Bildbasiertes Constructive Solid Geometry

Klasse der Goldfeather-Algorithmen

Ausarbeitung zum
Seminar Computergrafik
Sommersemester 2002

1 Einführung

Für den Entwurf von 3D-Modellen kann man entweder auf vorhandene Daten zurückgreifen (etwa einen 3D-Scanner) oder in einem künstlerischen Prozess die Geometrie selbst entwerfen. Letzteres hat die Eigenart, dass meist eine experimentelle Vorgehensweise, sogenanntes Trial-and-Error, zum Ziel führt. Der Designer möchte verschiedene Konzepte am Bildschirm interaktiv ausprobieren, um so am Ende ein für die Problemstellung geeignetes 3D-Modell zu erhalten.

Besonders wichtig sind Möglichkeiten zur intuitive Verknüpfung von Modellen. Neben den offensichtlichen Tätigkeiten des „Zusammenfügens“ (Vereinigungsmenge) bzw. des „Herausschneidens“ (Differenzmenge) hat sich erwiesen, dass zusätzlich „gemeinsame Anteile von Objekten“ (Schnittmenge) als Grundaktion notwendig ist. Diese drei Operationen reichen aber auch aus, um aus relativ einfachen Basisobjekten, den Primitiven, beliebige komplexe Modelle zu formen.

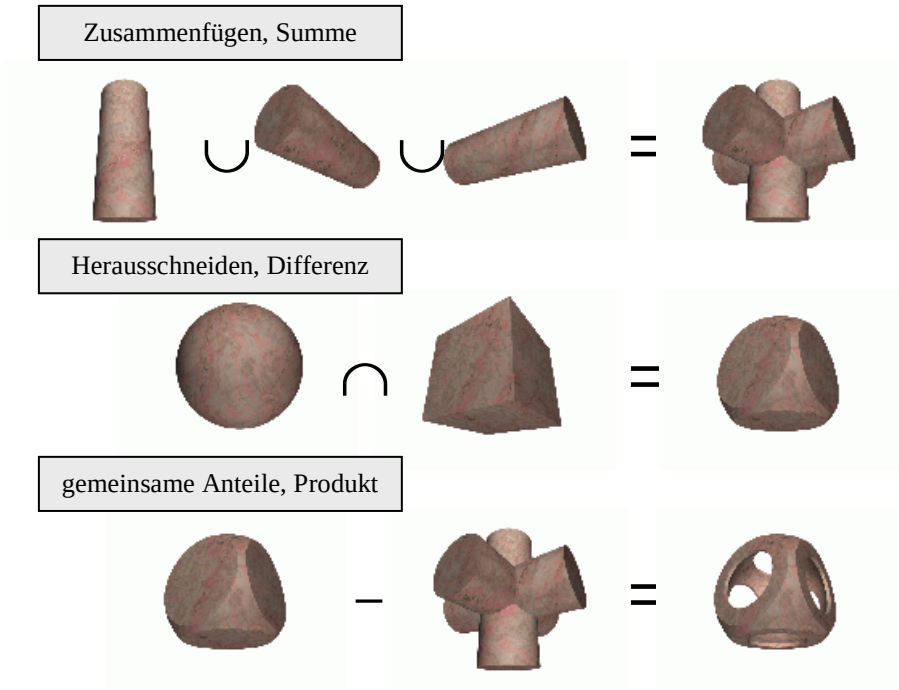

Abbildung 1: CSG-Basisoperationen

Constructive Solid Geometry, abgekürzt als CSG, nutzt genau diesen Ansatz. Ausgehend von dreidimensionalen Primitiven, die solid sind, also über eine geschlossene Hülle verfügen sind, entsteht durch hierarchische Anwendung der drei Grundoperationen Summe (Vereinigungsmenge), Differenz (Differenzmenge) und Produkt (Schnittmenge Anteile) das 3D-Modell.
Diese Ausarbeitung wird zuerst die benutzten Datenstrukturen besprechen. Den Hauptteil bildet jedoch die Darstellung von CSG-Modellen auf dem Bildschirm in Echtzeit, hierbei liegt der Schwerpunkt auf dem bildbasierten Algorithmus von Goldfeather[1], der von Wiegand[2] und später Stewart[3] an die Fähigkeiten und Möglichkeiten aktueller Hardware angepasst wurde.

2 Terminologie und Datenstrukturen

2.1 Der CSG-Baum

Ein *CSG-Ausdruck* besteht aus Bezeichnern für Primitive, die über Mengenoperationen verknüpft werden. In Abbildung 1 wurden für das Produkt jedoch keine Primitive mehr benutzt, sondern man griff auf die Ergebnisse der beiden vorherigen CSG-Ausdrücke zurück. Der voll ausgeschriebene CSG-Ausdruck für diese Differenz lautet daher:

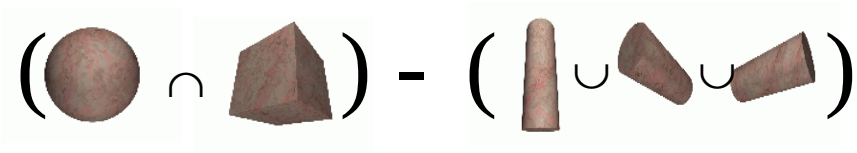

Abbildung 2: Ein einfacher CSG-Ausdruck

Durch die Klammernpaare entsteht eine hierarchische Struktur, welche durch den sogenannten *CSG-Baum* umgesetzt wird. Jedes Blatt entspricht einem Primitiv, jeder Knoten stellt eine CSG-Operation dar, die gegebenfalls um eine affine Transformation ergänzt werden kann. Nach Auswertung aller Teilbäume entsteht an der Wurzel das gewünschte 3D-Modell. In Abbildung 3 sind zum besseren Verständnis an den Knoten und an der Wurzel die Ergebnisse nach Auswertung der dazugehörigen Teilbäume ergänzt worden.

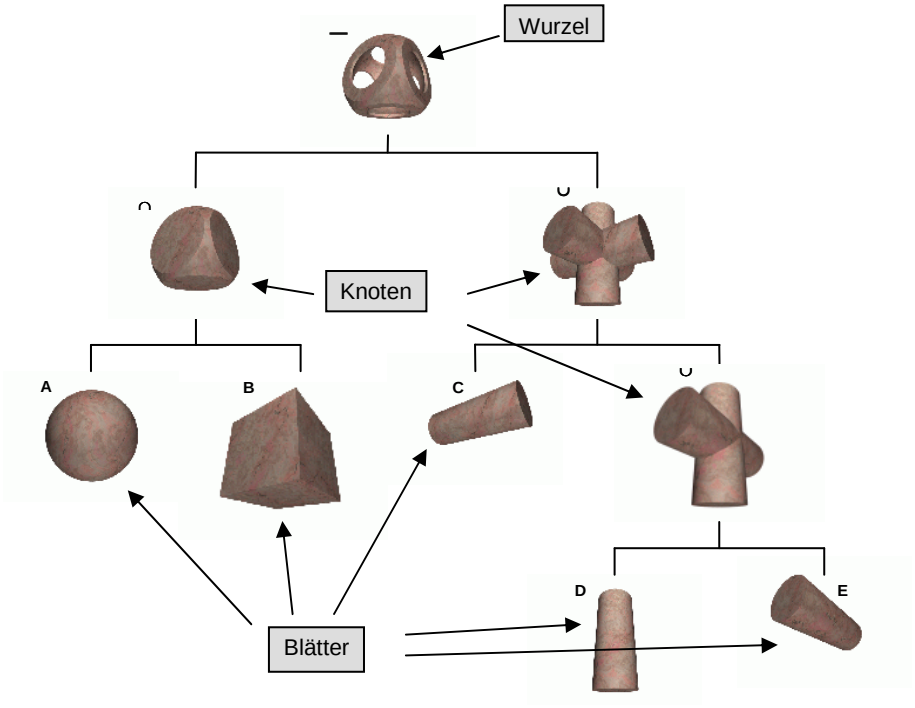

Abbildung 3: CSG-Baum

2.2 Klassifikation

Für jeden Punkt *P* muss eindeutig bestimmbar sein, ob er sich „*innerhalb*“ ($P \in A$) oder „*außerhalb*“ ($P \notin A$) eines beliebigen Primitivs *A* befindet. Durch Auswertung des CSG-Baums ist es möglich, für jeden Punkt auch die Zugehörigkeit bzw. Nicht-Zugehörigkeit zum CSG-Modell zu ermitteln. Dabei gilt für beliebige Teilbäume *X* und *Y*:

1. $P \in (X \cap Y)$ g.d.w. $P \in X \wedge P \in Y$
2. $P \in (X \cup Y)$ g.d.w. $P \in X \vee P \in Y$
3. $P \in (X - Y)$ g.d.w. $P \in X \wedge P \notin Y$

Ein Sonderfall stellen Punkte auf der Oberfläche dar. Sie fallen zwar in die Kategorie „innerhalb“, werden aber zusätzlich als „auf“ attributiert.

3 CSG-Rendering-Techniken im Überblick

3.1 B-rep-Verfahren

Die Darstellung eines CSG-Objekts mit Hilfe der „*boundary representation*“-Technik (*B-rep*) beruht auf der Idee, dass man im Objektraum alle Punkte der Oberfläche ermittelt. Bei vielen Primitivtypen ist dies jedoch schwierig, da eine implizite Formel, die alle Oberflächenpunkte umfasst, schlicht zu komplex ist. Für das eigentlich recht einfach aufgebaute Beispielobjekt ist das Problem bereits nahezu unlösbar.
Als Ausweg zerlegt man alle Primitive in kleinere Bausteine. Diese Tessellation setzt in der Regel auf Dreiecken auf. Sie hat in einer derartigen Genauigkeit zu erfolgen, dass sich keine der erhaltenen Dreiecke schneiden. Anschließend eliminiert man alle Dreiecke, die nicht auf der Oberfläche liegen. Übrig bleibt nur die Klassifikation „auf“, was dann dem gewünschten CSG-Objekt entspricht.

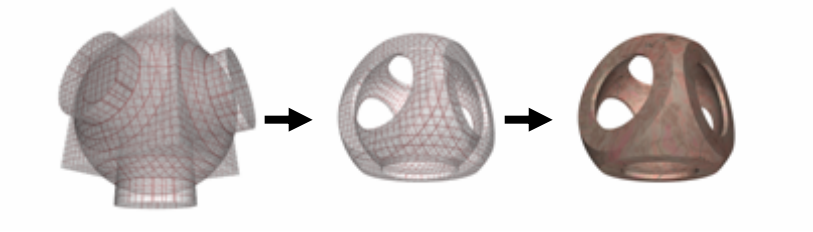

Abbildung 4: Tessellation für B-rep

Dieser objektbasierte Ansatz hat den Vorteil, dass das eigentliche Rendering mit einem beliebigem Verfahren erfolgen kann. Solange sich das zugrunde liegende CSG-Objekt nicht ändert, ist auch keine B-rep-Neuberechnung notwendig.
Ein wesentlicher Nachteil liegt in der Tessellation. Für viele Primitive ist sie nur approximativ durchführbar. Selbst für mathematisch einfache Primitive, wie z.B. Kugeln, existiert kein perfektes Verfahren. Heutzutage wird die Tessellation noch nicht durch Hardware unterstützt, was gerade für Echtzeitanwendungen kritisch ist.

3.2 z-Buffer-Verfahren

Einen gänzlich anderen Ansatz verfolgen die sogenannten *z-Buffer-Verfahren*. Sie setzen explizit auf die Verwendung von vorhandener Grafikhardware. Sie führt die Auswertung des CSG-Ausdruckes bildbasiert mit Hilfe von Tiefeninformationen (z-Buffer) durch, was bedeutet, dass das Rendering in den gesamten Prozess eingebunden ist. In der Regel ist der zugehörige CSG-Baum vorher durch Umformungen zu vereinfachen.
Da diese Verfahren keine Tessellation benutzen, haben sie nicht die Approximations-Probleme, sie sind stattdessen pixelgenau. Zwar führt die Hardware-Unterstützung zu einer erheblichen Beschleunigung, aber ist nun für jedes einzelne Bild die komplette Auswertung des CSG-Ausdruckes notwendig. Je nach Anwendungsgebiet kann daher auch das B-rep-Verfahren schneller sein.

3.3 Andere Verfahren

Neben den beiden Hauptverfahren B-rep und z-Buffer existieren noch andere Ansätze.
Spatial Enumeration ist ein objektbasiertes Verfahren. Man zerlegt den gesamten Raum in geeignete Teilräume, z.B. Würfel. Jeden einzelnen Teilraum klassifiziert man hinsichtlich des CSG-Objekts und entfernt alle, die „außerhalb“ liegen. Zwar entfällt die für B-rep notwendige Tessellation, allerdings ist die Klassifikation eines Teilraums nicht eindeutig durchführbar, da er eventuell auch als „teilweise innerhalb“ und „teilweise außerhalb“ eingestuft werden kann. Je detaillierter man den Raum unterteilt, desto genauer wird das Ergebnis der Klassifikation. Trotz Benutzung von effektiven Datenstrukturen, wie z.B. Octrees, entsteht meist ein sehr hoher Speicherbedarf, was neben den Approximations-Artefakten auch der größte Nachteil von Spatial Enumeration ist.
Ray Casting stellt einen Vertreter der bildbasierten Techniken dar. Für jeden Pixel erzeugt man einen Sichtstrahl und ermittelt alle Schnittpunkte mit den Primitiven. Anschließend wählt man denjenigen Punkt aus, der dem Betrachter am nächsten liegt und die Bedingung „innerhalb“ erfüllt. Es werden keine speziellen Anforderungen, wie das Vorhandensein eines z-Buffers, gestellt. Momentan gibt es jedoch keine Hardware-Unterstützung, weshalb dieses Verfahren auf Anwendungen beschränkt ist, die nicht zeitkritisch sind. Fast alle Raytracer benutzen Ray Casting.
Hybride Verfahren versuchen die Vorteile von objekt- und bildbasierten Techniken zu vereinigen: ein möglichst großer Teil des CSG-Ausdrucks soll mit nur minimalen Approximationen vorberechenbar sein, während im Renderingprozess die Hardware den Rest auswertet. Die Performance hängt von der Balance zwischen diesen beiden Bestandteilen ab. Hier hat sich aber noch kein Vertreter etablieren können.

4 Der Algorithmus von Goldfeather[1]

4.1 Zugrundeliegende Idee

Im Jahre 1989 wurde der sogenannte Goldfeather-Algorithmus[1] veröffentlicht, der zur Klasse der z-Buffer-Verfahren gehört. An der University of North Carolina in Chapel Hill entstand mit dem Pixel-Planes-System eine Hardwareplattform zur Grafikdarstellung. Die Erweiterung Pixel-Powers verfügte über die Möglichkeit der Darstellung von CSG-Objekten, wobei der Goldfeather-Algorithmus zum Einsatz kam.
Er macht sich zunutze, dass allein mit dem Tiefentest bereits einzelne Primitive oder beliebige Summen korrekt darstellbar sind. Als problematisch erweisen sich lediglich Produkte und Differenzen.
Da mengentheoretisch der Zusammenhang $A - B = A \cap \overline{B}$ gilt, ist jede Differenz auch als Produkt mit einem Komplement formulierbar. Man braucht sich also nur noch um eine effiziente Darstellung von Produkten kümmern.
Gerade das Produkt von Teilbäumen ist im Bildraum nicht trivial lösbar. Mittels einer Umformung des CSG-Baums durch eine sogenannten Normalisierung kann man erreichen, dass Produkte nur jeweils zwischen einem Teilbaum und einem Primitiv gebildet werden müssen. Die Normalisierung ist ebenfalls in der Lage, den gesamten Baum in eine „Summe von Produkten“ zu überführen. Sobald alle Produkte bestimmt wurden, kann man die Summe davon problemlos mit dem Tiefentest bilden.
In seiner ursprünglichen Form arbeitet der Goldfeather-Algorithmus nur mit konvexen Primitiven. Eine Erweiterung um konkave Primitive existiert jedoch, sie wird in diesem Kapitel auch beschrieben.

4.2 Normalisierung

Ausgehend von der Wurzel des CSG-Baums wendet man rekursiv folgende 9 Umformungsregeln an:

1.	$X - (Y \cup Z)$	$= (X - Y) - Z$
2.	$X \cap (Y \cup Z)$	$= (X \cap Y) \cup (X \cap Z)$
3.	$X - (Y \cap Z)$	$= (X - Y) \cup (X - Z)$
4.	$X \cap (Y \cap Z)$	$= (X \cap Y) \cap Z$
5.	$X - (Y - Z)$	$= (X - Y) \cup (X \cap Z)$
6.	$X \cap (Y - Z)$	$= (X \cap Y) - Z$
7.	$(X - Y) \cap Z$	$= (X \cap Z) - Y$
8.	$(X \cup Y) - Z$	$= (X - Z) \cup (Y - Z)$
9.	$(X \cup Y) \cap Z$	$= (X \cap Z) \cup (Y \cap Z)$

Tabelle 1: Normalisierungsregeln

Die Regeln umfassen alle assoziativen und distributiven Konstellationen, in denen ein nicht-normalisierter Knoten auftreten kann. Jack Goldfeather zeigte, dass diese Regeln jeden beliebigen CSG-Baum in dessen Normalform überführen.

Für den eingangs gezeigten CSG-Baum trifft an der Wurzel zweimal die Regel 1 zu. Keine andere Regel kann an einem anderen Knoten benutzt werden:

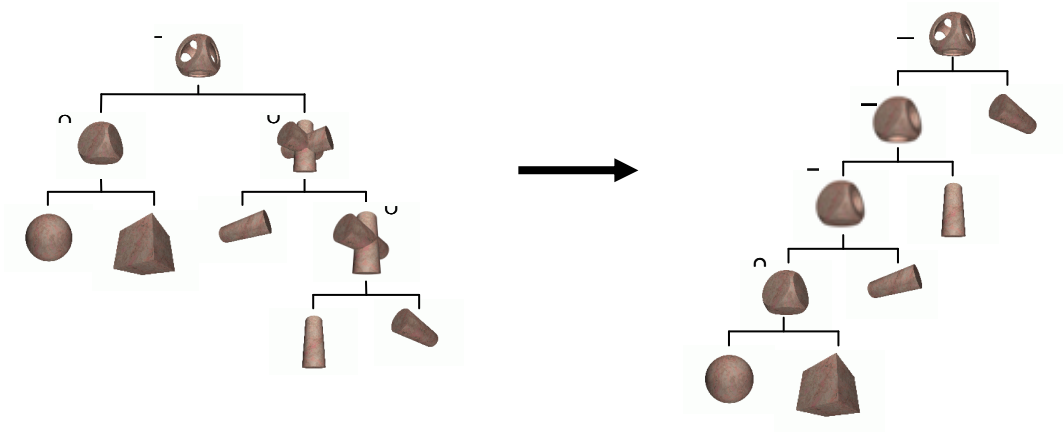

Abbildung 5: Normalisierung eines CSG-Baums

In Pseudocode gestaltet sich die Normalisierung recht einfach:

```
proc normalize(T: tree)
{
	if (T is a primitive)
		return;

	repeat
	{
		while (T matches a rule from 1-9)
			apply first matching rule;

		normalize(T.left);
	}
	until (T.op is a union) or
		((T.right is a primitive) and (T.left is not a union));

	normalize(T.right);
}
```

Die Regeln 2, 3, 5, 8 und 9 vergrößern den CSG-Baum. Im Extremfall führt dies zu einem exponentiellen Wachstum. In den meisten Fällen wirken sich viele neue Produkte aber gar nicht auf das Resultat aus, da sich die beteiligten Primitive nicht schneiden. Eine schnelle Überprüfung auf Schnitt erfolgt über die Bildung von Bounding Boxes. Für jedes Primitiv bildet man eine komplett umschließende Hülle, z.B. einen Würfel, die effizient auf einen Schnitt mit einer anderen Hülle auswertbar ist. Wenn sich die Hüllen zweier Primitive nicht schneiden, dann können sich die Primitive selbst auch nicht schneiden. Zwei einfach anwendbare Regeln zur Ausdünnung (pruning) des CSG-Baumes sind:

1.	$A \cap B = \emptyset$, wenn $bound(A) \cap bound(B) = \emptyset$
2.	$A - B = \emptyset$, wenn $bound(A) \cap bound(B) = \emptyset$

Tabelle 2: Ausdünnung des normalisierten CSG-Baums

Allerdings ist es möglich, dass sich Bounding Boxes durchdringen, obwohl die dazugehörigen Primitive keinerlei gemeinsamen Punkte haben. Dann würde man obige Regeln nicht anwenden, obwohl man es eigentlich dürfte. Hier wäre es angebracht, über effizientere Bounding Boxes nachzudenken. Diese machen aber nur Sinn, wenn man beim Rendering mehr Zeit spart, als man beim Ausdünnen investiert.

4.3 Paritätstest

Per Vereinbarung ist der Betrachter immer „außerhalb“ des CSG-Objekts. Für ein Produkt $Teilbaum \cap Primitiv$ ist es wesentlich zu wissen, ob ein Fragment des Teilbaums „innerhalb“ oder „außerhalb“ des beschneidenden Primitivs liegt. Fragmente des Teilbaums, vor denen laut z-less-Tiefentest nur die Vorderseite des Primitivs liegt, klassifiziert man als „innerhalb“. Fragmente des Teilbaums, vor denen sich, nach z-less-Tiefentest, zusätzlich noch die Rückseite des Primitivs befindet, sind wiederum als „außerhalb“ einzustufen. Das gleiche gilt, wenn weder Vorder- noch Rückseite des Primitivs vor dem Fragment liegen.

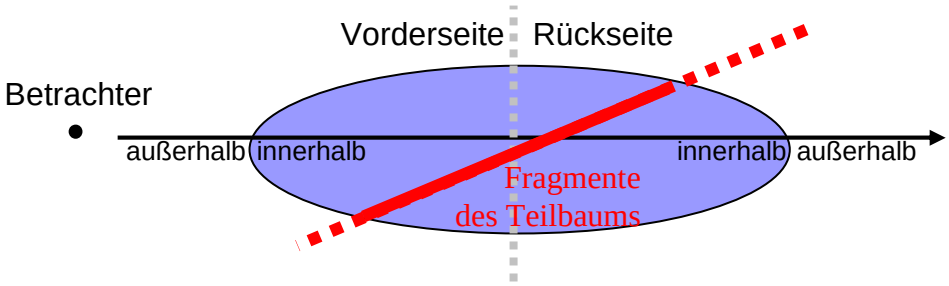

Abbildung 6: Klassifikation mit Paritätstest

Da nur zwei verschiedene Ergebnisse möglich sind – „innerhalb” oder „außerhalb“ – reicht ein einzelnes Bit aus, das man Paritäts-Bit nennt. Typischerweise bedeutet ein gelöschtes Bit „außerhalb“, während ein gesetztes für „innerhalb“ steht.

4.4 Darstellung von Produkten

Produkte der Form $Teilbaum \cap Primitiv$ zeichnet man, indem zuerst alle sichtbaren Fragmente des Teilbaums inkl. deren Tiefenwerte bestimmt werden. Der Paritätstest mit dem Primitiv ist entscheidend, welche Fragment wieder zu löschen sind:

- ist das Primitiv ein Komplement, so entfallen alle Fragmente, für die der Paritätstest „außerhalb“ ergab (Paritäts-Bit gelöscht)
- ist das Primitiv kein Komplement, dann sind alle Fragmente zu entfernen, die laut Paritätstest „innerhalb“ liegen (Paritäts-Bit gesetzt)

In den beiden folgenden Illustrationen soll das Produkt eines Primitivs *X* (auch ein einzelnes Primitiv ist ein Teilbaum) mit einem Primitiv *Y* gebildet werden. Zuerst wird *X* gezeichnet, die Tiefeninformation soll das bläuliche Dreieck verdeutlichen. Die nach dem Paritätstest schließlich gezeichneten Fragmente entsprechen der dicken schwarzen Linie.

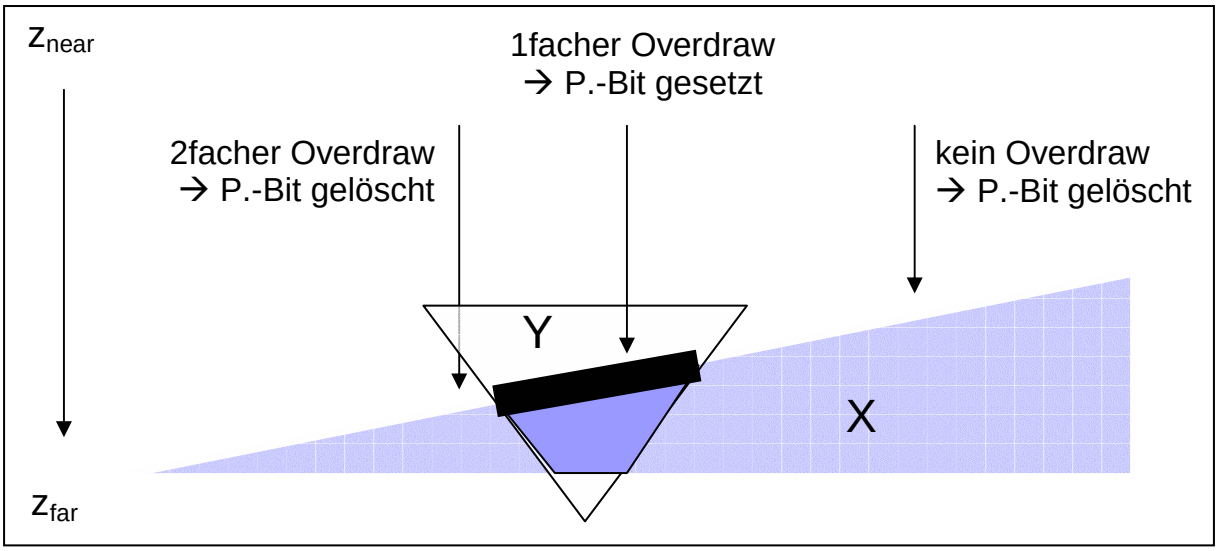

Abbildung 7: Paritätstest für X ∩ Y

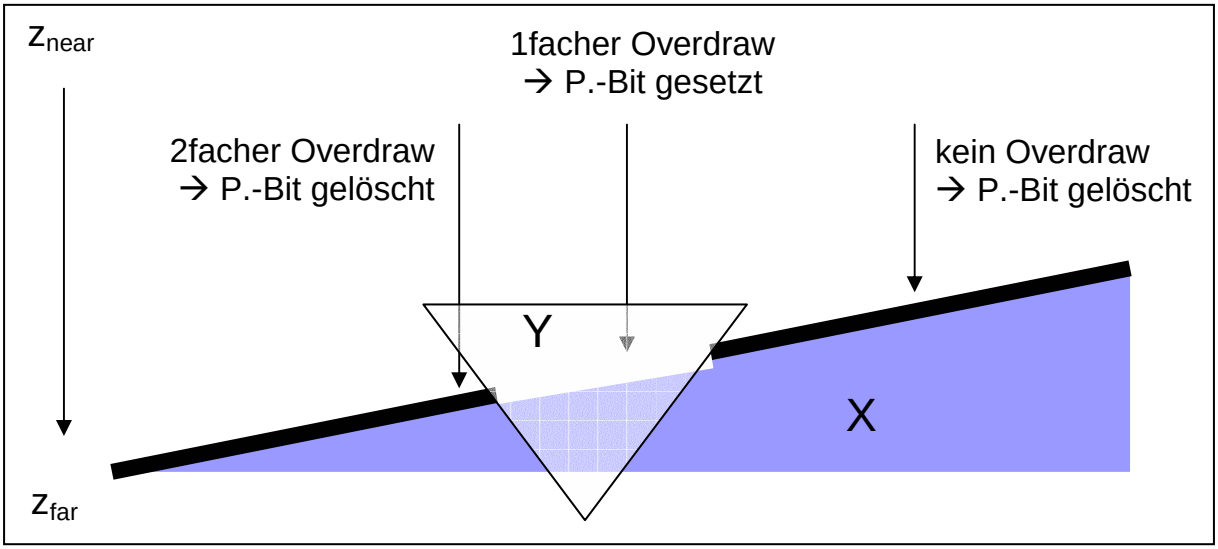

Abbildung 8: Paritätstest für X - Y

Nach $X \cap Y$ ist auch $Y \cap X$ zu zeichnen. Wenn Y kein Komplement ist, dann kann dieser Schritt auch entfallen.
In der Regel umfassen Produkte aber mehr als zwei Primitive. In diesem Fall ist jedes Primitiv zu rasterisieren und ein Paritätstest mit jedem anderen Primitiv des Produkts durchzuführen. Für ein Produkt, das aus 5 Primitiven besteht, sind bereits $5 \cdot 4 = 20$ Paritätstest erforderlich.
Generell ist ein Produkt in einem vom Framebuffer abgetrennten Buffer zu bestimmen, um die bereits berechneten Produkte, die fertig im Framebuffer liegen, nicht zu zerstören. Nachdem die Klassifikation aller Fragmente abgeschlossen ist, kopiert man die tatsächlich sichtbaren unter Beachtung der Tiefenwerte in den Framebuffer.

Der komplette Algorithmus in Pseudocode:

```
for (each product P)
{
	create second depth and color buffer;

	// from now all actions are performed in our second depth and color buffer !
	for (each primitive A in P)
	{
		clear second depth and color buffer;

		if (A is a complement)
			draw back of A;
		else
			draw front of A;

		for (each other primitive B in P)
		{
			do parity test;

			if (B is a complement)
				delete pixels where parity bit is set;
			else
				delete pixels where parity bit is not set;
		}

		copy remaining pixels using z-less to original frame buffer;
	}

	free our second depth and color buffer;
}
```

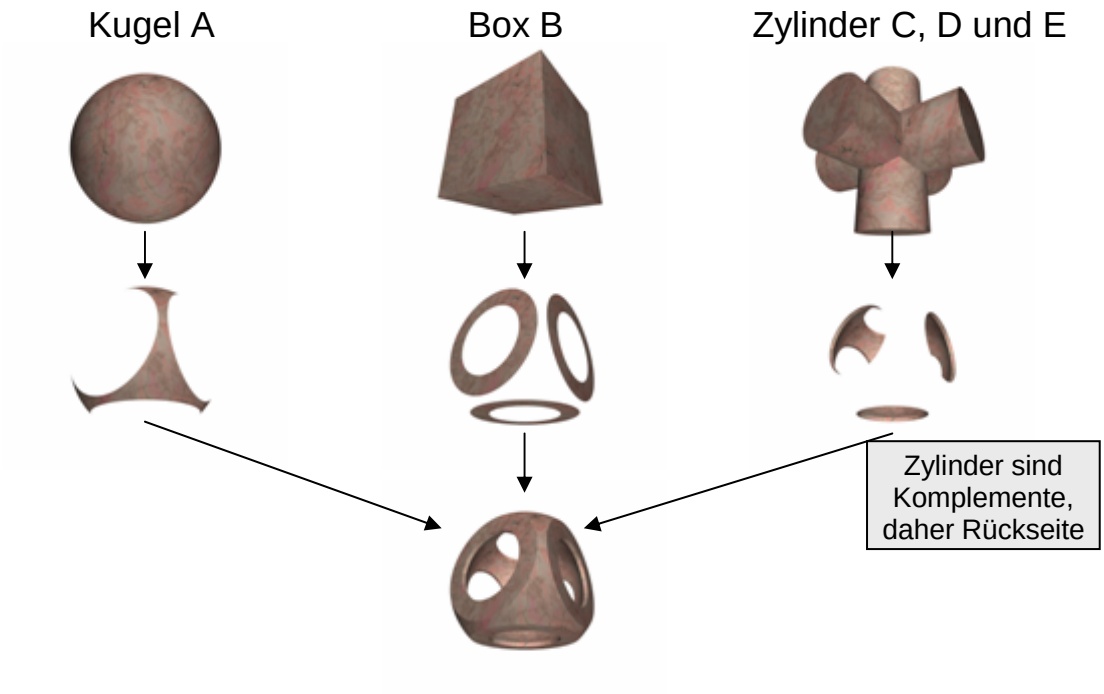

Abbildung 9: Sichtbare Fragment der einzelnen Primitive (C, D und E aus Platzgründen bereits vereinigt)

4.5 Erweiterung auf konkave Primitive

Konkave Primitive zeichnen sich dadurch aus, dass sie mehrere Paare Vorder- und Rückseiten besitzen können. Sei die maximal denkbare Anzahl an Paaren k, so nennt man das Primitiv k-konvex, da man es aus höchstens k konvexen Primitiven zusammensetzen kann. Dabei gilt:

$$konkaves\,Primitiv = Teilprimitiv1 \cup Teilprimitiv2 \cup ... \cup Teilprimitiv\,k$$

Diesen Term kann man dann ohne Probleme in den CSG-Baum einbinden und die bereits bekannten Vorgehensweisen nutzen.

5 Anpassung an OpenGL durch Tim Wiegand[2]

In seiner ursprünglichen Form war der Goldfeather-Algorithmus ganz auf die Pixel-Planes-Architektur ausgerichtet. Bei dieser stehen 2 Color- und 2 z-Buffer bereit, die Paritätsbits sind ebenfalls dort verfügbar.
OpenGL als offener Standard für moderne Grafikhardware bietet in der Regel jedoch nur 1 Color- und 1 z-Buffer. Zusätzlich existiert aber ein frei nutzbarer Stencil-Buffer. Dieser ist meist 8 Bit pro Pixel breit, ein Bit davon würde bereits für das Paritätsbit ausreichen. Zur Verbesserung der Effizienz ändert die Erweiterung von Wiegand den Goldfeather-Algorithmus ab und nutzt möglichst alle Stencil-Buffer-Bits.
Man klassifiziert alle Fragmente eines Primitivs, wobei ein Bit des Stencil-Buffers als Paritätsbit dient. Anschließend setzt man genau dort über eine OR-Verknüpfung eine Bitmaske in den Stencil-Buffer, wo das jeweilige Pixel auch tatsächlich sichtbar ist.
Damit ist man in der Lage, im Regelfall bis zu 7 Primitive eines Produkte nacheinander zu bearbeiten, bevor man in den Color-Buffer schreibt. Dies wiederum geschieht über ein selektives Neuzeichnen aller bearbeiteten Primitiven an den Stellen, wo die zugehörige Stencil-Buffer-Bitmaske und z-less-Tiefentest gesetzt ist.
Dadurch entfällt die Notwendigkeit eines zweiten Color-Buffers, ebenso braucht man nicht mehr den z-Buffer kopieren, was anfällig gegenüber Konvertierungsfehlern ist.
Jede OpenGL-Implementation, auch eine reine Softwarelösung, ist in der Lage, das Verfahren von Wiegand zur Darstellung von CSG-Objekten zu nutzen.

6 Layered Goldfeather

6.1 Funktionsweise

Stewart, Leach und John[3] beobachteten, dass in einem Produkt jedes Primitiv gegen jedes andere mit Hilfe des Paritätstests überprüft wird. Weiterhin stellten sie fest, dass sich ein großer Teil der Primitive eines Produkts sich gar nicht überlappt, der Paritätstest in vielen Fällen also überflüssig ist.
Die Anzahl Primitive, die ein Pixel überdecken, gibt die maximale Tiefenkomplexität *k* einer Szene an. Selbst für viele Primitive ist sie, abhängig vom Sichtwinkel, oft recht klein. In der folgenden Illustration liegt die maximale Tiefenkomplexität *k* zwischen 2 und 6. Die Szene selbst besteht aus 26 Primitiven.

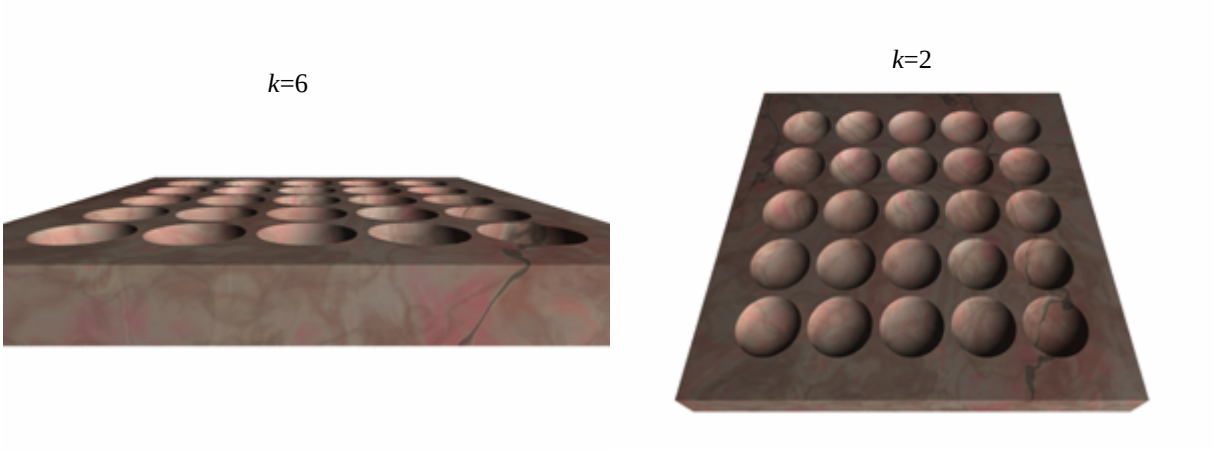

Abbildung 10: Tiefenkomplexität

Das Wissen um die Tiefenkomplexität nutzt man, indem nur noch jeweils die Primitive miteinander verglichen werden, die das gleiche Pixel überdecken. Dazu unterteilt man das Bild in Schichten. In jeder Schicht sind Pixel mit gleicher Tiefenkomplexität gruppiert. Für jede Schicht, statt wie vorher für jedes Primitiv, führt man anschließend den Paritätstest durch und kopiert alle übriggebliebenen Pixel in den Framebuffer.

Jetzt folgt zunächst der Pseudocode des Layered-Goldfeather-Verfahrens, im Anschluss daran werden die einzelnen Bestandteile genauer geklärt:

```
for (each product P)
{
      create second depth and color buffer;

      // from now all actions are performed in our second depth buffer !
      for (each layer of P)
      {
            clear second depth and color buffer;
```

(nächste Seite ...)

```
            for (each primitive A in P)
            {
                  if (A is a complement)
                        draw back of A, only pixels of current layer;
                  else
                        draw front of A, only pixels of current layer;
            }

            for (each primitive A in P)
            {
                  do parity test;

                  if (A is a complement)
                        delete pixels where parity bit is set;
                  else
                        delete pixels where parity bit is not set;
            }

            copy remaining pixels using z-less to original frame buffer;
      }

      free our second depth and color buffer;
}
```

6.2 Bestimmung der maximalen Tiefenkomplexität k

Bisher wurde nicht erwähnt, wie man überhaupt die maximale Tiefenkomplexität k ermittelt. Für diesen Schritt nutzt man den gelöschten Stencil-Buffer. Wenn man sowohl Color- als auch z-Buffer sperrt und als Stencil-Operation ein Inkrement von 1 setzt, dann erhält nach Zeichnen aller Primitive (ohne Beachtung von CSG) die Tiefenkomplexität der Pixel. Der maximale aller Stencil-Werte ist dann das gesuchte *k*.
Ein kleines Problem besteht jedoch darin, dass jedes konvexe Primitiv über Vorder- und Rückseite verfügt. Daher sind die Werte im Stencil-Buffer um den Faktor 2 zu groß. Das Culling einer Seite beseitigt auch diese Hürde. Aufgrund der begrenzten Bitbreite des Stencil-Buffers ist man im Regelfall auf 2^7=128 Schichten begrenzt, da von den 8 Bit des Stencil-Buffers im Extraktionsprozess eines für das Paritätsbit reserviert bleiben muss.

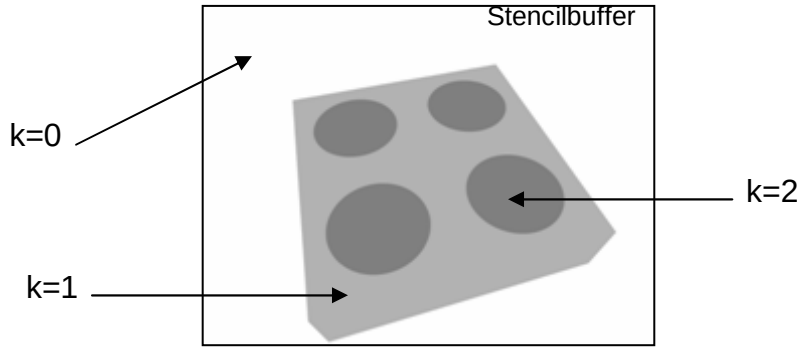

Abbildung 11: Bestimmung der Tiefenkomplexität

Genau genommen benötigt man aber nicht die Tiefenkomplexität einer kompletten Szene, sondern nur die der einzelnen Produkte, da die Summe aller Produkte allein mit dem Tiefentest korrekt evaluiert wird. Alle folgenden Schritte beziehen sich daher auf jeweils ein einzelnes Produkt.

6.3 Extraktion der Schichten

Die Schichten einer Szene im Bildraum extrahiert man, indem Color-, z- und Stencil-Buffer gelöscht und alle Primitive in einer konstanten Reihenfolge *k*-mal gezeichnet werden. Die Stencil-Operation bleibt unverändert, jedoch kommt diesmal noch eine Stencil-Funktion hinzu, die dafür sorgt, dass im *n*-ten Durchlauf nur die Fragmente mit dem Paritätstest bearbeitet werden, die genau den Stencil-Wert *n* aufweisen.

Abbildung 12: Extraktion der Schichten

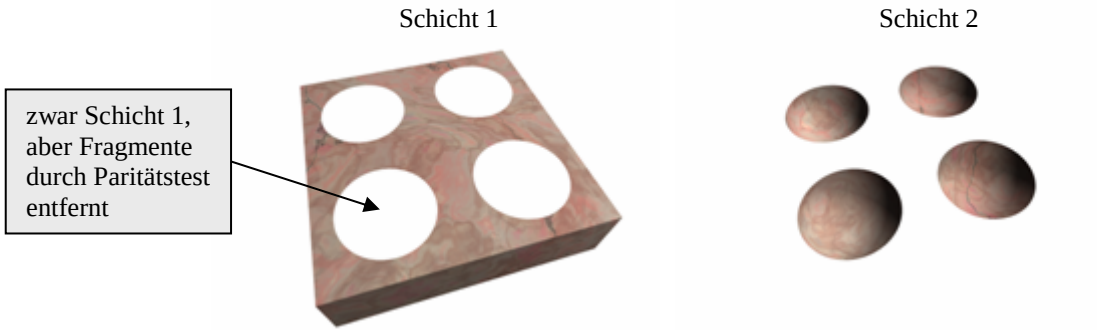

Abbildung 13: Sichtbare Fragmente der Schichten

6.4 Paritätstest einer Schicht

Eine komplette Schicht wird gegen alle Primitive des Produkts mit Hilfe des Paritätstests geprüft. Das bedeutet aber auch, dass man die Fragmente eines Primitivs gegen das Primitiv selber testet. Als Resultat würde sich jedes Primitiv selber wieder entfernen. Weil dieses Verhalten unerwünscht ist, muss man den Paritätstest dahingehend abändern, dass explizit Primitive, die den gleichen Tiefenwert aufweisen, in den Test einbezogen werden.

7 Vergleich

Der originale Goldfeather-Algorithmus weist eine zur Anzahl der Primitive *n* quadratische Anzahl Paritätstests n^2 auf. Man kann mit der Layered-Goldfeather-Technik diese Komplexität reduzieren, da die Paritätstests dann nur noch von der Anzahl Schichten *s* und den Primitiven *n* abhängen. Sie liegt in der Größenordnung $s \cdot n$, wobei als Schätzung $s = log\ n$ gilt, d.h. die Gesamtzahl an Paritätstests ist in etwa $n \cdot log\ n$. Allerdings erkauft man sich diese Reduktion der Tests mit einem mehrfachen Zeichnen aller Primitive. In der Folge ist das neue Verfahren nur dann schneller, wenn deutlich weniger Schichten *s* als Primitive *n* in der Szene auftauchen.

8 Ausblick

Bisher erfolgte kein Clipping des CSG-Baums am Sichtvolumen. Wenn man feststellt, dass sich die Bounding Box eines Teilbaums außerhalb des Sichtbereichs befindet, dann kann man komplett auf dessen Berechnung verzichten.
Gerade die stärkere Ausdünnung des normalisierten CSG-Baums ist erstrebenswert. In dieser Richtung existieren verschiedene Ansätze, die sich verschiedene Eigenschaften von Bounding Boxes zunutze machen.
Die Normalisierung selbst ist ein aufwändiger Bestandteil der Algorithmen. Sie ist nur erforderlich, wenn sich der originale CSG-Ausdruck ändert. In der Praxis betreffen Änderungen oft nur kleine Teile des CSG-Baums, so dass man sich mit Caching viele Regelüberprüfungen ersparen kann.

9 Quellen

[1] Jack Goldfeather, Steven Molnar, Greg Turk und Henry Fuchs: „*Near Real-Time CSG Rendering Using Tree Normalization and Geometric Pruning*“, IEEE Computer Graphics and Applications, Ausgabe 9(3), Seiten 20-28, 1989

[2] Tim F. Wiegand: „*Interactive Rendering of CSG models*“, Computer Graphics Forum, Ausgabe 14(4), Seiten 249-261, 1996

[3] Nigel Stewart, Geoff Leach und Sabu John: „*An improved Z-buffer CSG rendering algorithm*“, Proceedings of the Eurographics 98, Seiten 25-30, ACM Press, 1998

[4] http://www.nigels.com/research/

Mehrere Illustrationen entstanden nach Vorlage von [3] und [4].