der Oberfläche ist ($|f(p^t)| < \epsilon$). ϵ ist ein weiterer Steuerparameter, der auf die Qualität des erzeugten Polygonnetzes Einfluss hat. Er steuert, wie genau die Lösung des Optimierungsproblems berechnet wird. Wenn der Zeitschritt der Iteration zu groß ist, kann es zum Überschwingen kommen. Das Partikel wandert dann ständig zwischen der Außen- und Innenseite der Oberfläche hin und her. Um das zu verhindern und Konvergenz zu gewährleisten, wird die Schrittweite beim Durchqueren der Oberfläche auf die Hälfte reduziert.

```
Vector project(step, p, v)
{
   p = v_add(p,v_scale(f_grad(p),sign(v)*step));
   if(abs(v1=f_value(p))<epsilon)
     return p;

   if((v*v1)<0)
     step /=2;

   return project(step, p, v1);
}
```

Abbildung 21 zeigt, wie ein Mittelpunkt einer Kante in ein Partikel gewandelt wird und in mehreren Schritten zur Oberfläche wandert.

Bei dem beschriebenen Verfahren ist darauf zu achten, dass das Gitter des Marching Cube Algorithmus zum Erzeugen des Start-Netzes nicht zu grob ist. Es kann sonst passieren das die Topologie eines Objektes nicht mehr korrekt erkannt wird. Die Auflösung

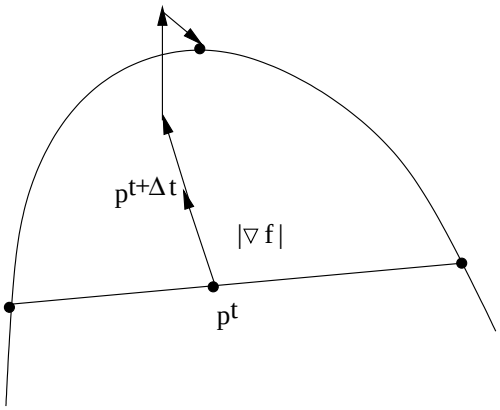

Abbildung 21: Finden eines neuen Samplepunktes auf der Oberfläche (nach Velho [33])

des Gitters muss hoch genug sein, um von jeder Komponente eines Objektes die Topologie zu erkennen. Für einen Torus muss die Gittergröße beispielsweise kleiner als das Loch im Torus sein.

4.3 Andere Verfahren

Mit den beschriebenen Verfahren ist es nicht oder nur sehr schwer möglich, scharfe Kanten zu erzeugen. Ohtake et al. haben in [27] einen Algorithmus entwickelt, mit dem dieses möglich ist. Das Prinzip basiert darauf, dass bei Impliziten Oberflächen mit scharfen Ecken und Kanten die Approximation der Oberfläche besser ist, wenn die Dreiecke des Polygon-Netzes an einem bestimmten inneren Punkt des Dreiecks tangential an der Oberfläche verlaufen. Dazu wird wieder ein Start-Netz mit einem Marching Cube Verfahren erzeugt. Daraus wird mit Hilfe der Dreiecks-Mittelpunkte ein duales Netz generiert. Danach werden die Eckpunkte des dualen Netzes so modifiziert, dass sie auf der Oberfläche liegen. Hiervon wird wieder ein duales Netz gebildet. Von diesem Netz werden die Ecken jetzt so verschoben, dass die Abweichung der Normalen zu den Oberflächen Normalen minimal wird. Das Minimum wird mit einem Quadratischen Fehler [16] bestimmt. Für eine bessere Approximation werden diese Schritte mit einer adaptiven Netzunterteilung verbunden. Abhängig von der lokalen Krümmung werden neue Punkte erzeugt.

Kobbelt et al. zeigen in [20] ebenfalls einen Algorithmus, der besonders für scharfe Kanten geeignet ist. Es werden dort auch Beispiele für die CSG-Modellierung gezeigt. Grundlage für ihren Algorithmus ist eine spezielle Distanz-Funktion und ein erweiterter Marching Cube Algorithmus.

Eine Methode, die auf der Coxeter-Freudenthal-Triangulierung basiert, wird von R. Persiano et al. in [29] beschrieben. Die adaptive Unterteilung wird hier durch *sweeping* erreicht. Also das Vertauschen der Reihenfolge von Eckkoordinaten. Eine *sweeping* Grenze wandert von links nach rechts und kennzeichnet damit die Regionen in denen die Dreiecke schon fertig getestet und akzeptiert wurden. Die noch zu testenden Dreiecke befinden sich auf der rechten Seite der Grenze. Wenn sie den rechten Rand der zu triangulierenden Zone erreicht hat, ist die Unterteilung beendet und alle Dreiecke sind überprüft.

Es gibt viele weitere Veröffentlichungen, die sich mit einer optimalen Triangulierung von Iso-Oberflächen oder allgemein einer Verbesserung von Polygon-Netzen beschäftigen. Zur Optimierung der Polygon-Anzahl eines Netzes sei auf [24] und [31] verwiesen.

5 Das Ayam Metaball Modul

Es wurden die Grundlagen zu impliziten Oberflächen und Verfahren zum Darstellen der Oberflächen vorgestellt. Gegenstand dieser Studienarbeit ist weiterhin die Implementation der vorgestellten Algorithmen. Ein wichtiger Teil, sollte das Erzeugen impliziter Oberflächen mittels adaptiver Verfahren sein. Dazu musste nach geeigneten Verfahren gesucht werden. In dieser Studienarbeit wurden zwei Verfahren näher vorgestellt.

5.1 Software für Implizite Oberflächen

Vor Beginn der Implementation erfolgte eine Recherche wie Metaballs bzw. implizite Oberflächen in anderen Programmen realisiert werden. Hauptsächlich ob und wie ein Modellieren in Echtzeit möglich ist. Es stellte sich heraus, daß kommerzielle Software, wie 3D Studio Max oder Maya, implizite Oberflächen nicht ohne weiteres unterstützen. Hier werden Subdivision Surfaces angeboten. Ein Nachrüsten über Plugins soll allerdings möglich sein. Diese konnten aber nicht getestet werden. Implizite Oberflächen können weiterhin mit dem Raytracer POV-Ray erzeugt werden. Allerdings ist hier keine Echtzeit-Modellierung möglich. Ein interessanter Ansatz ist das Hyperfun-Project [1]. Hier können mit einer Skriptsprache Bilder aus Funktionen der Art: $F(X) >= 0$ erzeugt werden. Mit dem System ist ein geometrisches Modellieren möglich. Auch hier ist aber keine Modellierung in Echtzeit möglich.

In Ayam werden die modellierten Szenen mittels der RenderMan®-Schnittstelle dargestellt. RenderMan® bietet ein relativ umständlich zu benutzendes Primitiv zum Erzeugen von Metaballs. Dieses ist nicht für ein Modellieren in Echtzeit geeignet und wird auch nur von wenigen Renderern unterstützt. Das Ermitteln der Parameter für das Primitiv ist recht kompliziert und unhandlich. Es blieb die Echzeit-Darstellung mittels

Polygonaler Netze. Hierbei wird die erzugte RIB-Datei zum Rendern der Szene allerdings teilweise recht gross (mehrere Megabyte), da nun statt eines Befehls zum Erzeugen eines Metaballs sämtliche Punkte des Polygonalen Netzes abgespeichert werden müssen. Die Grösse kann durch Optimierung des Netzes reduziert werden. Die Optimierungen werden in Sektion 5.5 genauer erläutert.

5.2 Aufbau eines Ayam Moduls

Da Ayam [30] eine Schnittstelle für externe Module zur Verfügung stellt, konnte ein Modul zum Erzeugen von impliziten Oberflächen geschrieben werden. Durch die Module können neue sogenannte *Custom Objects* erstellt werden. *Custom Objects* sind also völlig neue geometrische Objekte. Die Module müssen Schnittstellen (*callbacks*) zur Verfügung stellen, die von Ayam aufgerufen werden. Hiervon sind einige obligatorisch, wie Initialization, Create, Delete, Write/Read, andere sind optional, wie Draw, Shade, Conversion. Welche optionalen Schnittstellen verwendet werden, hängt von der Funktion des Moduls ab. Für das Metaball Modul werden z.B. die genannten optionalen Schnittstellen benötigt. Kann man Parameter des *Custom Objects* verändern, sind weiterhin die Schnittstellen SetProp und GetProp nötig. Das GUI von Ayam wurde mit Tcl/Tk realisiert. Die Parameter für die Module werden ebenfalls mittels Tcl/Tk Oberfläche zugänglich gemacht. Abbildung 24 zeigt die Benutzeroberfläche von Ayam mit Parametern für ein Meta-Objekt. Im Folgenden wird auf die speziellen Anforderungen für das Metaball Modul eingegangen.

Das Metaball Modul für Ayam implementiert zwei neue Typen von *Custom Objects*. Zunächst ein **Meta-Objekt**, welches ein Universum (siehe Seite 14), in dem modelliert wird, zur Verfügung stellt. In diesem Universum wird dann mit **Meta-Komponenten** ein Objekt modelliert. Die **Meta-Komponenten** sind dann Kind-Objekte des jeweiligen **Meta-Objektes**. In einer Szene können mehrere **Meta-Objekte** existieren, die sich nicht untereinander beeinflussen. Nun zur genaueren Beschreibung der einzelnen Komponenten und ihrer *callbacks* im Modul.

5.3 Meta-Objekt

Das **Meta-Objekt** ist der Hauptbestandteil des Metaball Moduls. Bevor modelliert werden kann, muss ein solches von Ayam erzeugt werden. Dieses erfolgt durch Anwählen des Menüeintrages *Create→Custom Object→MetaObj*. Hiermit wird eine Metaumgebung, das Universum, zum Modellieren mit Metaballs geschaffen. Die Umgebung enthält die einzelnen **Meta-Komponenten** die zusammen ein Objekt bilden. Das **Meta-Objekt** stellt weiterhin Routinen zum Laden und Speichern zur Verfügung und sorgt für das Generieren und Rendern der Metaballs.

5.3.1 Notify *callback*

Der `notify` *callback* wird immer dann aufgerufen, wenn sich an der Szene etwas geändert hat. Dies ist der Fall wenn z.B. eine **Meta-Komponente** verschoben, skaliert oder rotiert wurde. Eine Rotation wird erreicht, indem das Gitter bzw. der zu betrachtende Punkt vor der Bestimmung des Feldwertes im Raum transformiert wurde. Eine Verzerrung in Richtung der x-, y- oder z-Achse wird durch eine Multiplikation der Punkt-Koordinaten mit den Skalierungswerten möglich. In `notify` wird also vor dem Starten des Marching-Cubes bestimmt, in welchem Umfang ein Objekt verschoben oder rotiert wurde. Daraus wird dann eine Rotationsmatrix erzeugt die beim Berechnen der Dichtewerte für einen Punkt im Raum auf diesen angewendet wird. So erreicht man eine Verzerrung einzelner Komponenten ohne auf die zu Grunde liegenden Formeln der einzelnen Komponenten Rücksicht nehmen zu müssen. Abbildung 26 entstand mit diesen Möglichkeiten der Verzerrung. `notify` ruft nach der Bestimmung der Rotationsmatrix für eine Komponente die Routine calceffect auf, welche die Berechnung der Polygone für ein **Meta-Objekt** ausführt. Die Polygone werden dabei zwischen gespeichert, damit sie für die Darstellung nicht jedesmal neu berechnet werden müssen. Es wird folglich nur neu berechnet, wenn sich am Objekt etwas geändert hat. Die eigentliche Darstellung erfolgt dann im *draw-* bzw *shadecallback*. Diese greifen auf die von `notify` berechneten Polygone zurück.

5.3.2 Berechnung der Polygone

Zur Bestimmung der Polygone wird der in Abschnitt 3.2.1 beschriebene Marching Cube Algorithmus verwendet. Dieser erzeugt ein Start-Netz wie es später für die adaptive Verfeinerung nötig ist. Dabei wurde mit dem beschriebenen Tabellen-Verfahren zum Ermitteln der Schnittpunkte gearbeitet. Die Tabelle hierfür konnte auf der Homepage von Paul Bourke [12] gefunden werden. Zur Optimierung der Geschwindigkeit und Minimierung des Speicherverbrauchs wird ein Surfacetracker, siehe Seite 17, benutzt. Hierbei wird kein Gitter mit Würfeln benutzt, sondern nur eine einzige Cube-Struktur. Die Struktur wird nur einmal für den Start-Würfel mit den Eckkoordinaten gefüllt. Da immer nur Nachbar-Würfel betrachtet werden, kommt man zu den neuen Koordinaten durch eine einfache Addition bzw. Subtraktion der Würfelseiten-Länge. Für einen Nachbar-Würfel brauchen nur vier neue Feldwerte berechnet werden, da vier vom alten Würfel übernommen werden. Mit einem zusätzlichen 3D-Feld werden die schon betrachteten Würfel im Surfacetracker-Algorithmus markiert. Abbildung 22 zeigt den Ablauf beim Erzeugen der Polygone. Die Komponenten eines **Meta-Objektes** befinden sich in einer Liste und werden nacheinander bearbeitet.

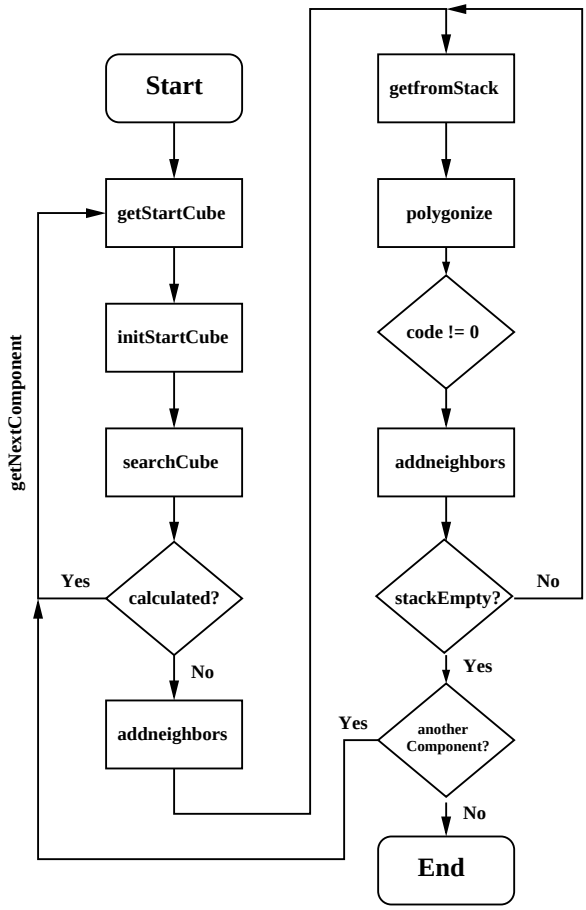

Abbildung 22: Programmablauf zum Erzeugen der Polygone

getStartCube Hier wird ausgehend von den Koordinaten einer **Meta-Komponente** im Raum bestimmt, welcher Würfel im Gitter sich im Inneren der Komponente befindet. Dazu werden seine Koordinaten im Gitter (siehe Abbildung 8) bestimmt.

initStartCube Initialisiert die Ecken der Würfel-Struktur mit den Raumkoordinaten und Dichte-Werten seiner Position.

searchCube Nachdem ein Würfel im Inneren einer Komponente bestimmt wurde, wird dieser von Innen nach Außen an die Oberfläche bewegt, bis er diese schneidet. Zuerst wird der Würfel nach oben bewegt. Wird keine Oberfläche erreicht, muss eine Suche in den anderen Richtungen (unten, links, rechts ...) erfolgen. Die Feststellung, ob der

Würfel die Oberfläche schneidet, erfolgt mit der weiter unten beschriebenen Routine `polygonize`. Diese liefert den auf Seite 14 beschriebenen Edge-Code zurück. Der Code zeigt an, welche Ecken sich oberhalb der Oberfläche befinden. Ist der Code null, muss sich der Würfel noch innerhalb der Oberfläche befinden. Dann wird der Würfel eine Position weiter verschoben, indem nur die Eckkordinaten verändert werden.

calulated Hier wird geprüft ob ein Würfel schon betrachtet wurde. Dazu wird ein Byte Array benutzt, das für jeden Würfel im Gitter einen Eintrag hat. Ein Byte wird gesetzt wenn der entsprechende Würfel schon betrachtet wurde. Damit bei jedem Durchlauf das Array nicht gelöscht und neu erzeugt werden muss, wird der Wert, der ein Byte als schon besucht makiert, jedesmal um eins erhöht. Somit wird jedesmal ein anderer Wert eine Stelle als schon besucht markieren.

addneighbors Dies ist ein wichtiger Teil des Surfacetrackers. Es werden alle Nachbarn des zu betrachtenden Würfels zum Stack hinzugefügt. Es muss also geprüft werden, ob sich vorne, hinten, oben, unten, links und rechts noch ein Würfel befindet der zur Oberfläche beiträgt. Ob es relevante Nachbar-Würfel gibt, kann mittels des Ecken-Codes bestimmt werden. Durch den Code kann man ermitteln welche Kanten die Oberfläche schneiden. An diesen Kanten muss also auch ein Nachbar-Würfel existieren, der die Oberfläche schneidet und somit zur ihr beiträgt. Der Stack besteht hier aus einem Speicherbereich in dem die Würfel-Strukturen sich hintereinander befinden. Ein Index-Wert zeigt dabei immer auf den letzten Eintrag. Gegebenenfalls muss der Stack durch ein *realloc()* vergrössert werden.

polygonize Mit der Routine `polygonize` werden die Polygone erzeugt. Zuerst wird hier der Ecken-Code bestimmt, um festzustellen welche Kanten des Würfels die Oberfläche des Objektes schneiden. Die Berechnung der Polygone erfolgt dann mit einem normalen Marching-Cube, wie er in Sektion 3.2.1 beschrieben wurde. Wenn der Nutzer eine adaptive Berechnung im eingestellt hat, wird anstelle des normalen Marching-Cube das adaptive Verfahren von Luiz Velho [34] benutzt.

Bei der Implementation wurde zuerst ohne einem Surfacetracker, siehe Seite 17, gearbeitet. Dieses Verfahren ist einfach und schnell umzusetzen. Doch es hat zwei Nachteile. Der Speicher-Verbrauch für das Cube Array steigt sehr schnell kubisch an. Zum Beispiel wurden für ein 80x80x80 Gitter etwa 45 Mb benötigt. Weiterhin ist die Zeit zum Erzeugen der Polygone unakzeptabel. Auf einem Rechner mit einer Athlon 1200Mhz CPU ist gerade noch ein Modellieren mit wenigen Komponenten in Echtzeit möglich.
Es wurde daher das beschriebene Surfacetracker-Verfahren implementiert. Ergebnis war ein spürbarer Geschwindigkeits-Gewinn. Da in dem Ayam-Modul anstelle eines 3D-

Arrays von Cube-Strukturen nur eine einzige Struktur verwendet wird, konnte auch der Speicher-Verbrauch stark gesenkt werden.

5.3.3 Adaptive Berechnung

Für eine adaptive Berechnung der Polygone wurde das Verfahren von Luiz Velho [34] implementiert. Dieser Algorithmus ist gut dokumentiert und ließ sich mit dem Marching Cube Algorithmus sowie dem Surfacetracker gut verbinden. Das Verfahren wird in Sektion 4.2 ab Seite 24 ausführlich beschrieben. Der Algorithmus basiert darauf, dass nicht ein Würfel unterteilt wird, sondern die schon erzeugten Polygone adaptiv verfeinert werden. Die Methode beginnt mit einer groben polygonalen Approximation der Oberfläche und unterteilt jedes Polygon rekursiv in Abhängigkeit der lokalen Krümmung der Oberfläche. Ein Dreieck wird dabei in vier kleinere Dreiecke aufgeteilt. Diese werden dann wieder dahingehend untersucht, wie gut sie sich der Oberfläche anpassen. Gegebenenfalls ist dann eine weitere Unterteilung nötig. Kriterium ob ein Dreieck weiter unterteilt werden muss, ist der Winkel zwischen den Normalen-Vektoren der Endpunkten einer Kante. Vergleiche hierfür Abbildung 20.

Durch dieses Verfahren ist es nun möglich relativ scharfe Kanten zu erzeugen. Es wäre sonst eine Erhöhung der Auflösung des Marching Cubes Gitters nötig, wodurch sich die Anzahl der entstehenden Polygone massiv erhöhen würde. Abbildung 25 verdeutlicht den Unterschied. Abbildung 23 zeigt wie an kritischen Stellen zusätzliche Polygone erzeugt wurden. Die adaptive Berechnung lässt sich durch verschiedene Parameter beeinflussen. Es zeigte sich, dass die Wahl der Parameter kritisch ist. Teilweise sind noch kleine Artefakte im Bild zu erkennen. Die Hauptursache hierfür ist wahrscheinlich die Berechnung der Normalen. Diese werden nach dem in Abschnitt 3.2.2 auf Seite 17 erläuterten Verfahren approximiert. Das Verfahren scheint bei starken Krümmungen Fehler zu produzieren. Es entstanden so scheinbar Löcher, die sich dann aber als Polygone mit falschen Ecken-Normalen herausstellten. Hier muss eventuell nach einer Verbesserung gesucht werden. Abhilfe konnte beim Experimentieren mit den Parametern und der Erhöhung der Gitterauflösung gefunden werden. Diese sind für den Benutzer frei einstellbar gemacht worden (Bild 24). Ein Parameter ist die *Flatness*, also der auf Seite 29 beschriebene Kosinus der Normalen der Eckpunkte einer Kante. Hier wird bestimmt, wie genau die Approximation der Oberfläche erfolgen soll. Weiter bestimmbar ist der Epsilonwert für die Genauigkeit der Projektion der `project` Prozedur und der Zeitschritt für die Iteration beim Lösen der Differentialgleichung. Diese Parameter müssen eventuell für einige Objekte angepasst werden. Intern musste weiterhin ein Iterationszähler einbaut werden, der die Unterteilung der Dreiecke nach einer bestimmten Anzahl von Iterationen abbricht. Durch die etwas fehlerhafte Berechnung der Normalen kann es sonst zu Situationen kommen, bei denen die eigentlichen Abbruchbedingungen, also die *Flatness* und der Epsilonwert, nicht zum Abbruch führen.

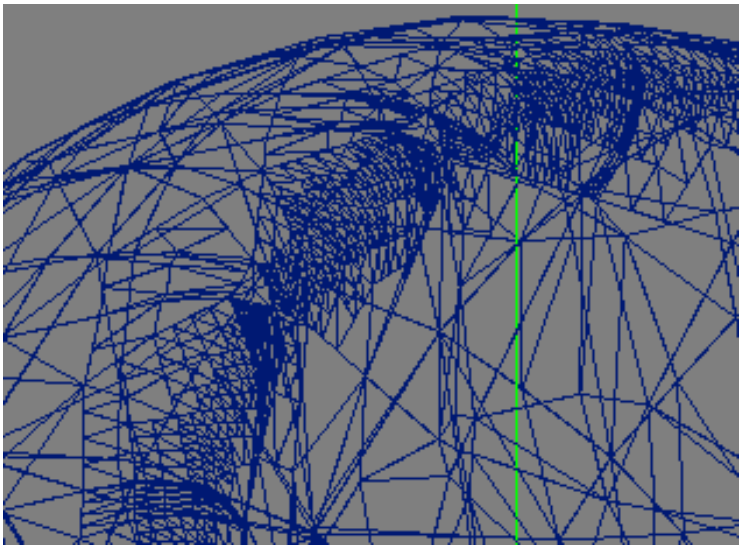

Abbildung 23: Detailzoom auf eine Stelle mit hoher Krümmung

Auf Abbildung 24 erkennt man weiterhin den Parameter *NumSamples*. *NumSamples* gibt an, in wie viele Voxel das Universum unterteilt werden soll. Dieser Wert gilt für alle drei Dimensionen. Ein Wert von 80 bedeutet also, dass das Universum in 80x80x80 Voxel unterteilt wird. Hiermit wird folglich die Feinheit des Gitters eingestellt. Es ist oft notwendig, diesen Wert zu erhöhen, um ein besseres Ergebnis zu erhalten. Damit steigt aber auch der Rechenaufwand, so daß zum Modellieren erstmal ein kleinerer Wert günstiger ist. Für die fertige Szene kann dann zum Rendern die Auflösung erhöht werden. Es muss beachtet werden, dass dabei auch der Speicherverbrauch steigt, da mehr Polygone erzeugt werden und der Cube-Stack (siehe Seite 17) grösser wird. Weiterhin kann bestimmt werden, ob eine adaptive Berechnung erfolgen soll. Zum Modellieren in Echtzeit mit eingeschalteter Adaptivität ist ein sehr schneller Rechner erforderlich. Deswegen kann die Adaptivität optional eingeschaltet werden. Wenn ein Objekt modelliert wurde, kann es anschließend adaptiv gerendert werden. Die erreichten Rechenzeiten für die Bilder 2 und 14, in verschiedenen Gitterauflösungen, zeigt Tabelle 1 auf Seite 43. Die höhere Gitterauflösung wurde jeweils so gewählt, dass die Szene mit der normalen Berechnung aussieht wie mit der geringeren Auflösung und adaptiver Berechnung. Die Szene Kugelzylinder (Abbildung 27) hat relativ hohe Rechenzeiten, da hier eine Kugel mit einer Custom Funktion, die eines Zylinders, geschnitten wurde. Die Custom Funktion sind frei wählbar und werden direkt von Tcl berechnet.

5.3.4 Bestimmung der Normalen

Um ein Bild schattiert darzustellen, werden die Normalen der Eckpunkte der Polygone benötigt. Da es umständlich oder gar nicht möglich ist, die Feld-Funktionen zu differenzieren, wird der Gradient approximiert. Die Bestimmung der Normalen erfolgt mit dem in Abschnitt 3.2.2 auf Seite 17 beschriebenen Vorwärts-Differenzen. Das dort benutzte Δ ist recht klein und wurde in der Implementation auf ein hundertstel der Seitenlänge eines Voxels gesetzt. Zu kleine Werte für Δ können allerdings zu numerischen Fehlern führen. Für die Implementation in Ayam wurden beide Varianten, Vorwärts- und Zentral-Differenzen, getestet. Dabei wurde festgestellt, dass die aufwändigeren Zentral-Differenzen kein besseres Ergebniss lieferte. Es konnte nur eine spürbare Steigerung der Rechenzeit beobachtet werden, da in dieser Variante die Funktion $f(P)$ öfter aufgerufen werden muss.

5.3.5 RIB-Export

Der RIB-Export ist ein wichtiger *callback*, welcher zum Rendern des Objektes durch die Renderman®-Schnittstelle nötig ist. Da Ayam nur modelliert und nicht selber rendert, müssen alle Objekte in das RIB Format (RenderMan Interface Bytestream Protocol) konvertiert werden. Das RenderMan®-Interface stellt mit *RiBlobby* zwar ein Primitiv für implizite Flächen zur Verfügung, aber dieses ist umständlich zu benutzen. Deshalb wird das modellierte Objekt als Polygone mit *RiPolygon* gespeichert. Da jedes Polygon einzeln gespeichert werden muss, können diese Dateien sehr gross werden. Dies kann mehre Megabyte erreichen, da viele Punkte doppelt vorhanden sind. Eine Optimierung der Punktanzahl wird in 5.5 beschrieben.

5.4 Meta-Komponente

Die **Meta-Komponente** bestimmt welche Form eine Komponente im Universum hat. Das Modul wurde so gestaltet, daß man verschiedene Feld-Funktionen, zum Erzeugen von Grundkörpern, auswählen kann. Neben festen Funktionen für Kugeln, Tori und Würfel, können auch eigene Funktionen benutzt werden, die über ein Interface angegeben werden. Über das Interface der **Meta-Komponente** können auch verschiedene Parameter einiger Grundkörper verändert werden. Bei einem Torus sind dies z.B. der innere und äussere Radius. Weiterhin gibt es die Möglichkeit Objekte als negativ zu kennzeichnen. Damit ist es möglich, Löcher in andere Objekte zu schneiden oder Einbuchtungen zu erzeugen.

5.5 Optimierung

Da nur einzelne Polygone erzeugt werden, entstehen im Polygon-Netz unnötig viele doppelte Punkte. Dies erhöht den Speicherverbrauch und die RIB-Dateien werden unnötig

groß. Eine einfache Optimierung des Speicherverbrauchs kann man schon erreichen, wenn man mehrfach verwendete Punkte aus dem Netz entfernt. Die Anzahl der Punkte kann im Durchschnitt auf ein Sechstel reduziert werden, da jeder Eckpunkt in etwa von sechs Polygonen benutzt wird. Diese Eliminierung kann schnell mit einer Hash-Tabelle zum Testen identischer Punkte durchgeführt werden. Man benötigt hier also eine Hash-Funktion, die mit 3D-Punkten arbeitet. Eine passende Funktion wird in [32] vorgestellt. Hier eine etwas abgewandelte Version:

$$\text{index} = (\text{int})(3 * \text{abs}(x) + 5 * \text{abs}(y) + 7 * \text{abs}(z)) * C + 0.5) \bmod T$$

- C ist ein Skalierungsfaktor
- T die Tabellengrösse
- **mod** ist der Modulo Operator
- 0.5 dient zur Rundung

Mit `index` wird dann in einem Array nach schon betrachteten 3D Punkten gesucht. Mit dieser Hash-Funktion wurde ein Optimierer für Polygonale Netze in Ayam realisiert, der eine Reduzierung der Eckpunkte auf etwa ein Sechstel erreicht. Hierbei wurde ein so genanntes offenes Hashing verwendet. Wenn ein errechneter Index schon auf eine belegte Stelle in einem Array zeigt, werden die Einträge in einer einfachen Liste gehalten. Der Listenkopf befindet sich dann an der errechneten Indexstelle im Array. Wenn bei der Suche nach einem Eintrag eine Liste gefunden wird, muss diese noch nach dem gesuchten Element untersucht werden. Der Optimierer arbeitet auch bei einer großen Anzahl von Punkten im Netz schnell. Die Ergebnisse des Optimierers zeigt Tabelle 2. Man erkennt, dass die Anzahl der Punkte auf etwa ein Sechstel reduziert wird.

Diese Optimierung wurde in dem von Ayam zur Verfügung gestellten **PolyMesh-Objekt** implementiert. Damit nun die Ergebnisse vom **Meta-Objekt** optimiert werden können, müssen sie vorher in ein Polygon-Netz bzw. *PolyMesh* konvertiert werden.

5.6 Ergebnisse

Mit dem implementierten Ayam-Modul und den beschriebenen Algorithmen ist das Erzeugen von Objekten aus impliziten Oberflächen in Echtzeit möglich. Abbildung 14 auf Seite 22 zeigt ein Objekt, das mit dem Metaball-Modul erzeugt/modelliert wurde. Hier wird auch der Vorteil einer adaptiven Berechnung deutlich. Es können mit weniger Polygonen bessere Resultate erzielt werden.
Eine Vergrößerung von Abbildung 14 ohne Schattierung zeigt Abbildung 23. Es wurde hier auf die kritischen Stellen an der entstandenen Kante gezoomt. Man erkennt deutlich, dass mehr Polygone adaptiv an diesen Stellen erzeugt wurden. Um den gleichen

Effekt ohne die Adaptivität zu erreichen, müsste man die Gitterauflösung stark erhöhen. Das hätte zur Folge, dass mehr Polygone als nötig erzeugt werden (siehe Abbildung 25). Abbildung 26 zeigt, dass es mit dem Modul möglich ist, komplexe Objekte zu erstellen. Bei diesem Objekt wurden nur Kugeln als Grundkörper verwendet. Diese wurden dann durch Skalierung und Rotation entsprechend verzerrt. Die Modellierung in Echtzeit im adaptiven Modus erfordert allerdings viel Rechenleistung. Deshalb wird dieser Modus optional angeboten. Hier muss noch weiter nach Optimierungen gesucht werden.

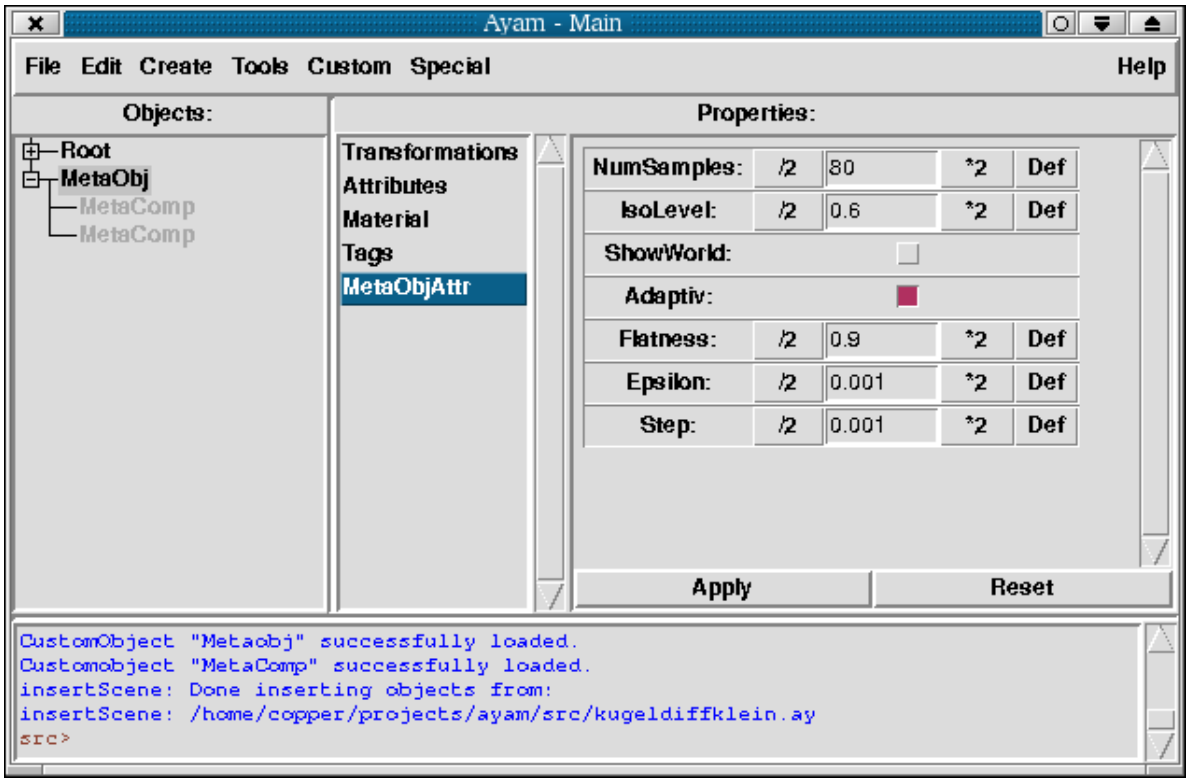

Abbildung 24: Einstellungen für ein Meta-Object in Ayam

6 Zusammenfassung

Diese Studienarbeit beschäftigte sich mit dem Erzeugen und Darstellen von impliziten Oberflächen. Dabei wurde auf die Grundlagen impliziter Funktionen und ihre Erweiterung zu impliziten Oberflächen eingegangen. Es wurde das Modellieren mit impliziten Oberflächen erläutert und die nötigen Grundfunktionen wie *Blobby Objects, Metaballs* und *Soft Objects* vorgestellt. Danach wurde das Darstellen impliziter Oberflächen betrachtet. Dabei wurde kurz die Möglichkeit des Raytracings und ausführlich das Erzeugen Polygonaler Netze erklärt. Zur Darstellung mittels Polygonaler Netze wurde der Marching Cube Algorithmus vorgestellt. Dabei wurden die mit ihm auftretenden Probleme erörtert und Lösungen vorgeschlagen. Zur Beschleunigung der Berechnung wurde ein Surfacetracker Verfahren vorgestellt. Mit diesem Verfahren werden nur die wirklich benötigten Voxel in einer Szene berechnet. Anschließend wurden Verfahren und Algorithmen für eine adaptive Polygonerzeugung genauer betrachtet. Diese adaptive Verfahren sind nötig, um nur an kritischen Stellen (z.B. eine starke Krümmung) an einem Objekt zusätzliche Polygone zu erzeugen und somit ein besseres Ergebnis zu erreichen. Es wurden zwei Verfahren vorgestellt, die sich darin unterscheiden, wo die Adaptivität ansetzt. Zum einen die Unterteilung des Raumes und zum anderen eine adaptive Unterteilung von Polygonen. Das Verfahren mit der adaptiven Unterteilung der Polygone wurde genauer vorgestellt. Daran anschließend wurde kurz auf andere existierende Lösungen eingegangen. Nachdem alle Grundlagen zum Erzeugen und Darstellen impliziter Oberflächen erläutert wurden, konnte das im Rahmen der Studienarbeit entwickelte Ayam Metaball Modul vorgestellt werden. Dabei wurde auf Details der Implementation eingegangen und eine Bewertung der mit dem Modul erreichten Ergebnisse vorgenommen. Mit dem Modul ist es gut möglich, komplexe Objekte zu erstellen. Aus den Tabelle 2 und besonders Abbildung 25 erkennt man, dass durch adaptive Berechnung weniger Polygone nötig sind, um ein besseres Bild zu erzeugen. Bei der Berechnung unter Berücksichtigung der Adaptivität muss eventuell noch an der Geschwindigkeit optimiert werden. Es wurde festgestellt, dass Verbesserungsbedarf beim Ermitteln der Oberflächennormalen besteht. Hierfür muss nach geeigneten Algorithmen gesucht werden.

Szene	Gittergröße	Rechenzeit normal	Rechenzeit adaptiv
kugeldiff	80	5 ms	120 ms
kugeldiff	360	115.5 ms	1272.6 ms
metaflight	120	62 ms	973.7 ms
metaflight	360	556.9 ms	7934.9 ms
kugelzylinder	80	542.1 ms	6534.1 ms
kugelzylinder	150	2174.7 ms	20227.2 ms

Tabelle 1: Rechenzeiten mit einer Athlon 1200Mhz CPU

Szene	Gittergröße	Polygone	Punkte normal	Punkte optimiert
kugeldiff	80	708/10691	2124/32073	354/6359
kugeldiff	360	14100/44325	42300/132975	7046/22548
metaflight	120	3719/19736	11157/59208	1856/11211
metaflight	360	33739/107083	101217/321249	16866/54318
kugelzylinder	80	2788/9574	8364/28722	1392/4964
kugelzylinder	150	10740/32484	32220/97452	5368/16288

Tabelle 2: Polygone und Punkte für normale und adaptive Berechnung

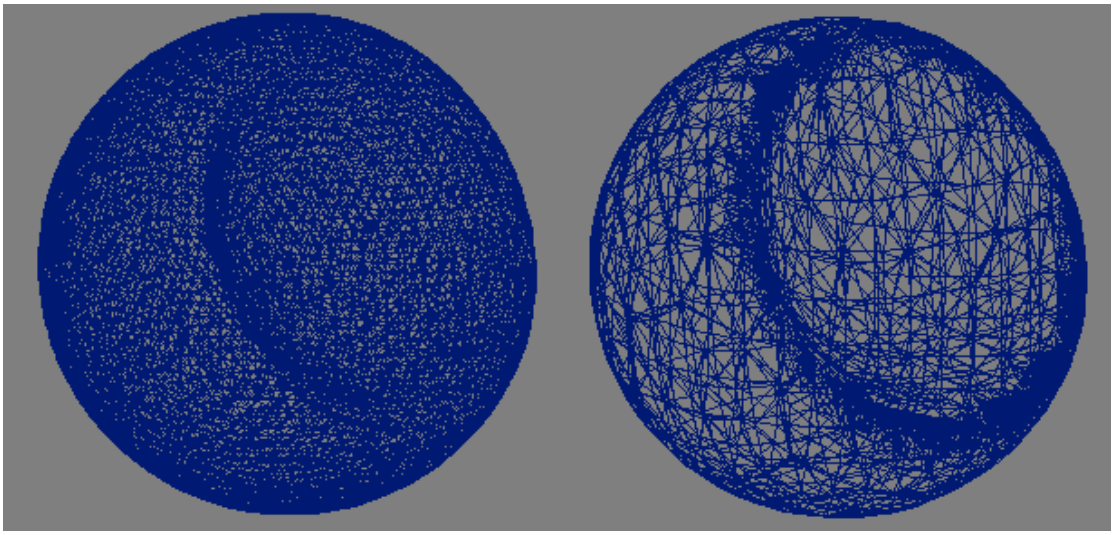

Abbildung 25: Metaobjekt bestehend aus einer positiven und einer negativen Komponente, links normal (20364 Polygone), rechts adaptiv (8778 Polygone)

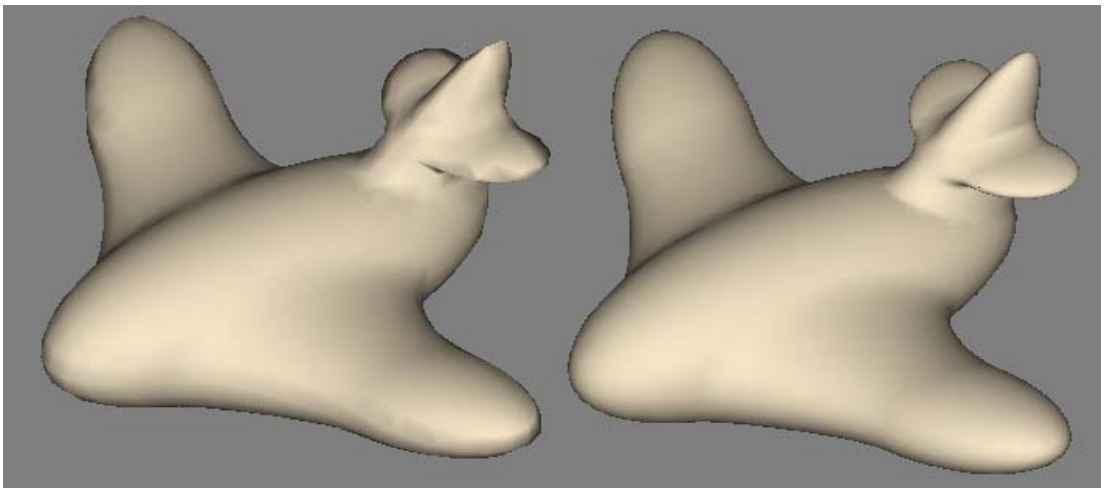

Abbildung 26: Auflösung 120, links normal, recht adaptiv

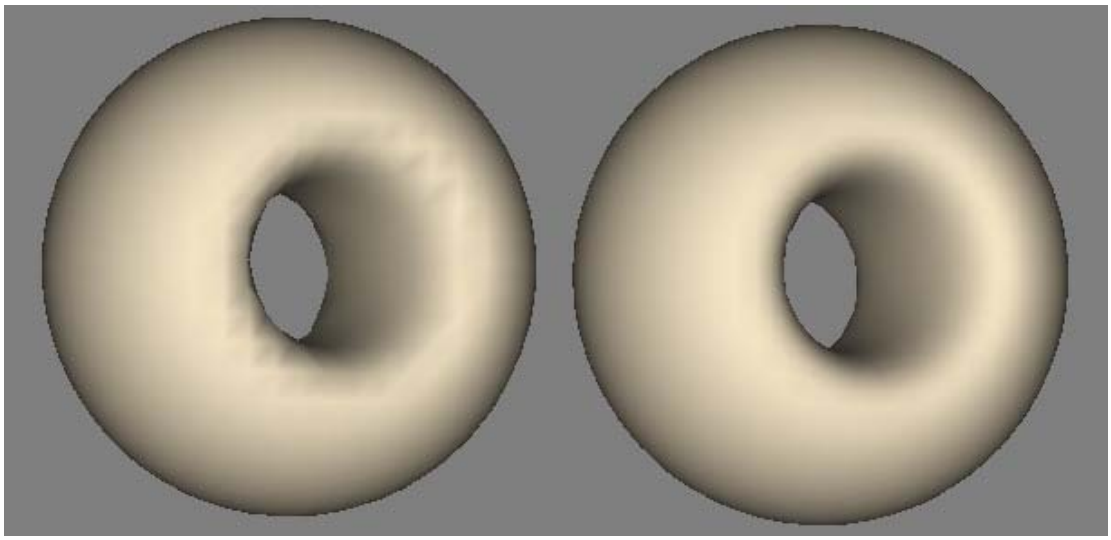

Abbildung 27: Auflösung 80, links normal, recht adaptiv

Literatur

[1] HyperFun Project – Language and Software Tools for F-rep Geometric Modelin. http://www.hyperfun.org/.

[2] P. de Casteljau: Courbes et Surfaces à Poles. Technischer Bericht A. Citroen, Paris, 1963.

[3] F.Evans and S. Skiena and A. Varshney: Completing sequential triangualtion is hard, Technical Report Deparment of Computer Science, State University of New York, Stony Book. March 1996.

[4] A. Belyaev and Y. Ohtake. A comparsion of mesh smoothing methods. *Submitted for Publication*, 2001.

[5] P. Bézier. Définition numérique des courbes et surfaces i,ii. *Automatisme*, 11,12:625–632 and 17–21, 1966/67.

[6] James F. Blinn. A generalization of algebraic surface drawing. *ACM Transactions on Graphics*, 1(3):235–256, July 1982.

[7] Jules Bloomenthal. Polygonisation of implicit surfaces. *Computer Aided Geometric Design*, 5:341–355, 1988.

[8] Jules Bloomenthal. Bulge elimination in implicit surface blends. In *Implicit Surfaces'95*, pages 7–20, Grenoble, France, April 1995. Proceedings of the first international workshop on Implicit Surfaces.

[9] Jules Bloomenthal. *Skeletal Design of Natural Forms*. PhD thesis, The University of Calgary, January 1995.

[10] Jules Bloomenthal. Implicit surfaces, 1997. A reference article on implicit surfaces, to appear in the Encyclopedia of Computer Science and Technology (Marcel Dekker, Inc., NY).

[11] Jules Bloomenthal and Ken Shoemake. Convolution surfaces. *Computer Graphics*, 25(4):251–256, July 1991. Proceedings of SIGGRAPH'91 (Las Vegas, Nevada, July 1991).

[12] Paul Bourke. Paul bourke personal page. http://astronomy.swin.edu.au/∼pbourke/.

[13] E. Catmull and J. Clark. Recursively generated B-spline surfaces on arbitrary topological meshes. *Computer-Aided Design*, 10:350–355, September 1978.

[14] D. Doo and M. Sabin. Behaviour of recursive division surfaces near extraordinary points. *Computer-Aided Design*, 10:356–360, September 1978.

[15] Francine Evans, Steven S. Skiena, and Amitabh Varshney. Optimizing triangle strips for fast rendering. In Roni Yagel and Gregory M. Nielson, editors, *IEEE Visualization '96*, pages 319–326, 1996.

[16] Michael Garland. *Quadric Based Polygonal Surface Simplification*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsbourgh, 1999.

[17] John Hart. Ray tracing implicit surfaces. In *Modeling, Visualizing and Animating Implicit Surfaces*, pages 13.1–13.15, 1993. SIGGRAPH Course Notes 25.

[18] A.Belyaev H.Yagou, Y.Ohtake. Mesh smoothing via mean and median filtering applied to face normals. *Proc. of Geometric Modeling and Processing 2002*, July 2002.

[19] Amitabh Varshney Steven Skiena Elvir Azanli Jihad El-Sana, Fancine Evans. Efficiently computing and updating triangle strips for real-time rendering. *Computer-Aided Design*, 32:753–772, 2000. special issue on Multiresolution Geometrics Models.

[20] Leif P. Kobbelt, Mario Botsch, Ulrich Schwanecke, and Hans-Peter Seidel. Feature-sensitive surface extraction from volume data. In Eugene Fiume, editor, *SIGGRAPH 2001, Computer Graphics Proceedings*, pages 57–66. ACM Press / ACM SIGGRAPH, 2001.

[21] Paul Haeberli Kurt Akeley and Derrick Burns. tomesh.c, a c-program on the sgi developer's toolbox cd, 1990.

[22] D. Terzopoulos L. Velho L. H. de Figueriedo, J. de M. Gomes. Physical-based methods for polygonization of implicit surfaces. *Proceedings of Graphics Interface 92*, 1992.

[23] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. In Maureen C. Stone, editor, *Computer Graphics (SIGGRAPH '87 Proceedings)*, volume 21, pages 163–169, July 1987.

[24] C. Montani, R. Scateni, and R. Scopigno. Discretized marching cubes. In R. D. Bergeron and A. E. Kaufman, editors, *Visualization '94 Proceedings*, pages 281–287, Washington D. C., USA, 1994. IEEE Computer Society, IEEE Computer Society Press.

[25] Paul Ning and Jules Bloomenthal. An evaluation of implicit surface tilers. *IEEE Computer Graphics and Applications*, 13(6):33–41, 1993.

[26] Hitoshi Nishimura, Makoto Hirai, Toshiyuki Kawai, Toru Kawata, Isao Shirakawa, and Koichi Omura. Object modeling by distribution function and a method of image

generation. *The Transactions of the Institute of Electronics and Communication Engineers of Japan*, J68-D(4):718–725, 1985. In Japanese (translated into English by Takao Fujiwara while at Centre for Advanced Studies in Computer Aided Art and Design, Middlesex Polytechnic, England, 1989).

[27] Yutaka Ohtake and Alexander G. Belyaev. Dual/primal mesh optimization for polygonized implicit surfaces. *Proceedings of the seventh ACM symposium on Solid modeling and applications Saarbrücken*, pages 171–178, 2002.

[28] A. Opalach and S. C. Maddock. An overview of implicit surfaces. In *Introduction to Modelling and Animation Using Implicit Surfaces*, pages 1.1–1.13, 1995.

[29] R. Persiano, J. Comba, and V. Barbalho. An adaptive triangulation refinement scheme and construction, proceedings of the vi brazilian symposium on computer graphics and image processing (sibgrapi), out. 1993., 1993.

[30] Randolf Schultz. Ayam – a free 3D modeling environment for the RenderMan interface. http://www.ayam3d.org/, 2001.

[31] Raj Shekhar, Elias Fayyad, Roni Yagel, and J. Fredrick Cornhill. Octree-based decimation of marching cubes surfaces. In *IEEE Visualization*, pages 335–342, 1996.

[32] Martin Kuchar Vaclav Skala. Hash funktion for geometry reconstruction in rapid prototyping. *Conference of Scientific Computing*, pages 379–387, 2000.

[33] Luiz Velho. Simple and efficient polygonization of implicit surfaces. *Journal of Graphics Tools*, 1(2):5–25, 1996.

[34] Luiz Velho, Luiz Henrique de Figueiredo, and Jonas Gomes. Hierarchical generalized triangle strips. *The Visual Computer*, 15(1):21–35, 1999.

[35] Alan Watt. *3D-Computergrafik*. Pearson Studium, 2002.

[36] Andrew P. Witkin and Paul S. Heckbert. Using particles to sample and control implicit surfaces. In Andrew Glassner, editor, *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, Computer Graphics Proceedings, Annual Conference Series, pages 269–278. ACM SIGGRAPH, ACM Press, July 1994. ISBN 0-89791-667-0.

[37] Brian Wyvill, Craig McPheeters, and Geoff Wyvill. Animating soft objects. *The Visual Computer*, pages 235–242, August 1986.

[38] Geoff Wyvill, Craig McPheeters, and Brian Wyvill. Data structure for soft objects. *The Visual Computer*, pages 227–234, August 1986.