TECHNISCHE UNIVERSITÄT DRESDEN

FAKULTÄT INFORMATIK
INSTITUT FÜR COMPUTERGRAFIK
UND VISUALISIERUNG
PD DR. S. GUMHOLD

Hauptseminar
"Graphische Datenverarbeitung"

Blister: GPU-based rendering of Boolean combinations of free-form triangulated shapes

Tobias Hilbrich
(Mat.-Nr.: 3020348)

Betreuer: Dr.-Ing. Wilfried Mascolus

Dresden, 24. Januar 2006

Inhaltsverzeichnis

1 Inhalt **1**

2 Einführung in CSG Ausdrücke **1**
2.1 Grundlagen 1
2.2 Darstellung 1
2.3 Anwendung 2

3 Bisherige Ansätze **2**
3.1 Überblick 2
3.2 Berechnung der Oberfläche 3
3.3 Darstellung ohne Oberflächenberechnung 3

4 Wichtige Regeln für CSG Ausdrücke **3**
4.1 Oberfläche 3
4.2 Klassifizierung 4
4.3 Operationen und Wahrheitswerte 4

5 Blister **5**
5.1 Grundidee 5
5.2 Der *DepthPeeling* Ansatz 5
5.3 Klassifikation bei Blister 7
5.3.1 Ein Klassifikationsbeispiel 7
5.3.2 Match & Flip 9
5.3.3 Minimale ID‘s 13
5.3.4 Resultat der Vereinfachungen 14
5.3.5 GPU Klassifikation 14
5.4 Zusammenführung der Schichten 16
5.5 Mögliche Erweiterungen und Optimierungen 17

6 Zusammenfassung und Ausblick **18**

Literatur **19**

A DepthPeel Pixelshader für DirectX **21**

B Code fragment für die Umsetzung der Stenciloperationen unter DirectX **22**

1 Inhalt

In diesem Dokument wird eine Technik beschrieben die es erlaubt CSG Ausdrücke in Echtzeit darzustellen und auszuwerten. Dabei wird nicht die eigentliche (triangulierte) Objekthülle berechnet. Dadurch ist es möglich auch dynamische CSG Ausdrücke auszuwerten. Die Technik wurde von John Hable und Jarek Rossignav entwickelt und in [JH05] vorgestellt. Zur Umsetzung dieser Ideen wird die *Depth Peeling* Technik und eine Vielzahl von *Stencilbuffer*-Operationen genutzt. Des Weiteren wird eine spezielle Normalform verwendet - die sogenannte *Blist-Normalform*. Der zugehörige *Renderer* wurde *Blister* genannt.
Zusätzlich wird eine eigene vereinfachte Blister Implementierung vorgestellt die auf DirectX[1] basiert.

2 Einführung in CSG Ausdrücke

2.1 Grundlagen

Der Begriff CSG steht für *Constructive Solid Geometry*. Dabei handelt es sich um geschlossene 3D-Objekte. Diese Objekte werden Primitive genannt und auf ihnen sind die mathematischen Operationen Vereinigung($\cup$), Schnitt($\cap$), Differenz($-$) und Negation($\neg$) definiert. Um die verwendeten Primitive auf Grafikkarten darstellen zu können, wird für Blister vorrausgesetzt, dass Primitive trianguliert sind. Ansonsten werden keine besonderen Anforderungen an die Primitive gestellt d.h. diese können beliebig komplex[2] sowie auch konkav sein.
Um dies zu veranschaulichen, soll hier ein Beispiel gegeben werden. Die Abbildungen 1 und 2 zeigen zwei beispielhafte Primitive. Dabei soll Abbildung 1 als Primitiv A und Abbildung 2 als Primitiv B bezeichnet werden.

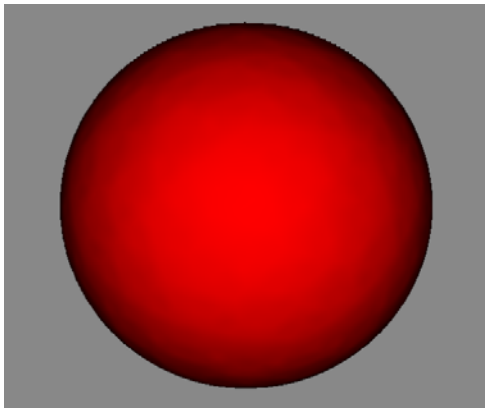

Abbildung 1: Beispielprimitiv A

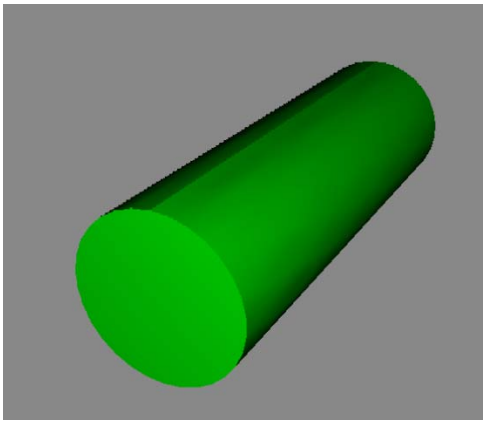

Abbildung 2: Beispielprimitiv B

Nun sollen die Operationen Vereiningung($\cup$), Schnitt($\cap$) und Differenz($-$) auf die Primitive A und B angewandt werden. Abbildung 3 zeigt den CSG Ausdruck $Y = A \cup B$, Abbildung 4 den CSG Ausdruck $Y = A \cap B$ und Abbildung 5 zeigt den CSG Ausdruck $Y = A - B$.

2.2 Darstellung

CSG Ausdrücke werden meist etwas verkürzt aufgeschrieben. Dabei werden die Schnittoperationen($\cap$) nicht explizit aufgeschrieben und ein Vorrang dieser Operationen angenommen. Des Weiteren werden

[1] eingetragenes Produkt von Microsoft: http://www.microsoft.com
[2] müssen aber auf aktuellen Grafikkarten darstellbar sein

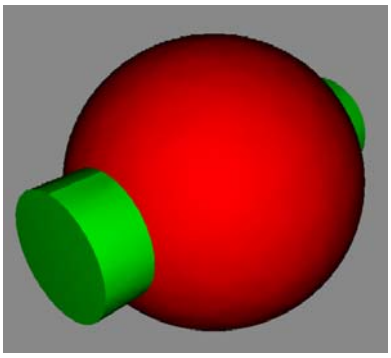

Abbildung 3: $A \cup B$

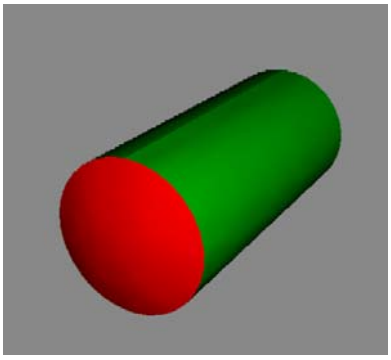

Abbildung 4: $A \cap B$

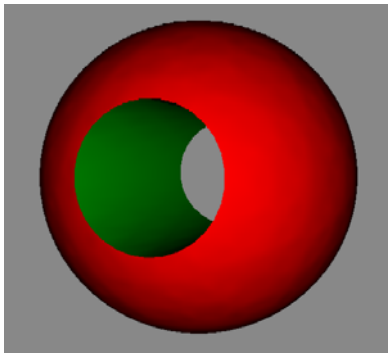

Abbildung 5: $A - B$

Klammern zwischen Vereinigungs-/Schnittoperationen von mehr als 2 Primitiven weggelassen.
Somit würde der CSG Ausdruck $Y = ((((A \cap B) \cap C) \cup D) - ((E \cup F) \cap G))$ vereinfacht als $Y = (ABC \cup D) - (E \cup F)G)$ geschrieben werden.
Um Algorithmen auf CSG Ausdrücke anwenden zu können, werden diese oft als (binäre) Bäume dargestellt. Dabei enthalten die Blätter die einzelnen Primitive und die Knoten die einzelnen Operationen.
In Abbildung 6 ist ein solcher Baum für das obige Beispiel dargestellt.

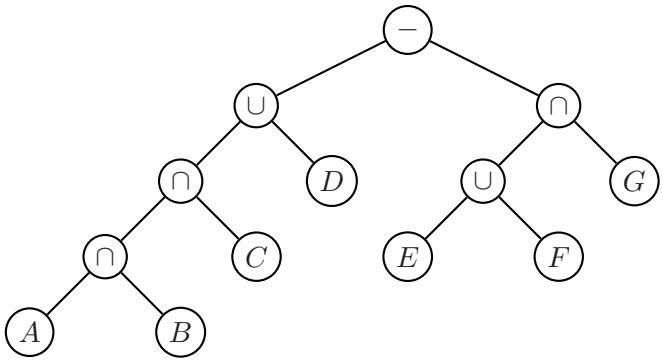

Abbildung 6: Baum zum CSG Ausdruck $Y = (ABC \cup D) - (E \cup F)G)$

2.3 Anwendung

CSG Ausdrücke werden besonders im CAD-Bereich eingesetzt. Konkrete Anwendungen sind zum Beispiel der Entwurf von Werkstücken oder die CNC[3] Simulation. Ein weiterer Anwendungsbereich sind Programme zur Modellierung 3-Dimensionaler Objekte wie Maya oder 3D-Studio.

3 Bisherige Ansätze

3.1 Überblick

Es gibt zwei prinzipielle Ansätze zur Darstellung von CSG Ausdrücken. Der erste Ansatz berechnet die Hülle desjenigen Objekts das aus einem CSG Ausdruck entsteht (*CSG Objekt*). Der zweite Ansatz

[3] programmierte Bearbeitung von Werkstücken besonders verbreitet sind CNC-Fräsen und CNC-Drehen.

verfolgt eine blickwinkelabhängige Darstellung des Objektes ohne vorher die Hülle des CSG Objekts berechnen zu müssen.

3.2 Berechnung der Oberfläche

Mithilfe von Schnittoperationen von Dreiecken oder der Tesselierung kann die Hülle eines CSG Objekts berechnet werden. Ein derartiger Ansatz wurde zum Beispiel in [RA85] vorgestellt. Die Berechnung der Hülle ist eine sehr komplexe Operation die für Primitive mit vielen Dreiecken und CSG Ausdrücke mit vielen Primitiven nicht in Echtzeit ausgeführt werden kann. Nachdem aber einmal die Hülle berechnet ist, kann das resultierende Objekt wie ein normales 3D Objekt ohne spezielle Techniken dargestellt werden. Somit ist diese Art der Darstellung günstig für statische CSG Ausdrücke. Bei Veränderungen der Positionen von Primitiven oder des CSG Ausdrucks selbst ist dieser Ansatz allerdings meist zu langsam um eine Echtzeitdarstellung zu ermöglichen. Des Weiteren kann bei diesem Ansatz meist die hohe Rechenleistung der Grafikkarte nicht ausgenutzt werden.

3.3 Darstellung ohne Oberflächenberechnung

Bei diesem Ansatz wird der CSG Ausdruck für jedes dargestellte Bild ausgewertet. Dabei werden Operationen auf Pixelebene durchgeführt, um festzustellen ob ein einzelner Pixel innerhalb des Bereiches liegt der von einem CSG Ausdruck beschrieben wird. Um diesen Ansatz umzusetzen, werden meist besondere Normalformen verwendet. Eine dieser Normalformen ist die *Disjunktive Normalform* welche in [GJ86] vorgestellt wurde. Diese ermöglicht es mithilfe eines Paritätstests festzustellen, ob ein Pixel innerhalb eines Produkts von Primitiven liegt. Dieser Algorithmus konnte zunächst nur auf konvexe Primitive angewandt werden und wurde später so erweitert das er auch auf konkave Primitive anwendbar ist. Der Nachteil dieser Normalform ist, dass es spezielle Fälle geben kann bei der die Anzahl der einzelnen Produkte von Primitiven exponentiell wächst. Da die Komplexität der Algorithmik von der Anzahl der Primitive abhängt, ist diese in solchen Fällen nur begrenzt einsetzbar. Der hier vorgestellte Ansatz nutzt die Blist-Normalform, welche kein exponentielles Wachstum aufweist.
Diese Gruppe von Ansätzen kann auf dynamische CSG Ausdrücke angewandt werden. Dafür sind aber spezielle Darstellungsverfahren nötig die langsamer sind als die Darstellung einer vorberechneten Hülle. Meist ist es möglich diese Verfahren auf Grafikkarten auszuführen um deren enormes Rechenpotential auszunutzen.

4 Wichtige Regeln für CSG Ausdrücke

Im folgenden sollen Regeln und Gesetze vorgestellt werden die bei CSG Ausdrücken angewandt werden können. Dabei werden hier speziell die Regeln vorgestellt die zum Blister Ansatz führen. Es gibt neben den folgenden Regeln auch noch weitere die aber für den Blister Ansatz nicht von Bedeutung sind.
Dabei wird der Begriff *CSG Objekt* für das aus einen CSG Ausdruck resultierende (3D) Objekt genutzt.

4.1 Oberfläche

Da beim Blister-Ansatz nicht die (triangulierte) Oberfläche eines CSG Objekts berechnet werden soll, ist es wichtig herauszufinden wie sich die Oberfläche eines CSG Objektes zusammensetzt und welche Eigenschaften diese besitzt. Dies wird vom folgenden Satz beschrieben:

> Die Oberfläche eines CSG Objekts besteht aus den Oberflächenteilen der Primitive die an dem CSG Ausdruck beteiligt sind.

Dieser Satz wurde in [Til84] vorgestellt und hat wichtige Konsequenzen. Das erste Resultat ist, dass es genügt alle Oberflächenteile[4] der einzelnen Primitive zu betrachten. Allerdings gehören diese nicht zwingend zum CSG Objekt. Somit müssen die richtigen Oberflächenteile ausgewählt werden. Dieses Auswählen nennt man klassifizieren. Beim klassifizieren wird überprüft ob ein Oberflächenteil den CSG Ausdruck erfüllt oder nicht. Also ob es in dem Bereich liegt der durch einen CSG Ausdruck beschrieben wird.

4.2 Klassifizierung

Da es sich bei den beteiligten Primitiven um *Solide*(also ausgefüllte) Objekte handelt, beschreibt jedes dieser Objekte eine Teilmenge des $\mathbb{R}^3$. Ein CSG Ausdruck führt auf diesen Teilmengen die Mengenoperationen Schnitt, Vereinigung und Differenz aus. Somit beschreibt ein CSG Ausdruck wieder eine Teilmenge des $\mathbb{R}^3$. Für einen CSG Ausdruck Y soll diese Teilmenge als $span(Y)$ bezeichnet werden. Damit gehören alle Oberflächenteile zum CSG Objekt die eine Teilmenge von $span(Y)$ sind.
Bei der Darstellung auf Bildschirmen wird pixelweise gezeichnet. Somit ist die Oberfläche eines Primitivs eine Menge von Pixeln. Um diese Pixel zu klassifizieren muss geprüft werden, ob die von den Pixeln beschriebenen Punkte in $span(Y)$ liegen oder nicht. Die Oberfläche des CSG Objekts besteht dann aus den Pixeln, aller Oberflächenteile der beteiligten Primitive, deren Positionen in $span(Y)$ liegen.

4.3 Operationen und Wahrheitswerte

Die oben beschriebene Variante der Klassifizierung verlangt es $span(Y)$ zu berechnen. Diese Menge ist aber im Allgemeinen nicht endlich. Somit muss die Berechnung von $span(Y)$ vermieden werden. Da die Primitive als triangulierte Objekte vorliegen, ist es mithilfe sogenannter *Stenciloperationen* möglich zu bestimmen, ob ein Pixel innerhalb eines Primitivs liegt oder nicht. Aufbauend darauf kann iterativ bestimmt werden ob ein Pixel in $span(Y)$ liegt. Beim Testen ob ein Pixel innerhalb eines Primitives liegt, bezeichnet das Resultat *IN* das der Pixel innerhalb des Primitivs liegt und *OUT* das der Pixel außerhalb des Primitivs liegt. Es ist auch ein Vergleich mit Wahrheitswerten möglich, dabei steht *IN* für *T*(*True*) und *OUT* für *F*(*False*). Zusätzlich zu den bisherigen Operationen (Schnitt, Vereinigung, Differenz) soll hier auch die Negation($\neg$) formal eingeführt werden.
Für die einzelnen Operationen gelten folgende Regeln:

Sei X $\in \{IN, OUT\}$ *dann gilt:*

$$\begin{aligned}
(IN \cup X) &= IN & (IN \cap X) &= X & \neg IN &= OUT \\
(X \cup IN) &= IN & (X \cap IN) &= X & \neg OUT &= IN \\
(OUT \cup X) &= X & (OUT \cap X) &= OUT & & \\
(X \cup OUT) &= X & (X \cap OUT) &= OUT & &
\end{aligned}$$

Zusätzlich gelten noch semantische Äquivalenzen die es erlauben CSG Ausdrücke umzuformen ohne deren Bedeutung zu verändern:

Seien Y,Z beliebige CSG Ausdrücke dann gilt:

$$\begin{aligned}
\neg(Y \cup Z) &= (\neg Y \cap \neg Z) \\
\neg(Y \cap Z) &= (\neg Y \cup \neg Z) \\
(Y - Z) &= (Y \cap \neg Z)
\end{aligned}$$

[4] engl. *surfels*

Diese Regeln ermöglichen es einen Algorithmus anzugeben der berechnet ob ein Pixel in $span(Y)$ liegt:

Sei Y ein CSG Ausdruck und p ein Pixel der Klassifiziert werden soll, dann rufe die rekursive Funktion klass(Y,p) auf.
Diese ist wie folgt definiert:

$$klass(Y,p) = \begin{cases} klass(Z1,p) \cup klass(Z2,p) & \text{wenn Y von der Form } (Z1 \cup Z2) \text{ ist.} \\ klass(Z1,p) \cap klass(Z2,p) & \text{wenn Y von der Form } (Z1 \cap Z2) \text{ ist.} \\ klass(Z1,p) \cap \neg klass(Z2,p) & \text{wenn Y von der Form } (Z1 - Z2) \text{ ist.} \\ IN & \text{wenn Y ein Primitiv ist und p in Y liegt.} \\ OUT & \text{wenn Y ein Primitiv ist und p nicht in Y liegt.} \end{cases}$$

Dieser Algorithmus ist zwar sehr einfach aber er ist der grundlegenden Ansatz für Blister. Die schwierige Aufgabe dabei ist diesen Algorithmus so zu modifizieren, dass er auch auf einer Grafikkarte ausgeführt werden kann.

5 Blister

5.1 Grundidee

Um einen gegebenen CSG Ausdruck *Y* darzustellen, werden zuerst alle Oberflächenteile der an *Y* beteiligten Primitive gesammelt und gespeichert. Diese werden dann Pixel für Pixel gegen Y klassifiziert. Anschließend werden alle Pixel die den Test bestanden haben auf das finale Bild kopiert. Dabei muss beachtet werden das die Tiefenreihenfolge der Pixel nicht verändert werden darf, also das auf dem fertigen Bild auch die Pixel zu sehen sind die dem Betrachter am nähesten sind.
Wichtig dabei ist diese einzelnen Schritte direkt auf der Grafikkarte durchzuführen. Um die einzelnen Oberflächenteile zu bestimmen, wird ein *DepthPeeling* durchgeführt. Diese Technik wurde in [Eve02] vorgestellt. Für die Klassifizierung auf der GPU werden mehrere Stenciloperationen verwendet. Zusätzlich wird eine spezielle Normalform des CSG Ausdrucks benötigt - Die Blist-Normalform. Das Zusammenfügen des fertigen Bildes kann durch eine Stenciloperation oder einen speziellen Pixelshader durchgeführt werden. Diese einzelnen Techniken werden im Folgenden genau beschrieben und erläutert.

5.2 Der *DepthPeeling* Ansatz

Das *DepthPeeling* ist eine Technik die eine beliebige Szene in Tiefenschichten zerlegt. Dabei enthält die erste Schicht genau die Oberflächenteile die dem Betrachter am nähesten sind. Jede folgende Schicht enthält genau die Oberflächenteile die dem Betrachter am nähesten sind, aber hinter der vorhergehenden Schicht liegen. Somit können bei einem vollständigen *DepthPeel* alle Oberflächenteile dargestellt werden. Zusätzlich sind die einzelnen Schichten nach der Tiefe sortiert, womit es auch möglich ist die Schichten wieder zu einem korrekten Bild zusammenzusetzen.
In den Abbildungen 7,8,9 und 10 wird solch eine Zerlegung für eine einfache Szene dargestellt.
In den Abbildungen 11, 12, 13 und 14 wird solch eine Zerlegung für eine komplexe Szene dargestellt. Diese Szene besitzt je nach Blickwinkel bis zu 10 Schichten. Zur Umsetzung dieser Technik werden zwei Tiefenpuffer benötigt. Der erste Tiefenpuffer dient als Z-Buffer und sorgt dafür das immer die Pixel zu sehen sind die dem Betrachter am nähesten sind. Der zweite Puffer enthält die Tiefe der Pixel aus der vorhergehenden Schicht. Dieser wird im Folgenden als *Referenzpuffer* bezeichnet. Zur Umsetzung des DepthPeelings dürfen beim Renderen einer Schicht nur die Pixel akzeptiert werden deren Tiefe größer als die Tiefe im Referenzpuffer ist. Das Problem dabei ist das übliche Grafikkarten keine zwei Tiefenpuffer besitzen. Eine Ausnahme bilden die GeForce FX Karten (oder höhere Versionen) von NVIDIA. Diese

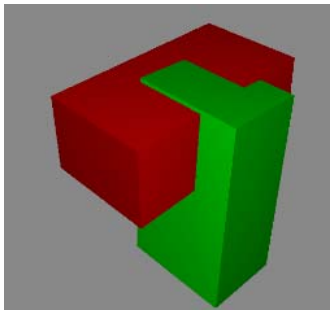

Abbildung 7: *DepthPeel* Schicht 1

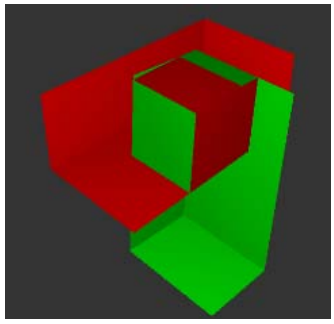

Abbildung 8: *DepthPeel* Schicht 2

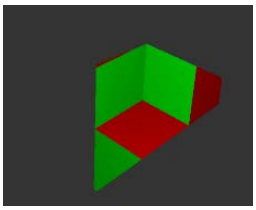

Abbildung 9: Schicht 3

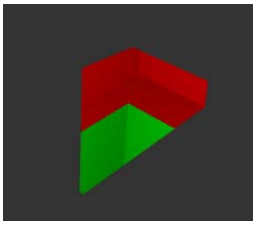

Abbildung 10: Schicht 4

haben die Fähigkeit einen Tiefenpuffer als Textur zu verwenden. Da vom Referenzpuffer nur gelesen wird, reicht es aus diesen mithilfe einer Textur darzustellen. Das Auslesen des Tiefenwertes aus der Textur kann bei OpenGL mithilfe eines sehr kurzen Fragment Programms umgesetzt werden. Dieses kann in den Quelldateien zu [Eve02] gefunden werden. Bei DirectX ist dies nicht ganz so einfach. Die Dokumentation sieht zwar Tiefentexturen vor, aber beim Ausführen von Texturoperationen auf eine derartige Textur tritt ein undokumentiertes Verhalten auf.[5] Trotzdem lässt sich ein entsprechender Pixelshader konstruieren der aber nicht so exakte Ergebnisse liefert wie die OpenGL Variante. Der entsprechende Shader[6] ist als Anlage A auf Seite 21 beigelegt. Der folgende Algorithmus in PseudoCode(DirectX Orientiert) beschreibt das genaue Vorgehen zur Darstellung eines *DepthPeels*:

- Initialisierung
  - lege die Tiefentextur(en) an
  - lade den Pixelshader für den Referenzpuffervergleich (im Folgenden als TestDepth bezeichnet)
- Darstellung der Schicht i
  - setDepthSurface(DepthTextureSchicht[i])
  - setColorSurface(ColorSurfaceSchicht[i])
  - clearSurfaces()
  - beginRendering()
  - setTexture(DepthTextureSchicht[i-1])
  - setPixelShader(TestDepth)
  - renderSzene()
  - endrendering()

Da diese Technik nur auf GeForce FX Karten oder moderneren GeForce Karten umgesetzt werden kann, ist ein DepthPeeling auf anderen Grafikkarten nur begrenzt möglich. Auf anderen Karten kann ein ähnlicher Ansatz verwendet werden, nur das anstatt einer Tiefentextur ein Farbpuffer als Textur verwendet wird. Mithilfe eines Pixelshaders kann dann ein Tiefenwert in diesen Farbpuffer geschrieben werden, dabei kommt es allerdings zu einem Genauigkeitsverlust, da in einem Tiefenpuffer bis zu 24 bit für einen Tiefenwert zur Verfügung stehen und in einem Farbpuffer jeweils 8 bit pro Farbe zur Verfügung stehen. Dies ergibt zwar jeweils 24 nutzbare bit bei beiden Puffern, aber es ist kaum verlustfrei möglich einen 24 bit Wert innerhalb eines Pixelshaders in drei 8 bit Werte umzuwandeln. Weiterhin würde bei diesem Ansatz ein zusätzlicher Renderpass benötigt werden[7] der die Geschwindigkeit zusätzlich noch senken würde.

[5] Bei DirectX 9.0c und einer GeForce 6800 Go
[6] Als DirectX Assembler Shader
[7] Zum Speichern der Tiefe in einem Farbpuffer

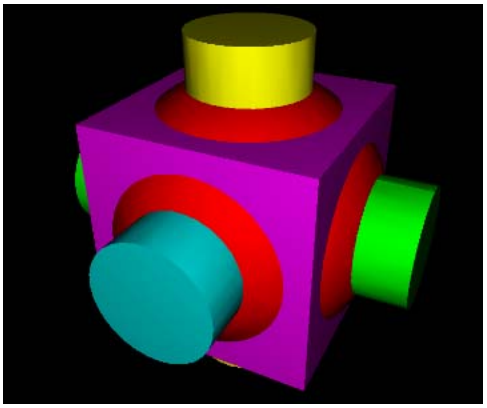

Abbildung 11: *DepthPeel* Schicht 1

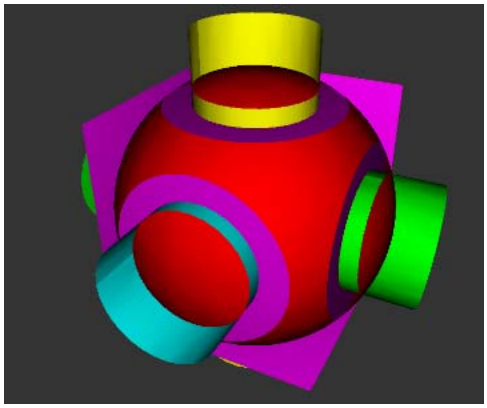

Abbildung 12: *DepthPeel* Schicht 2

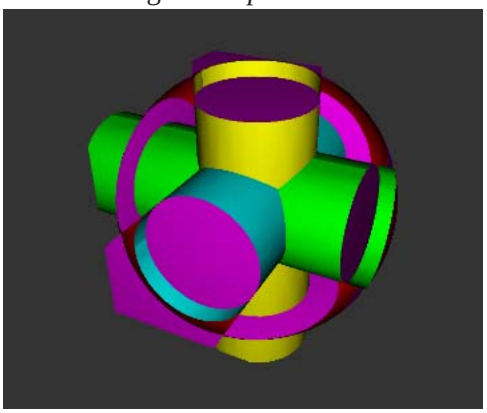

Abbildung 13: *DepthPeel* Schicht 3

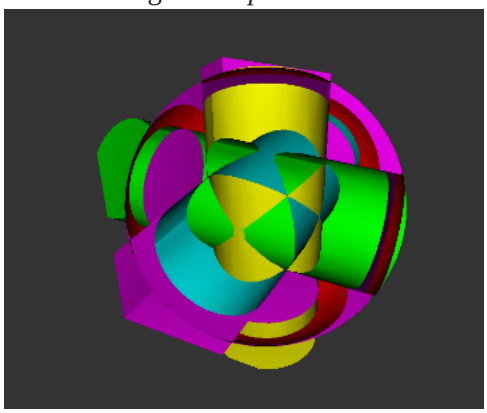

Abbildung 14: *DepthPeel* Schicht 4

5.3 Klassifikation bei Blister

5.3.1 Ein Klassifikationsbeispiel

Im Folgenden soll die Blist-Normalform motiviert werden. Ziel dieser ist es einen CSG Ausdruck möglichst einfach und schnell auszuwerten. Einfach bedeutet, dass das Auswerten auf einer GPU[8] realisierbar sein muss. Es ist offensichtlich das der in Abschnitt 4.3 auf Seite 5 vorgestellte Algorithmus nicht einfach genug ist. Dies liegt vorallem an der nötigen Rekursion. Das Ziel ist diesen Algorithmus zu linearisieren. Dazu soll hier zunächst eine spezielle Variante des Algorithmus vorgestellt werden. Diese Variante nutzt die feste Abarbeitungsreihenfolge *Left*[9]*-Order*[10]*-Right*[11].
Nun soll dieser modifizierte Algorithmus auf das Beispiel aus Abschnitt 2.2 auf Seite 2 angewandt werden. Allerdings nicht auf den Original CSG Ausdruck *Y*, sondern auf eine Positive-Normalform von *Y*. Die Idee dabei ist das in einer Positiven-Normalform Negationen nur über Primitiven auftreten und alle Differenzen ersetzt werden. Damit verringert sich die Anzahl an Fällen die betrachtet werden müssen. Dabei gibt es dann nur noch die Operationen Schnitt und Differenz(jeweils nicht negiert). Primitive können dabei negiert oder auch nicht negiert auftreten. Zur Umwandlung in Positive-Normalform kann der

[8] Prozessor einer Grafikkarte
[9] *left* = in die linke rekursive Verzweigung gehen
[10] *order* = die Operation dieses Knotens ausführen
[11] *right* = in die rechte rekursive Verzweigung gehen(wenn nötig)

folgende Algorithmus benutzt werden:

$$pos(Y) = \begin{cases} \big(pos(Z1) \cup pos(Z2)\big) & \text{wenn } Y \text{ von der form } (Z1 \cup Z2) \text{ ist.} \\ \big(pos(Z1) \cap pos(Z2)\big) & \text{wenn } Y \text{ von der form } (Z1 \cap Z2) \text{ ist.} \\ \big(pos(Z1) \cap pos(\neg Z2)\big) & \text{wenn } Y \text{ von der form } (Z1 - Z2) \text{ ist.} \\ \big(pos(\neg Z1) \cap pos(\neg Z2)\big) & \text{wenn } Y \text{ von der form } \neg(Z1 \cup Z2) \text{ ist.} \\ \big(pos(\neg Z1) \cup pos(\neg Z2)\big) & \text{wenn } Y \text{ von der form } \neg(Z1 \cap Z2) \text{ ist.} \\ \big(pos(\neg Z1) \cup pos(Z2)\big) & \text{wenn } Y \text{ von der form } \neg(Z1 - Z2) \text{ ist.} \\ pos(Z1) & \text{wenn } Y \text{ von der form } \neg\neg Z1 \text{ ist.} \\ Y & \text{wenn } Y \text{ von der form } \neg P \text{ oder } P \text{ und } P \text{ ein Primitiv ist.} \end{cases}$$

Mithilfe dieses Algorithmus kann nun unsere Beispielgleichung *Y* in Positive-Normalform umgewandelt werden:

$$\begin{aligned} pos(Y) &= pos\big(((((A \cap B) \cap C) \cup D) - ((E \cup F) \cap G))\big) \\ &= (pos\big((((A \cap B) \cap C) \cup D)\big) \cap pos\big(\neg((E \cup F) \cap G))\big) \\ &= \ldots \text{ Auswertung der linken Seite(keine Änderungen da dort keine Negationen)} \\ &= ((((A \cap B) \cap C) \cup D) \cap pos\big(\neg((E \cup F) \cap G))\big) \\ &= ((((A \cap B) \cap C) \cup D) \cap (pos\big(\neg(E \cup F)\big) \cup pos\big(\neg G\big))) \\ &= ((((A \cap B) \cap C) \cup D) \cap ((pos\big(\neg E\big) \cap pos\big(\neg F\big)) \cup \neg G)) \\ &= ((((A \cap B) \cap C) \cup D) \cap ((\neg E \cap \neg F) \cup \neg G)) \\ &= (ABC \cup D) \cap (\neg E \neg F \cup \neg G) \text{ vereinfachte Schreibweise} \\ &= Y' \end{aligned}$$

Diese Positive-Normalform kann nun als Eingabe für den modifizierten Klassifikationsalgorithmus genutzt werden. Der betrachtete Pixel soll innerhalb der Primitive *A,B,E* und somit außerhalb der Primitive *D,C,F,G* liegen. Um diese Auswertung möglichst übersichtlich zu halten, wird wo möglich die vereinfachte Schreibweise verwendet:

$$\begin{aligned} klass(Y,p) &= klass\big((ABC \cup D) \cap (\neg E \neg F \cup \neg G), p\big) \\ &= klass\big((ABC \cup D), p\big) \cap klass\big((\neg E \neg F \cup \neg G), p\big) \\ &= (klass\big(ABC, p\big) \cup klass\big(D, p\big)) \cap klass\big((\neg E \neg F \cup \neg G), p\big) \\ &= ((klass\big(AB, p\big) \cap klass\big(C, p\big)) \cup klass\big(D, p\big)) \cap klass\big((\neg E \neg F \cup \neg G), p\big) \\ &= (((\mathbf{klass}\big(\mathbf{A}, \mathbf{p}\big) \cap klass\big(B, p\big)) \cap klass\big(C, p\big)) \cup klass\big(D, p\big)) \cap klass\big((\neg E \neg F \cup \neg G), p\big) \\ &= (((IN \cap klass\big(B, p\big)) \cap klass\big(C, p\big)) \cup klass\big(D, p\big)) \cap klass\big((\neg E \neg F \cup \neg G), p\big) \\ &= ((\mathbf{klass}\big(\mathbf{B}, \mathbf{p}\big) \cap klass\big(C, p\big)) \cup klass\big(D, p\big)) \cap klass\big((\neg E \neg F \cup \neg G), p\big) \\ &= ((IN \cap klass\big(C, p\big)) \cup klass\big(D, p\big)) \cap klass\big((\neg E \neg F \cup \neg G), p\big) \\ &= (\mathbf{klass}\big(\mathbf{C}, \mathbf{p}\big) \cup klass\big(D, p\big)) \cap klass\big((\neg E \neg F \cup \neg G), p\big) \\ &= (OUT \cup klass\big(D, p\big)) \cap klass\big((\neg E \neg F \cup \neg G), p\big) \\ &= \mathbf{klass}\big(\mathbf{D}, \mathbf{p}\big) \cap klass\big((\neg E \neg F \cup \neg G), p\big) \\ &= OUT \cap klass\big((\neg E \neg F \cup \neg G), p\big) \\ &= OUT \end{aligned}$$

Die interessante Beobachtung bei dieser Klassifizierung ist, dass obwohl der Algorithmus rekursiv ist eigentlich eine lineare Auswertung stattfand. Dabei wurde zuerst das Primitiv A ausgewertet und danach die Primitive B bis D, in ihrere Reihnfolge. Dies funktioniert bei beliebigen CSG Ausdrücken. Eine zweite Beobachtung ist das nicht alle Primitive gegen p getestet wurden (z.B. G wurde nicht gegen p getestet).

Somit ist es bei der Klassifizierung eines Pixels möglich die Primitive von links beginnend auszuwerten und je nach Resultat der einzelnen Klassifikationen, Teile des CSG Ausdrucks zu überspringen. Die Abbildung 15 veranschaulicht dies indem sie alle möglichen Klassifikationsvarianten eines Pixels gegen die Beispielgleichung *Y* darstellt. Die im obigen Beispiel veranschaulichte Klassifikation entspricht dem grünen Pfad in der Abbildung. Die Abbildung 15 verdeutlicht zusätzlich noch das es bei jedem Test, ob

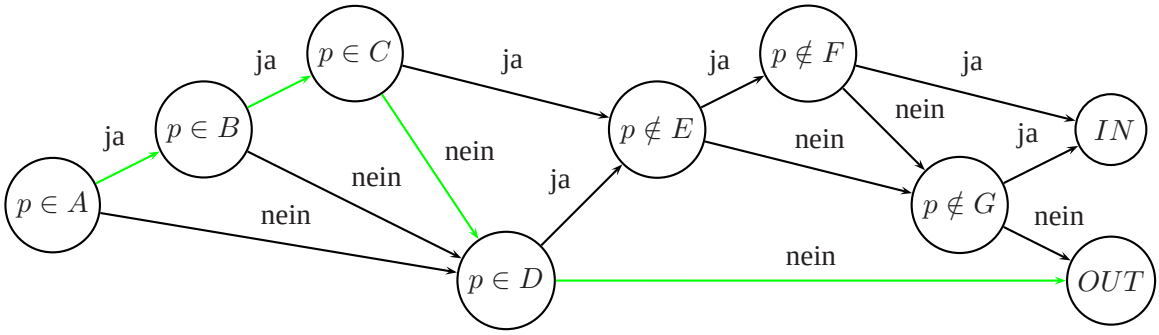

Abbildung 15: Alle Klassifikationsvarianten für $Y = (ABC \cup D) - (E \cup F)G)$

ein Pixel in einem Primitiv liegt oder nicht, nur zwei Entscheidungsvarianten gibt. Die eine führt zum rechten Nachbarn des Primitivs und die andere zu einem weiter entfernten Primitiv. Dieses weiter entfernte Primitiv liegt aber immer rechts des Primitivs gegen das getestet wird. Somit ist es möglich einen Pixel zu klassifizieren indem alle Primitive eines CSG Ausdrucks von links nach rechts gegen *p* getestet werden. Unter Umständen werden dabei aber Primitive ausgelassen. Dieser Ansatz wird im folgenden weiter ausgebaut und formalisiert.

5.3.2 Match & Flip

Die Klassifikation auf der GPU soll nach folgendem Prinzip geschehen:

- Für jeden zu klassifizierenden Pixel wird eine ID gespeichert.
- Diese ID identifiziert das Primitiv gegen das der Pixel als nächstes getestet werden muss.
- Anfangs wird diese ID mit der ID des ersten Primitivs eines CSG Ausdrucks initialisiert.
- Zur Klassifikation werden dann alle Primitive des CSG Ausdrucks von links nach rechts mit folgenden Regel durchgegangen:
  - Wenn der Pixel die ID des aktuell betrachteten Primitives(*P*) hat, dann ersetze die ID des Pixels entweder mit der ID des rechten Nachbarprimitives von P oder mit der ID eines weiter entfernten Primitivs.
    In solch einem Fall ist das Primitiv *aktiv für p*
  - Wenn der Pixel eine andere ID beinhaltet, dann verändere die ID des Pixels nicht.
    In solch einem Fall ist das Primitiv *nicht aktiv für p*

Dieses Vorgehen beschreibt eine Wanderung auf einem Pfad von Abbildung 15. Dabei muss allerdings noch beschrieben werden, wie sich die ID‘s ergeben durch die ersetzt wird und wann welche der beiden Entscheidungsvarianten gewählt wird. Dazu werden die folgenden Funktionen eingeführt:

Sei *P* ein Primitiv aus einem CSG Ausdruck *Y*, dann ist:

- *next(P)* der rechte Nachbar von *P* in *Y*
- *match(P)* das im vorhergehenden Abschnitt beschriebene Primitiv das in Y weiter rechts von *next(P)* liegt

- *flip(P)* ein binäres Flag das angibt wann *next(P)* bzw *match(P)* zu betrachten ist
- *rest(P)* derjenige reduzierte CSG Ausdruck der zu betrachten ist, wenn das Primitiv *P* für einen Pixel aktiv ist.

Die Berechnung von *next(P)* ist einfach. Bei der Berechnung von *match(P)* wird allerdings ein etwas komplexerer Ansatz benötigt. Dazu soll vorher *rest(P)* genau beschrieben werden. Diese Funktion beschreibt den CSG Ausdruck der noch abzuarbeiten ist wenn das Primitiv *P* aktiv wird. Da die Primitive von links nach rechts abgearbeitet werden, fallen somit alle Primitive im CSG Ausdruck weg die links von *P* liegen. Mit dieser Vorschrift kann *rest(P)* bestimmt werden. Bei der Bestimmung von *match(P)* ist es sinnvoll den Fall, dass ein weiter entferntes Primitiv ausgewählt wird genau zu betrachten. Dabei fällt auf, dass dieses immer aus der mit dem Primitiv verbundenen Booleschen-Operation innerhalb von *rest(P)* hinausführt. Dazu einige Beispiele:

- $Y = AB \cup C$, $match(A)$ ist das Primitiv C. Da $next(A) = B$ ist und dieses aktiv wird wenn der Pixel in A liegt. Somit muss $match(A)$ aktiv werden wenn der Pixel nicht in A liegt. In diesem Fall ist das Ergebniss der Auswertung von B uninteressant, da der Teilausdruck AB bzw. $A \cap B$ unabhängig von B als Ergebniss OUT liefert. Somit ist das nächste zu betrachtende Primitiv C also $match(A) = C$. Betrachtet man diesen CSG Ausdruck als Baum so entspricht die Verbindung von A und $match(A)$ den Blau eingezeichneten Pfad in Abbildung 16.

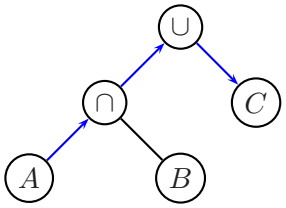

Abbildung 16: Pfad zur Bestimmung von $match(A)$ im CSG Ausdruck $Y = AB \cup C$

- $Y = ABC \cup D$, $match(B)$ ist das Primitiv D, da $rest(B) = BC \cup D$ ist und dies genau dem vorhergehenden Beispiel entspricht.
- $Y = (ABC) \cap D$, $match(A)$ ist das Primitiv D. Da $next(A) = B$ ist und dieses aktiv wird wenn der Pixel in A liegt. Somit muss $match(A)$ aktiv werden, wenn der Pixel nicht in A liegt. In diesem Fall ist das Ergebniss der Auswertung von B bzw. des Teilausdrucks BC uninteressant, da der Teilausdruck ABC bzw. $(A \cap B) \cap C$ unabhängig von B, C als Ergebniss OUT liefert (wegen $klass(A, p) = OUT$). Somit ist das nächste zu betrachtende Primitiv D, also $match(A) = D$. Betrachtet man diesen CSG Ausdruck als Baum so entspricht die Verbindung von A und $match(A)$ den Blau eingezeichneten Pfad in Abbildung 17.

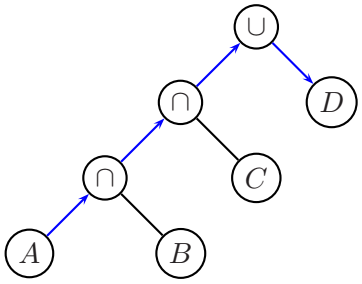

Abbildung 17: Pfad zur Bestimmung von $match(A)$ im CSG Ausdruck $Y = ABC \cup D$

- $Y = (A \cup B)C$, $match(A)$ ist das Primitiv C. Da $next(A) = B$ ist und dieses aktiv wird, wenn der Pixel nicht in A liegt. Somit muss $match(A)$ aktiv werden, wenn der Pixel in A liegt. In diesem Fall ist das Ergebniss der Auswertung von B uninteressant, da der Teilausdruck $A \cup B$ unabhängig von B als Ergebniss IN liefert (wegen $klass(A, p) = IN$). Somit ist das nächste zu betrachtende Primitiv C also $match(A) = C$. Betrachtet man diesen CSG Ausdruck als Baum so entspricht die Verbindung von A und $match(A)$ den Blau eingezeichneten Pfad in Abbildung 18.

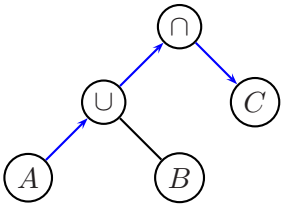

Abbildung 18: Pfad zur Bestimmung von $match(A)$ im CSG Ausdruck $Y = (A \cup B)C$

- $Y = ((A \cup B) \cup C)D$, $match(B)$ ist das Primitiv D, da $rest(B) = (B \cup C)D$ ist und dies genau dem vorhergehendem Beispiel entspricht.
- $Y = ((A \cup B) \cup C)D$, $match(A)$ ist das Primitiv D. Da $next(A) = B$ ist und dieses aktiv wird, wenn der Pixel nicht in A liegt. Somit muss $match(A)$ aktiv werden, wenn der Pixel in A liegt. In diesem Fall ist das Ergebniss der Auswertung von B bzw. des Teilausdrucks $B \cup C$ uninteressant, da der Teilausdruck $(A \cup B) \cup C$ unabhängig von B, C als Ergebniss IN liefert (wegen $klass(A, p) = IN$). Somit ist das nächste zu betrachtende Primitiv D also $match(A) = D$. Betrachtet man diesen CSG Ausdruck als Baum so entspricht die Verbindung von A und $match(A)$ den Blau eingezeichneten Pfad in Abbildung 19.

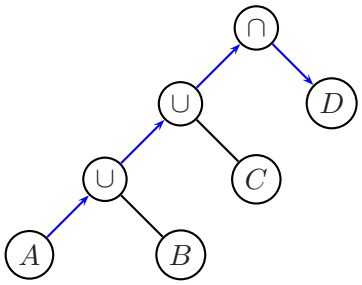

Abbildung 19: Pfad zur Bestimmung von $match(A)$ im CSG Ausdruck $Y = ((A \cup B) \cup C)D$

Von diesen Beispielen lässt sich eine Vorgehensweise zur Bestimmung von $match(P)$, eines Primitives P in einem CSG Ausdruck Y, ableiten. Zuerst wird $rest(P)$ bestimmt. Danach wird im zugehörigen Baum zu $rest(P)$ die Operation des Vaterknotens gespeichert. Nun muss im Baum solange nach oben gewandert werden bis ein Knoten mit einer anderen Operation als der gespeicherten gefunden wird. Von dieser Operation aus muss im rechten Teilbaum das Primitiv gefunden werden, welches am weitesten links in diesem Teilbaum steht. In Pseudocode lautet der Algorithmus somit wie folgt:

```
CSGExpression match(CSGExpression P) {
    CSGExpression Q;
    Operation Op;

    Op = P.father.Op;
    Q = P.father;

    while (Q.Op == Op)
        Q = Q.father;

    Q = Q.rightChild();
    while (!isPrimitive(Q))
        Q = Q.leftChild();

    return(Q);
    }
```

Wichtig bei diesem Algorithmus ist, dass er für das Primitiv P mit dem Baum $rest(P)$ aufgerufen wird. Dies erzeugt zusätzlichen Aufwand der durch eine Modifikation des Algorithmuses vermieden werden kann. Dabei muss am Anfang des Algorithmus noch solange nach oben gewandert werden bis ein Knoten erreicht ist der ein linker Nachfolger seines Vaterknotens ist.
Nun muss noch $flip(P)$ formalisiert werden. Dazu sei P ein Primitiv eines CSG Ausdrucks Y und p ein Pixel. An den obigen Beispielen war zu sehen das je nach CSG Ausdruck entweder bei $p \in P$ zu $match(P)$ oder bei $p \notin P$ zu $match(P)$ gegangen werden musste. Die Entscheidung wann $match(P)$ als neue ID eingetragen werden muss, wird von $flip(P)$ bestimmt. Dazu wird die Bedeutung von $flip(P)$ folgendermaßen festgelegt:

- Wenn $flip(P) = false$ und $p \in P$ ist, dann wird als neue ID $next(P)$ eingetragen.
- Wenn $flip(P) = false$ und $p \notin P$ ist, dann wird als neue ID $match(P)$ eingetragen.
- Wenn $flip(P) = true$ und $p \in P$ ist, dann wird als neue ID $match(P)$ eingetragen.
- Wenn $flip(P) = true$ und $p \notin P$ ist, dann wird als neue ID $next(P)$ eingetragen.

Anhand der obigen Beispiele und der Semantik con ***flip(P)*** ergibt sich das $flip(P) = true$ ist, wenn die Operation des Vaterknotens in $rest(P)$ die Vereinigung ($\cup$) ist. Wenn die Operation des Vaterknotens von P in $rest(P)$ der Schnitt ($\cap$)ist, so ist dementsprechend $flip(P) = false$. Zusätzlich muss $flip(P)$ noch invertiert werden wenn P ein negiertes Primitiv ist. Somit ergibt sich der folgende Algorithmus für die Bestimmung von $flip(P)$:

```
boolean flip(CSGExpression P) {
    Operation Op;
    Op = P.father.Op;
    boolean Erg;

    if (Op == OPERATION_UNION) //Vereinigung
        Erg = true;
    else Erg = false;

    if (P.isNegated())
        Erg = !Erg;

    return(Erg);
    }
```

Auch hier ist wieder zu beachten das der Algorithmus für die Bestimmung von $flip(P)$ nur auf $rest(P)$ angewandt werden darf. Analog zum Algorithmus zur Bestimmung von $match(P)$ kann auch hier eine entsprechende Erweiterung gemacht werden um nicht erst $rest(P)$ bestimmen zu müssen. Des Weiteren können beide Algorithmen miteinander verbunden werden.

Ein Problem dieser Algorithmen ist, dass sie z.B. für den CSG Ausdruck $Y = A$ kein Ergebniss liefern würden. Da das Primitiv A keinen Vaterknoten hätte. Dies liegt daran, dass dies ein Sonderfall wäre bei dem $next(A) = IN$, $match(A) = OUT$ und $flip(A) = false$ wäre. Um eine Erweiterung der Algorithmen zu vermeiden, kann jeder CSG Ausdruck Y zu einem neuen CSG Ausdruck Y' wie folgt erweitert werden:

- $Y' = (Y \cup OUT) \cap IN$

Diese Erweiterung ist mit den Algorithmen und den Definitionen von $match(P)$, $next(P)$ und $flip(P)$ verträglich. (Alternativ wäre auch $Y' = (Y \cap IN) \cup OUT$ möglich.)

P	$rest(P)$	$next(P)$	$match(P)$	$flip(P)$
A	$(ABC \cup D)(\neg E \neg F \cup \neg G)$	B	D	$false$
B	$(BC \cup D)(\neg E \neg F \cup \neg G)$	C	D	$false$
C	$(C \cup D)(\neg E \neg F \cup \neg G)$	D	E	$true$
D	$D(\neg E \neg F \cup \neg G)$	E	OUT	$false$
E	$(\neg E \neg F \cup \neg G)$	F	G	$true$
F	$(\neg F \cup \neg G)$	G	IN	$false$
G	G	OUT	IN	$false$

Tabelle 1: $match(P)$, $next(P)$, $flip(P)$ und $rest(P)$ für alle Primitive von $Y = (ABC \cup D)(\neg E \neg F \cup \neg G)$

Für das Beispiel aus Abschnitt 4.3 auf Seite 5 sind in Tabelle 5.3.2 die hier definierten Funktionen für alle Primitive angegeben.

Somit ist es gelungen einen beliebigen CSG Ausdruck in eine Form zu bringen die es ermöglicht alle Primitive durchzulaufen und dabei pro Pixel die folgenden Operationen durchzuführen:

- Wenn die gespeicherte ID des Pixels die ID des Primitivs ist, dann prüfe ob der Pixel im Primitiv liegt und ersetze anhand von $next(P)$, $match(P)$ und $flip(P)$ die ID des Pixels.
- Sonst tue nichts.

Diese Algorithmik ist *einfach* genug um auf der GPU ausgeführt zu werden. Allerdings müssen die ID‘s auf der GPU mit wenigen bit an Information dargestellt werden.

5.3.3 Minimale ID‘s

Bei der Klassifikation auf der GPU werden sogenannte *Stenciloperationen* verwendet. Dabei stehen 8 bit Speicher pro Pixel zur Verfügung. Von diesen 8 bit werden später 2 bit für spezielle Funktionen benötigt. Somit bleiben 6 bit zur Darstellung der ID eines Primitivs, welche jeweils pro Pixel gespeichert werden muss. Mit diesen 6 bit lassen sich $2^6 = 64$ verschiedene ID‘s darstellen. Da jeweils eine ID für IN und OUT benötigt wird, bleiben insgesammt noch 62 ID‘s für Primitive. Wenn jedes Primitiv eine ID zugewiesen bekommt, ließen sich nur CSG Ausdrücke mit bis zu 62 Primitiven darstellen. Dies wäre eine enorme Einschränkung.
Um diese Einschränkung zu umgehen, ist es möglich ID‘s wiederzuverwenden. Dabei muss gelten, dass $next(P)$ und $match(P)$ jeweils eindeutig bestimmt sind. Da aber alle Primitive von links nach rechts durchlaufen werden, ist es möglich ID‘s von Primitiven die bereits abgearbeitet wurden wiederzuverwenden. Dabei gilt eine ID I kann für ein Primitiv P wiederverwendet werden, wenn es kein Primitiv links von P gibt mit $match(P) = I$, wobei $match(P)$ rechts von P liegt. Diese Definition kann in einem einfachen Algorithmus umgesetzt werden. Dieser wurde in [JH05] vorgestellt. Dabei wird eine Liste mit belegten ID‘s gespeichert. Der Algorithmus läuft folgendermaßen ab:

```
//Initialisierung:
    List L;
    L.empty();
    //Primitiv(i) referenziert i.tes Primitiv im CSG Ausdruck
    Primitiv(0).Id = 0;
    L.push_key(0);
//Durchlauf
    for (int i = 0; i < numPrimitives;i++)
    {
        //gebe die ID dieses Primitivs frei
```

```
        L.remove_key(Primitiv(i).Id);

        //reserviere die nierigste freie ID f. next(P)
        int id = L.min_free_key();
        Primitiv(i).pNext->Id = id;
        L.push_key(id);

        //reserviere die niedrigste freie ID f. match(P)
        id = L.min_free_key();
        Primitiv(i).pMatch->Id = id;
        L.push_key(id);
    }
```

Mithilfe dieser Algorithmik ist es laut [JH05] möglich CSG Audrrücke mit mindestens 3909 Primitiven darzustellen. Wobei bei den meisten CSG Ausdrücken mehr Primitive dargestellt werden können.

5.3.4 Resultat der Vereinfachungen

Durch die vorhergehenden Berechnungen und Umformungen ist es gelungen die rekursive Klassifikation zu linearisieren. Dabei kann ein CSG Ausdruck nun als eine Liste von Primitiven dargestellt werden bei der jedes Primitiv zwei Zeiger besitzt ($match(P)$ und $next(P)$). Das Resultat der Linearisierung für das Beispiel aus 2.2 auf Seite 2 ist in Abbildung 20 dargestellt. Dabei sind auch die ID‘s der einzelnen Primitive enthalten. Obwohl der CSG Ausdruck 7 Primitive enthält werden nur 3 verschiedene ID‘s benötigt. In der Abbildung wird $match(P)$ bei $flip(P) = false$ als roter Pfeil und bei $flip(P) = true$ als grüner Pfeil dargestellt.

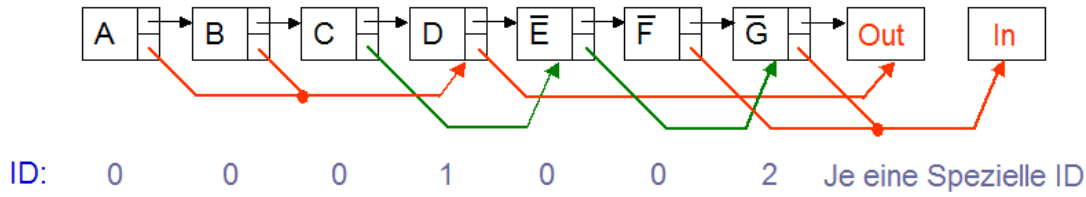

Abbildung 20: Resultat der linearisierung für den CSG Ausdruck $Y = (ABC \cup D)(\neg E \neg F \cup \neg G)$

5.3.5 GPU Klassifikation

Zur Klassifikation auf der GPU werden *Stenciloperationen* verwendet. Diese Operationen verwenden einen speziellen Teil des Depth-Puffers. Dabei stehen auf modernen Grafikkarten bis zu 32 Bit Depth-Puffer zur Verfügung, dieser wird meist so aufgeteilt das von diesen 32 bit, 24 bit als Z-Puffer verwendet werden und 8 bit für Stenciloeprationen zur Verfügung stehen. Diese 8 bit sind pro Pixel verfügbar. Somit kann für jeden Pixel 8 bit Information gespeichert werden. Mit diesen bits können zwei verschiedene Operationen durchgeführt werden:

I Test zum verwerfen von Pixeln (Ähnlich dem Tiefentest) mit der Regel:
Pixel darf gezeichnet werden, wenn *(StencilRef* $\vee$ *StencilMask) CompFunc (StencilBufferValue* $\vee$ *StencilMask)* erfüllt ist (aus [Mic05])

II Modifikation des Pufferinhaltes mit der Regel:
NewStencilBufferValue = (StencilBufferValue $\wedge \neg$*StencilWriteMask)* $\vee$ *(StencilWriteMask* $\wedge$ *StencilOP(StencilBufferValue))* (aus [Mic05])

Dabei haben die einzelnen Platzhalter die folgenden Bedeutungen:

StencilRef ein 8 bit Referenzwert der gesetzt werden kann. (ist bei einem Renderpass für alle Pixel gleich)

StencilMask eine Maske die beim Stenciltest gesetzt werden kann um nur bestimmte Pixel zu testen

CompFunc eine Stenciltest-Operation (entspricht einer Relation), dabei gibt es die Relationen $<$, $\leq$, $>$, $\geq$, $=$, $\neq$ sowie *immer akzeptieren* und *nie akzeptieren*

StencilBufferValue der Stencil-Pufferwert für den aktuellen Pixel

NewStencilBufferValue der neue Stencil-Pufferwert für den aktuellen Pixel

StencilWriteMask eine Maske die festlegt welche bits des Stencil-Puffers verändert werden dürfen

StencilOP eine Operation die beim Bilden des neuen Stencil-Pufferwerts genutzt und auf den alten Wert angewendet wird
Es gibt die Operationen:

- alten Wert behalten
- neuer Wert = 0
- alten Wert durch *StencilRef* ersetzen
- inkrementieren ohne Überlauf
- dekrementieren ohne Überlauf
- inkrementieren mit möglichem Überlauf
- dekrementieren mit möglichem Überlauf
- bitweise invertieren

Die beiden Anwendungsmöglichkeiten des Stencilpuffers genügen um die Klassifikation auf der GPU durchzuführen. Dabei wird ein Stencilbit benötigt um festzustellen ob ein Pixel innerhalb eines Primitivs liegt. Dieses bit wird als *parity* bezeichnet. Ein weiteres bit wird als temporärer Speicher genutzt um zu markieren ob die ID eines Pixels der ID des aktuellen Primitivs entspricht. Dieses bit wird als *utility* bezeichnet. Dieses ist nötig, da beim Schreiben einer neuen ID überprüft werden muss, ob die ID des Pixels der ID des aktuellen Primitives entspricht. Weitherin muss die ID des nächsten Primitivs gesetzt werden, dies führt dazu das in solch einem Fall ein *StencilRef*-Wert zum Vergleich und ein *StencilRef*-Wert zum Schreiben nötig wäre. Diese beiden Werte sind im Allgemeinen unterschiedlich somit kann dieser Schritt nicht mit einer einzelnen Stencil-Operation umgesetzt werden, wodurch das utility bit genutzt werden muss. Die restlichen 6 bit werden zum Speichern der ID genutzt.
Um festzustellen ob ein Pixel innnerhalb eines Primitvs liegt ist die folgende Regel wichtig: *Jeder Strahl der durch ein Primitiv führt hat eine gerade Anzahl von Schnittpunkten mit dem Primitiv*. Dies gilt da alle Primitive solide sind. Zum Feststellen ob ein Pixel in einem Primitiv liegt, kann ein Verfahren angewandt werden das z.B. in [SZ04] vorgestellt wurde. Dabei wird das *parity* bit auf 0 gesetzt und anschließend das Primitiv gerendert. Das *parity* bit eines Pixels p wird invertiert, wenn ein Pixel gezeichnet wird der dem Betrachter näher gelegen ist als p. Diese Z-Puffer Entscheidung erzeugt einen Strahl der durch p und der Kameraposition verläuft. Somit gilt wenn nach dem Rendervorgang $parity = 1$ ist, dann waren ungerade viele Pixel vor p, was bedeutet das p auf einem Strahl innerhalb des Primitivs liegt. Dieser Strahl wird von dem Pixel q, das vor p liegt und diesem Pixel am nähesten ist, und einem Pixel r hinter p aufgespannt. Der Pixel r muss wegen obiger Bedingung existieren. Somit folgt das p im Primitiv liegt. Bei $parity = 0$ liegt p außerhalb des Primitivs.
Zur Klassifikation gegen ein einzelnes Primitiv P auf der GPU müssen somit die folgenden Schritte mit Stenciloperationen umgesetzt werden:

```
setze pixel.utility = 0

wenn pixel.id == id(P)
    pixel.utility = 1
    pixel.parity = 0
```

```
setColorBufferWriteEnable(false)
setZBufferWriteEnable(false)
render(P) mit
    pixel.parity = !pixel.parity wenn Pixel Z-Puffertest besteht


wenn flip(P) == false
    wenn pixel.utility == 1 und pixel.parity == 1
        setze pixel.id = next(P)
    wenn pixel.utility == 1 und pixel.parity == 0
        setze pixel.id = match(P)

wenn flip == false
    wenn pixel.utility == 1 und pixel.parity == 1
        setze pixel.id = match(P)
    wenn pixel.utility == 1 und pixel.parity == 0
        setze pixel.id = next(P)
```

Eine beispielhafte Umsetzung dieser Operationen für DirectX ist als Anlage B auf Seite 22 zu finden.

In Abbildung 21 und 22 sind die Schichten 1 und 3 für das Beispiel aus Abbildung 7 dargestellt (anderer Blickwinkel!). Die Abbildungen 23 und 24 zeigen die Schichten 2 und 4 des Beispiels aus Abbildung 11 (anderer Blickwinkel!).

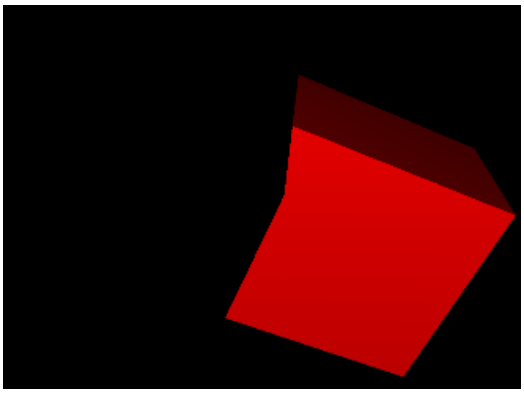

Abbildung 21: Klassifiziertes *DepthPeel* Schicht 1

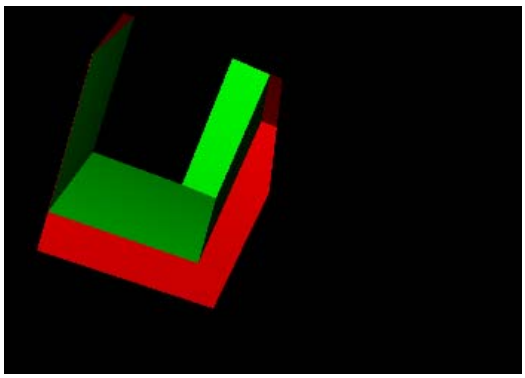

Abbildung 22: Klassifiziertes *DepthPeel* Schicht 3

5.4 Zusammenführung der Schichten

Im vorhergehenden Abschnitt wurde beschrieben wie die Klassifikation einer Schicht eines *DepthPeels* auf der GPU ausgeführt werden kann. Das Objekt zu einem CSG Ausdruck entsteht indem alle diese klassifizierten Schichten zusammengefügt werden. Hierzu gibt es 2 Ansätze:

I Speichern der Depth-Puffer aller klassifizierten Schichten und kopieren von diesen Puffern mithilfe einer Stenciloperation.

II Auf allen Schichten die Pixel die die ID *OUT* haben mit einem *ColorKey* überschreiben. Anschließend kopieren der Pixel der einzelnen Schichten, wobei alle Pixel mit dem ColoKey ausgelassen werden.

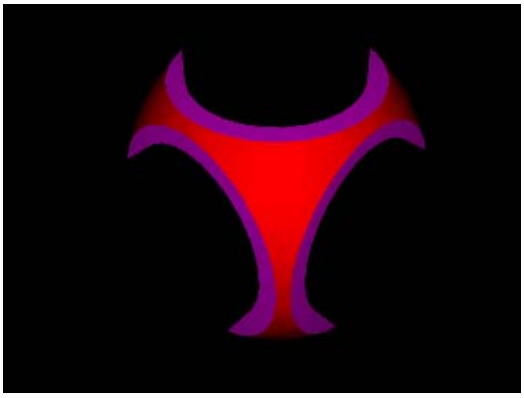

Abbildung 23: Klassifiziertes *DepthPeel* Schicht 2

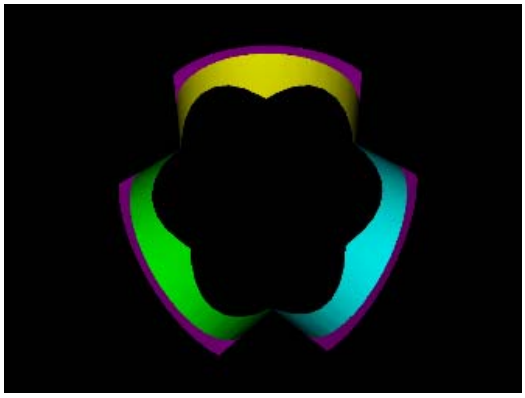

Abbildung 24: Klassifiziertes *DepthPeel* Schicht 4

Dabei muss jeweils die Tiefenreihenfolge berücksichtigt werden und somit mit der letzten Schicht beim kopieren begonnen werden.
In Abbildung 25 ist das fertige Objekt für den CSG Ausdruck aus Abbildung 7 dargestellt. Das fertige Objekt für das Beispiel aus Abbildung 11 ist in Abbildung 26 dargestellt.

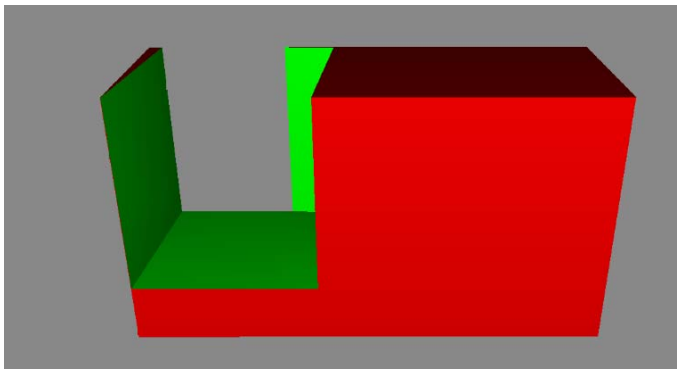

Abbildung 25: Darstellung eines CSG Ausdrucks mit Blister

5.5 Mögliche Erweiterungen und Optimierungen

Die in den vorhergehenden Abschnitten beschriebenen Techniken können noch um folgende Details erweitert werden:

- Beim Klassfizieren auf der GPU müssen die einzelnen Stenciloperationen auf die einzelnen Pixel angewandt werden. Anstatt diese Operationen auf alle Pixel anzuwenden ist es möglich aktive Bereiche zu berechnen und die Stenciloperationen nur auf diese Bereiche anzuwenden.
- Durch die tiefen Sortierung der einzelnen Schichten des *DepthPeels* ist es auch möglich Objekte mit Transparenz darzustellen.
- Bei den Stenciloperationen ist es mögich das utility bit wegzulassen, indem eine OpenGL-Extension[12] verwendet wird die ein *XOR* als Stenciloperation zur Verfügung stellt.
- Es ist möglich zusätzlich ein Shadowmapping für ein CSG Objekt zu konstruieren.
- CSG Bäume können umgeformt werden um die Anzahl der benötigten ID‘s zu reduzieren.

[12] Erweiterung von OpenGL die treiberspezifische(Grafikkarte) Operationen anbietet

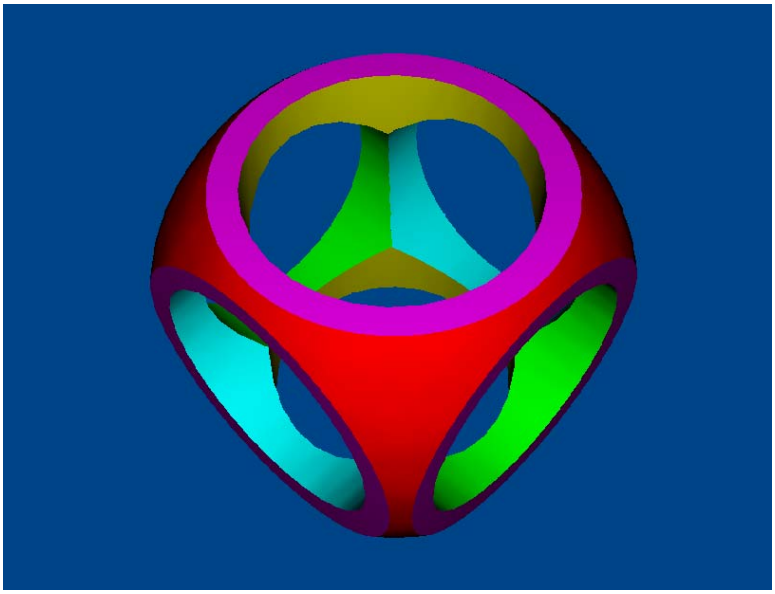

Abbildung 26: Darstellung eines CSG Ausdrucks mit Blister

6 Zusammenfassung und Ausblick

Es wurde ein Verfahren zur Darstellung von CSG Ausdrücken vorgestellt. Dabei wurde nicht explizit die Hülle des resultierenden CSG Objekts berechnet, sondern eine Blickwinkel abhängige Darstellung die den CSG Ausdruck *per frame* auf der Grafikkarte auswertet. Zur Umsetzung wurde die Blist-Normalform verwendet die einen CSG Ausdruck linearisiert. Der Renderer der diese Blistnormalform nutzt nennt sich Blister und arbeitet nach folgendem Prinzip:

Initialisierung: Bilde die Blist-Normalform

Darstellung:

- Bestimme das Depth Peel der Schicht i.
- Klassifiziere die Schicht i indem jedes Primitiv mit der in Abschnitt 5.3.5 vorgestellten Algorithmik ausgewertet wird.
- Füge die einzelnen Schichten zusammen.

Das Verfahren hat eine Komplexität von $O(n * k)$ wobei n die Anzahl der Primitive angibt und k die Anzahl der benötigten Tiefenschichten angibt. In den meisten Fällen wächst die Anzahl der Tiefenschichen langsamer als die Anzahl der Primitive. Somit hängt die Komplexität von den am CSG Ausdruck beteiligten Primitiven ab. Das hierzu benötigte *DepthPeeling* kann bisher nur auf GeForce FX Karten(oder höher) umgesetzt werden. Zusätzlich wurde eine eigene DirecX Umsetzung der Kernalgorithmen von Blister vorgestellt. Diese Algorithmen sind Bestandteil einer einfachen eigen entwickelten DirectX Umsetzung von Blister. Alle Abbildungen mit CSG Objekten, Tiefenschichten oder klassifizierten Schichten wurden mithilfe dieser Implementation dargestellt.

Literatur

[Eve02] EVERITT, C. *Interactive order-independent transparency. Tech Report, nVidia Corporation*. `http://developer.nvidia.com`. 2002

[GJ86] GOLDFEATHER J., Hultquist J. Fuchs H. *Fast constructive solid geometry display in the pixel-powers graphics system*. Annual Conference on Computer Graphics and Interactive Techniques, 107-116. 1986

[JH05] JOHN HABLE, Jarek R. *Blister: GPU-based rendering of Boolean combinations of free-form triangulated shapes*. `http://www.johnhable.com/Blister/Blister.pdf`. 2005

[Mic05] MICROSOFT, Corporation. *DirectX Documentation for C++*. DirectX 9.0 SDK Update August 2005: Stencil Buffer Techniques. 2005

[RA85] REQUICHA A, Voelcker H. *Boolean operations in solid modeling: Boundary evaluation and merging algorithms*. Procedings of the IEEE, 75, 1, 30-44. 1985

[SZ04] STEFAN ZERBST, Eike A. *3D Spiele Programmierung*. Markt+Technik, München. 2004

[Til84] TILOVE, R.B. *A Null-Object Detection Algorithm for Constructive Solid Geometry*. Communications of the ACM 27, 7, 684-694. 1984

A DepthPeel Pixelshader für DirectX

```
ps.2.x
//line for 1024*768!!!
//def c11,0.5,-0.5,0.500001,0.500001 //x,y for biasing x,y coordinates (
    mult). z,w for second biasoperation (add) the 0.000001 is for geting the
     corect pixel and dependent upon resolution
//line for 1440*900
def c11,0.5,-0.5,0.5000012,0.5000012 //x,y for biasing x,y coordinates (
    mult). z,w for second biasoperation (add) the 0.000001 is for geting the
     corect pixel and dependent upon resolution
def c12,-0.000001,0.5,0.0,0.0

//the color from the vertex shader
dcl t0
//the position of this pixel
dcl t1
//the depth texture sampler
dcl_2d s0

//calculate t1.w=1.0/t1.w
rcp r1.w,t1.w

//calculate r1.x=t1.x/t1.w, r1.y=t1.y/t1.w, r1.z=t1.z/t1.w
mul r1.xyz,t1.xyz,r1.w

//make r1.xy as texture coordinates r1.x = (0.5*r1.x)+0.5, r1.y = (0.5*r1.y
    )+0.5 (values between 0 and 1)
mul r1.xy,r1.xy,c11.xy
add r1.xy,r1.xy,c11.zw

//add some tolerance to the z value---(otherwise problems)
add r1.z,c12.x,r1.z

//load the depthtexture value at r1.xy (compares texture->depth with r1.z
    result is 0 or 1)
texld r2,r1,s0

//do the compare
//if r2.g == 0 --->> r1.z > texture[r1.xy].depth (pixel is OK, behind the
    last pixel in the depth texture)
//if r2.g == 1 --->> r1.z < texture[r1.xy].depth (pixel is not OK, in front
     of the last pixel in the depth texture)
sub r2.x,c12.y,r2.g
texkill r2

//calculate the resulting color
//the following is just an example calculating the color will differ from
    case to case!
mul r2,c10,t0
mov r2.w,c10.w
mov oC0,r2
```

B Code fragment für die Umsetzung der Stenciloperationen unter DirectX

```
//=============================================================
// 0. Init von p und utility
//=============================================================

//setze compare mask f. ID
m_renderDevice->GetDynamicRenderer()->SetRenderState(RS_STENCILMASK,0x3F);

//wenn stencil vergleich und ZVergleich bestanden sind
m_renderDevice->GetDynamicRenderer()->SetRenderState(RS_STENCILPASS,
    RDSTENCILOP_REPLACE);

//setzt die Stencil test function
m_renderDevice->GetDynamicRenderer()->SetRenderState(RS_STENCILFUNC,
    RDCMP_EQUAL);

//setzt den neuen Wert
m_renderDevice->GetDynamicRenderer()->SetRenderState(RS_STENCILREF,(DWORD)
    (0x40|m_bID));

//setzt die writemask
//Mask: [1][1][0][0][0][0][0][0] = 0xC0
m_renderDevice->GetDynamicRenderer()->SetRenderState(RS_STENCILWRITEMASK,0
    xC0);

//render now
renderFullscreenQuad (matView,vPos);

//=============================================================
// 1. Operation bestimmen von p
//=============================================================

//setze Maske die nur auf ID begrenzt
//Mask: [0][1][0][0][0][0][0][0] = 0x3F
m_renderDevice->GetDynamicRenderer()->SetRenderState(RS_STENCILMASK,0x40);

//wenn stencil vergleich und ZVergleich bestanden sind
m_renderDevice->GetDynamicRenderer()->SetRenderState(RS_STENCILPASS,
    RDSTENCILOP_INVERT);

//setzt die Stencil test function
m_renderDevice->GetDynamicRenderer()->SetRenderState(RS_STENCILFUNC,
    RDCMP_EQUAL);

//setzt den vergleichswert
m_renderDevice->GetDynamicRenderer()->SetRenderState(RS_STENCILREF,0x40);

//setzt die writemask
//Mask: [1][0][0][0][0][0][0][0] = 0x80
m_renderDevice->GetDynamicRenderer()->SetRenderState(RS_STENCILWRITEMASK,0
    x80);

//render now
m_renderDevice->GetDynamicRenderer()->SetRenderState(RS_ZBUFFERENABLE,true)
    ;
m_pModell->RenderDynRen (1);
```

```
//=============================================================
// 2.Operation bestimmen von naechster ID (wenn P = 0, also Pixel
    ausserhalb des Objektes)
//=============================================================

//Mask: [1][1][0][0][0][0][0][0] = 0xC0
m_renderDevice->GetDynamicRenderer()->SetRenderState(RS_STENCILMASK,0xC0);

//wenn Stencil vergleich und ZVergleich gegluekt sind
m_renderDevice->GetDynamicRenderer()->SetRenderState(RS_STENCILPASS,
    RDSTENCILOP_REPLACE);

//setzt die Stencil test function
m_renderDevice->GetDynamicRenderer()->SetRenderState(RS_STENCILFUNC,
    RDCMP_EQUAL);

if (!m_bFlip)
{
    //setzt den vergleichswert
    m_renderDevice->GetDynamicRenderer()->SetRenderState(RS_STENCILREF,(
        DWORD)(0x40|this->m_bMatch));
}
else
{
    //setzt den vergleichswert
    m_renderDevice->GetDynamicRenderer()->SetRenderState(RS_STENCILREF,(
        DWORD)(0x40|this->m_bNext));
}

//setzt die writemask
//Mask: [1][1][1][1][1][1][1][1] = 0xFF
m_renderDevice->GetDynamicRenderer()->SetRenderState(RS_STENCILWRITEMASK,0
    xFF);

//render einmal den ganzen Bildschirm
renderFullscreenQuad (matView,vPos);

//=============================================================
// 3.Operation bestimmen von naechster ID (wenn P = 1, also Pixel in diesem
    Objekts)
//=============================================================

//Mask: [1][1][0][0][0][0][0][0] = 0xC0
m_renderDevice->GetDynamicRenderer()->SetRenderState(RS_STENCILMASK,0xC0);

//wenn Stencil vergleich und ZVergleich gegluekt sind
m_renderDevice->GetDynamicRenderer()->SetRenderState(RS_STENCILPASS,
    RDSTENCILOP_REPLACE);

//setzt die Stencil test function
m_renderDevice->GetDynamicRenderer()->SetRenderState(RS_STENCILFUNC,
    RDCMP_EQUAL);

if (!m_bFlip)
{
    //setzt den vergleichswert
```

```
    m_renderDevice->GetDynamicRenderer()->SetRenderState(RS_STENCILREF,(
        DWORD)(0xC0|this->m_bNext));
}
else
{
    //setzt den vergleichswert
    m_renderDevice->GetDynamicRenderer()->SetRenderState(RS_STENCILREF,(
        DWORD)(0xC0|this->m_bMatch));
}

//setzt die writemask
//Mask: [1][1][1][1][1][1][1][1] = 0xFF
m_renderDevice->GetDynamicRenderer()->SetRenderState(RS_STENCILWRITEMASK,0
    xFF);

//render einmal den ganzen Bildschirm
renderFullscreenQuad (matView,vPos);

//=============================================================
// 4.Reset von Utility and Parity (3. u. 4. erzeugt (utility = 1 und je
    nach Fall parity = 1)
//=============================================================

//setze compare mask fuer ID
m_renderDevice->GetDynamicRenderer()->SetRenderState(RS_STENCILMASK,0xFF);

//wenn stencil vergleich und ZVergleich geglueckt sind
m_renderDevice->GetDynamicRenderer()->SetRenderState(RS_STENCILPASS,
    RDSTENCILOP_REPLACE);

//setzt die Stencil test function
m_renderDevice->GetDynamicRenderer()->SetRenderState(RS_STENCILFUNC,
    RDCMP_ALWAYS);

//setzt den neuen Wert
m_renderDevice->GetDynamicRenderer()->SetRenderState(RS_STENCILREF,(DWORD)
    (0x00));

//setzt die writemask
//Mask: [1][1][0][0][0][0][0][0] = 0xC0
m_renderDevice->GetDynamicRenderer()->SetRenderState(RS_STENCILWRITEMASK,0
    xC0);

//render now
renderFullscreenQuad (matView,vPos);
```