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:

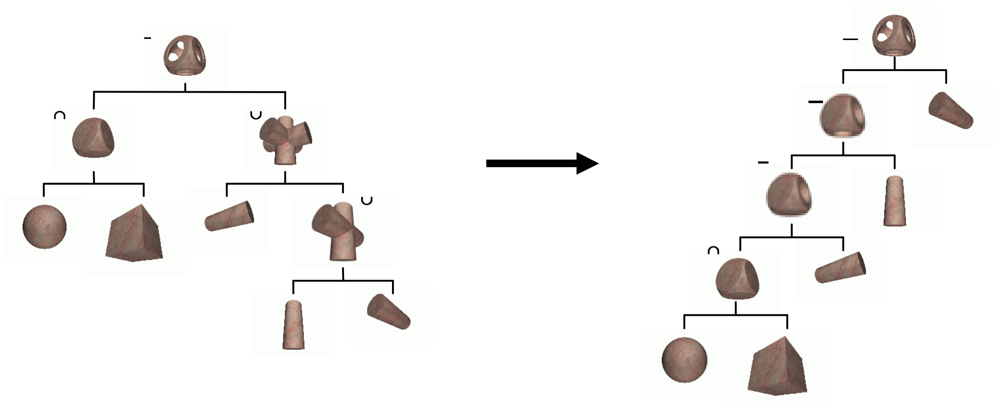

Abbildung 5: Normalisierung eines CSG-Baums

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 $\text{bound}(A) \cap \text{bound}(B) = \emptyset$
2. $A - B = \emptyset$, wenn $\text{bound}(A) \cap \text{bound}(B) = \emptyset$

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.

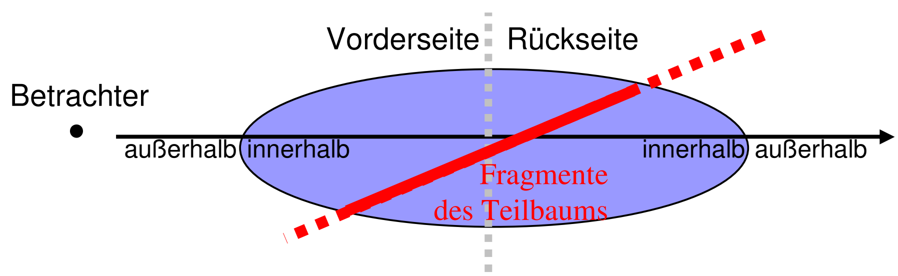

Abbildung 6: Klassifikation mit Paritätstest

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.

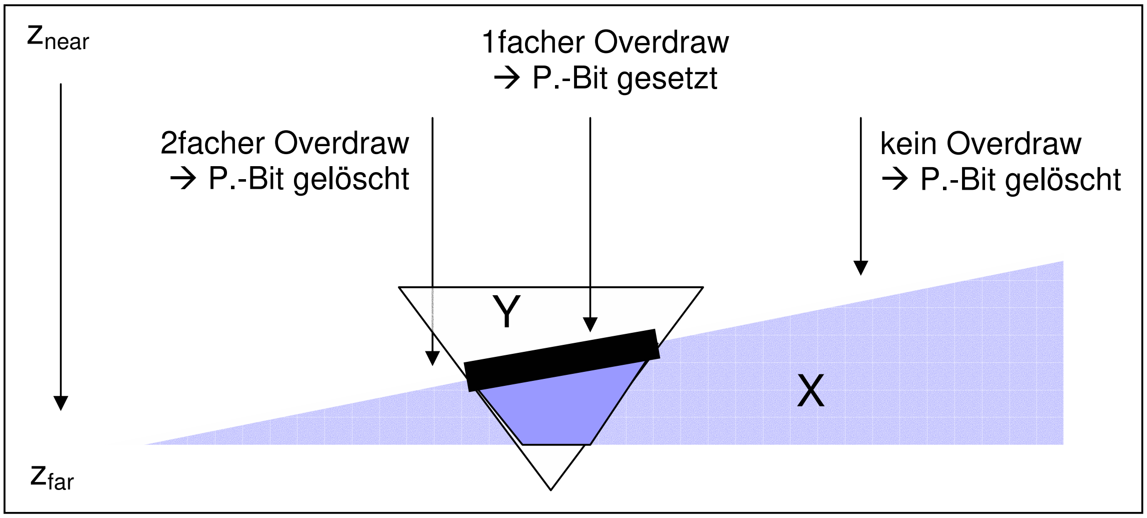

Abbildung 7: Paritätstest für X ∩ Y

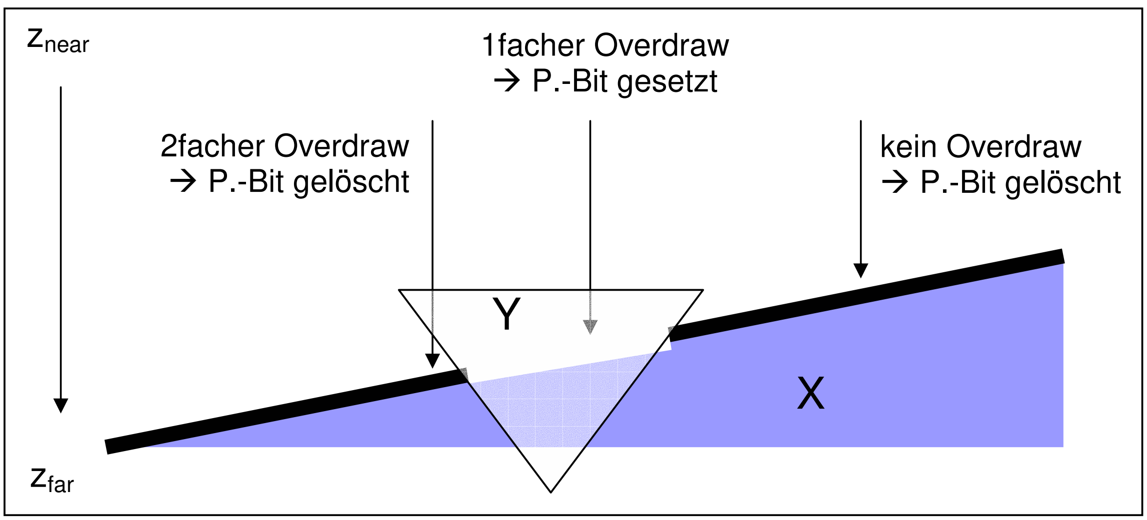

Abbildung 8: Paritätstest für X - Y

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;
}
```

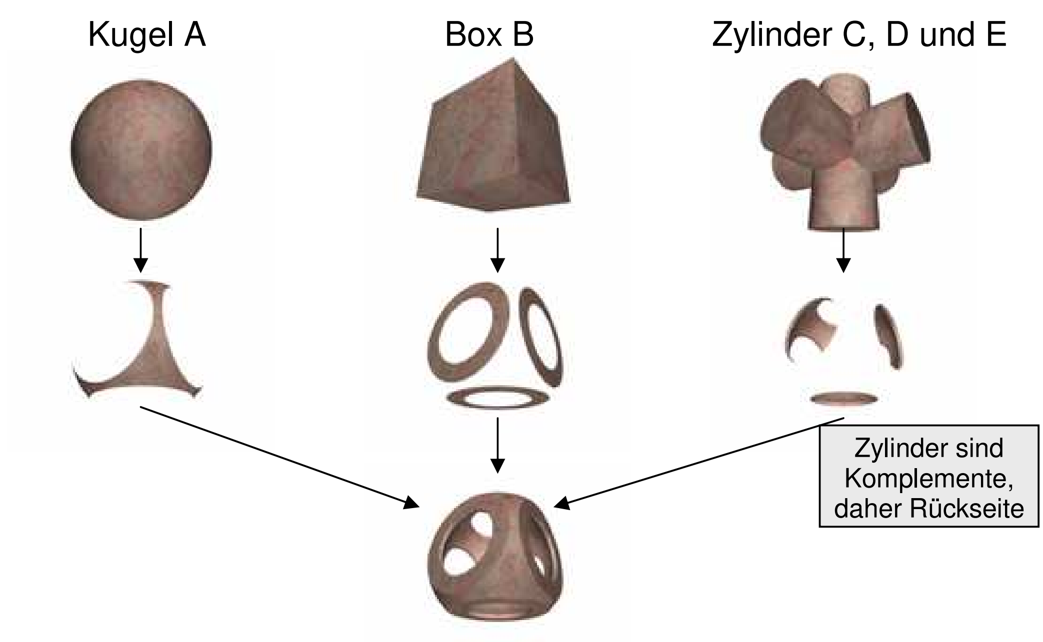

Abbildung 9: Sichtbare Fragment der einzelnen Primitive (C, D und E aus Platzgründen bereits vereinigt)

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:

$$konkaves\ Primitiv = Teilprimitiv\,1 \cup Teilprimitiv\,2 \cup ... \cup Teilprimitiv\,k$$

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.